Spring Security 参考手册

Ben Alex · Luke Taylor · Rob Winch · Gunnar Hillert

Table of Contents

Spring security 是一个强大的和高度可定制的身份验证和访问控制框架。它是确保基于Spring的应用程序的标准。

前言

Spring Security 为基于javaEE的企业应用程序提供一个全面的解决方案。正如你将从这个参考指南发现的，我们试图为你提供一个有用的并且高度可配置的安全系统。

安全是一个不断移动的目标，采取一个全面的全系统的方法很重要。在安全领域，我们鼓励你采取"layers of security"(安全层)，这样每一层尽可能的在自己范围内诶保证安全，连续的层提供额外的安全性。安全层更密集你的程序将更加健壮更加安全。在最底层，你需要处理传输安全和系统识别这些问题，以减少中间人攻击。接下来，你讲通常利用防火墙，或许使用VPN或者IP担保以确保只有授权的系统能够尝试连接。在企业环境中你可以部署一个DMZ将面向公众的服务器和数据库以及应用服务器分隔开来。你的操作系统也将扮演重要的部分，解决问题，类似，使用非特权用户运行进程和提高文件系统安全性。操作系统通常会配置自己的防火墙。但愿在某处前进的道路上，你会试图阻止拒绝服务和暴力攻击。一个入侵检测系统将在监视和相应攻击时非常有用，这种系统能采取保护动作，比如实时阻断违规的TCP/IP地址。在更高的层，你的java虚拟机希望被配置为尽量减少不同的java类型授予的权限，然后将你的应用程序增加到器自身的制定域特定的安全配置。Spring Security 使后者 ，应用程序将更加安全更加容易。

当然你需要妥善处理上面提到的所有安全层，连同各层的管理因素。这样的管理因素答题包括安全公告监测、补丁、人员审查、审计、变更控制、工程管理系统、数据备份、灾难恢复、性能基准测试、负载监控、集中式日志记录、事件相应程序等。

Spring Secruity 致力于在企业应用程序安全层对你进行帮助，你会发现这里有如此不同的需求正如业务问题的领域。一个银行应用程序具有与电子商务应用不同的需求。电子商务应用程序同企业销售自订花工具具有不同的需求。这些定制需求使得应用安全有趣、有挑战性和有回报。

请阅读入门第二部分，从“入门”开始。 这一章将为你介绍的框架和基于命名空间的配置系统，你可以配置好应用，来了解Spring Security 如何工作和一些你可能 需要使用的类 。你需要阅读 架构与实现。本指南的其余部分是更传统的引用样式结构，设计用于按需进行阅读。我们也建议你尽可能于都完一般的应用安全问题。Spring Secruity 不是万能的，不能解决所有的安全问题。重要的实在应用设计开始之初就考虑到安全性。后期改造不是一个好的主意。特别是，如果你正在构件一个Web应用程序，你应该知道许多潜在的漏洞，比如跨站脚本、请求伪造和会话劫持。这些你一开始就应该考虑。这个网站 (http://www.owasp.org/) 维护了一个大网站应用漏洞列表和一些有用的参考信息。

我希望这个参考手册对你来说比较有用，欢迎你的反馈和建议。 suggestions.

最后欢迎你来到Spring Security 社区。 community.

入门

本指南的稍后张杰会对框架的架构和实现类进行一个深度的讨论，如果你的相对Spring Security进行一个深度定制没这一章节将会包含你需要了解的内容。在本章我们将会介绍Spring Security 3.0 给项目的历史进行简要的概述，简单的讲讲如何开始使用设个框架。尤其是我们将看看命名空间配置，他提供与传统Spring Bean你必须连接所有实现类的途径更简单的方式保护你的应用程序。

我们也会看看实例应用。在你阅读后面的章节之前你指的试着运行体验它。当时你对框架连接更多的时候你还可以汇过来回顾一下。 project website 同时请参阅项目的网站，因为他有创建这个项目的有用的信息，以及文章、视频和教程。

简介

Spring Security是什么?

Spring Security 提供了基于javaEE的企业应有个你软件全面的安全服务。这里特别强调支持使用SPring框架构件的项目，Spring框架是企业软件开发javaEE方案的领导者。如果你还没有使用Spring来开发企业应用程序，我们热忱的鼓励你仔细的看一看。熟悉Spring特别是一来注入原理两帮助你更快更方便的使用Spring Security。

人们使用Spring Secruity的原因有很多，单大部分都发现了javaEE的Servlet规范或EJB规范中的安全功能缺乏典型企业应用场景所需的深度。提到这些规范，重要的是要认识到他们在WAR或EAR级别无法移植。因此如果你更换服务器环境，这里有典型的大量工作去重新配置你的应用程序员安全到新的目标环境。使用Spring Security 解决了这些问题，也为你提供许多其他有用的，可定制的安全功能。

正如你可能知道的两个应用程序的两个主要区域是“认证”和“授权”（或者访问控制）。这两个主要区域是Spring Security 的两个目标。“认证”，是建立一个他声明的主题的过程（一个“主体”一般是指用户，设备或一些可以在你的应用程序中执行动作的其他系统）。“授权”指确定一个主体是否允许在你的应用程序执行一个动作的过程。为了抵达需要授权的店，主体的身份已经有认证过程建立。这个概念是通用的而不只在Spring Security中。

在身份验证层，Spring Security 的支持多种认证模式。这些验证绝大多数都是要么由第三方提供，或由相关的标准组织，如互联网工程任务组开发。另外Spring Security 提供自己的一组认证功能。具体而言，Spring Security 目前支持所有这些技术集成的身份验证：

- HTTP BASIC 认证头 (基于 IETF RFC-based 标准)
- HTTP Digest 认证头 (IETF RFC-based 标准)
- HTTP X.509 客户端证书交换 (IETF RFC-based 标准)
- LDAP (一个非常常见的方法来跨平台认证需要, 尤其是在大型环境)
- Form-based authentication (用于简单的用户界面)
- OpenID 认证
- Authentication based on pre-established request headers (such as Computer Associates Siteminder) 根据预先建立的请求有进行验证
- JA-SIG Central Authentication Service (CAS，一个开源的SSO系统)
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (Spring远程协议)
- Automatic "remember-me" authentication (你可以勾选一个框以避免预定的时间段再认证)
- Anonymous authentication (让每一个未经验证的访问自动假设为一个特定的安全标识)
- Run-as authentication (在一个访问应该使用不同的安全标识时非常有用)
- Java Authentication and Authorization Service (JAAS)
- JEE container autentication (所以如果愿你以可以任然使用容器管理的认证)
- Kerberos
- Java Open Source Single Sign On (JOSSO) *
- OpenNMS Network Management Platform *
- AppFuse *
- AndroMDA *

- Mule ESB *
- Direct Web Request (DWR) *
- Grails *
- Tapestry *
- JTrac *
- Jasypt *
- Roller *
- Elastic Path *
- Atlassian Crowd *
- Your own authentication systems (see below)
- 表示由第三方提供

很多独立软件供应商，因为灵活的身份验证模式二选择Spring Security。这样做允许他们快速的集成到他们的终端客户需求的解决方案而不用进行大量工程或者改变客户的环境。如果上面的验证机制不符合你的需求，Spring Security 是一个开放的平台，要实现你 自己的验证机制检查。Spring Security 的许多企业用户需要与不遵循任何安全标准的“遗留”系统集成，Spring Security可以很好的与这类系统集成。

无论何种身份验证机制，Spring Security提供一套的授权功能。这里有三个主要的热点区域，授权web请求、授权方法是否可以被调用和授权访问单个域对象的实例。为了帮助让你分别了解这些差异，认识在Servlet规范网络模式安全的授权功能，EJB容器管理的安全性和文件系统的安全。Spring Security在这些重要的区域提供授权功能，我们将在手册后面进行介绍。

历史

Spring Security 以“The Acegi Secutity System for Spring” 的名字始于2013年晚些时候。一个问题提交到Spring 开发者的邮件列表，询问是否已经有考虑一个机遇Spring 的安全性社区实现。那时候Spring 的社区相对较小（相对现在）。实际上Spring自己在2013年只是一个存在于ScourseForge的项目，这个问题的回答是一个值得研究的领域，虽然目前时间的缺乏组织了我们对它的探索。

考虑到这一点，一个简单的安全实现建成但是并没有发布。几周后，Spring社区的其他成员询问了安全性，这次这个代码被发送给他们。其他几个请求也跟随而来。到2014年一月大约有20万人使用了这个代码。这些创业者的人提出一个SourceForge项目加入是为了，这是在2004三月正式成立。

在早些时候，这个项目没有任何自己的验证模块，身份验证过程依赖于容器管理的安全性和Acegi安全性。而不是专注于授权。开始的时候这很适合，但是越来越多的用户请求额外的容器支持。容器特定的认证领域接口的基本限制变得清晰。还有一个相关的问题增加新的容器的路径，这是最终用户的困惑和错误配置的常见问题。

Acegi安全特定的认证服务介绍。大约一年后，Acegi安全正式成为了Spring框架的子项目。1.0.0最终版本是出版于2006 -在超过两年半的大量生产的软件项目和数以百计的改进和积极利用社区的贡献。

Acegi安全2007年底正式成为了Spring组合项目，更名为"Spring Security"。

如今Spring Security有一个强大的和积极的开源社区。在支持论坛上有成千上万的关于Spring Security的消息。有一个活跃的核心的开发人员，他们的代码本身和一个活跃的社区，也经常分享补丁和支持他们的同龄人。

发布版本号

了解Spring Security发布版本号如何工作是很有用的，他可以帮助你识别出工作（或缺乏的功能）设计到参与迁移到项目的未来版本，每个发布使用3个整数，MAJOR.MINOR.PATCH（主版本、次要版本、补丁版本）.这样做的目的是主版本是不兼容的，API大范围的升级。次要版本应该保留大部分源代码和二进制兼容旧版本的次要版本，认为可能有一些设计变更和不兼容的更新。补丁版本应该向前向后完美兼容。包含一些bug和缺陷修复这些意外的改变。

在某种程度上，你受到变化的影响取决于你的代码是如何紧密集成的。如果你正在做大量定制，你更可能受到比简单的命名空间配置更大的影响。

你应该总是推出一个新版本之前彻底测试你的应用程序。

Getting Spring Security

你可以通过几种方式获取Spring Security。你可以从 Spring Security 页面下载一个分发包。从Maven库下载分离的jar文件。另外你也可以从源代码自己编译。

使用Maven

一个最小的SPring Security Maven 依赖通常和下面的类似:

pom.xml

```
  <dependencies>
<!-- ... other dependency elements ... -->
<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-web</artifactId>
 <version>{spring-security-version}</version>
</dependency>
<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-config</artifactId>
 <version>{spring-security-version}</version>
</dependency>
</dependencies>
```

如果你使用的额外的功能比如LDAP，OpenID，等等，你需要包含适当的模块，查阅Section 1.4.3,"Project Modules" 项目模块.

Maven 仓库

所有GA发布版本（版本号以.RELEASE结尾）都被部署到Maven Central ，所以不需要在你的pom里设置额外的库

如果你使用了一个 SNAPSHOT 版本，你需要确认你设置了Snapshot库，如下:

pom.xml

```
  <repositories>
<!-- ... possibly other repository elements ... -->
<repository>
 <id>spring-snapshot</id>
 <name>Spring Snapshot Repository</name>
 <url>http://repo.spring.io/snapshot</url>
</repository>
</repositories>
```

如果你正在使用一个里程碑或者发布候选版本，你需要确保你所定义的Spring里程碑库，如下图所示:

pom.xml

```
  <repositories>
<!-- ... possibly other repository elements ... -->
<repository>
 <id>spring-milestone</id>
 <name>Spring Milestone Repository</name>
 <url>http://repo.spring.io/milestone</url>
</repository>
</repositories>
```

Spring 框架 Bom

Spring Security是针对Spring 框架 4.1.6RELEASE 建立的{spring-version}, 但是快要与4.0.x正常工作。很多用户都会有这个问题，Spring Security 传递依赖解析到Spring框架4.1.6 RELEASE这可能导致奇怪的classpath问题。{spring-version}

规避这个问题的一种方式是在你的pom文件的 <dependencyManagement> 的部分.另一种办法是像下面这样在 `pom.xml` 中包含 `spring-framework-bom` `<dependencyManagement>` :

pom.xml

```
  <dependencyManagement>
 <dependencies>
 <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-framework-bom</artifactId>
  <version>{spring-version}</version>
  <type>pom</type>
  <scope>import</scope>
 </dependency>
 </dependencies>
</dependencyManagement>
```

这样将确保Spring Security传递的所有依赖都使用 Spring 4.16.RELEASE {spring-version} 模块.

这种但是使用了 Maven’s "bill of materials" (BOM) 概念， Maven 2.0.9+可用. 如果想了解Maven如何解析依赖请参考Maven的依赖机制文档的介绍。 Maven’s Introduction to the Dependency Mechanism documentation.

Gradle

最小的Gradle一组典型的依赖如下:

build.gradle

```
 dependencies {
compile 'org.springframework.security:spring-security-web:{spring-security-version}'
compile 'org.springframework.security:spring-security-config:{spring-security-version}'
}
```

如果你使用了 LDAP, OpenID, 等等. 你需要包含相应的模块，请参阅“Project Modules”项目模块.

Gradle Repositories

所有的GA版本 (以.RELEASE结尾)都部署到了Maven Central库, 所以使用mavenCentral()库就足够了 .

build.gradle

```
 repositories {
mavenCentral()
}
```

如果你正在使用SNAPSHOT版本，你需要确保你使用了Spring的Snapshot库，定义如下:

build.gradle

```
 repositories {
maven { url 'https://repo.spring.io/snapshot' }
}
```

如果你正在使用里程碑或者发布候选版。你需要确认你使用了SPring的里程碑仓库，定义如下:

build.gradle

```
 repositories {
maven { url 'https://repo.spring.io/milestone' }
}
```

使用 Spring 4.0.x 和 Gradle

解析依赖传递是Gradle默认会使用最新的版本。这意为这当在Spring4.1.6RELEASE 下使用Spring Security 4.0.2RELEASE我们不需要做额外的工作。 {spring-security-version} {spring-version} 然而有时出现一些问题，最好适应Gradle的ResolutionStrategy 按照下面这种方式处理 Gradle’s ResolutionStrategy:

build.gradle

```
 configurations.all {
resolutionStrategy.eachDependency { DependencyResolveDetails details ->
 if (details.requested.group == 'org.springframework') {
  details.useVersion '{spring-version}'
 }
}
}
```

这样将确保Spring Security 传递的依赖使用Spring 4.1.6.RELEASE {spring-version}模块.

> 这个例子使用 Gradle 1.9,但可能需要在gradle的未来版本中进行修改，因为这是在一个Gradle的孵化功能.

项目模块

在 Spring Security 3.0, 代码库被分割到单独的jar，这样可以更清楚的分隔不用功能区域和第三方依赖。如果你使用Maven构建你的项目，那么这些都是需要你添加到你的 `pom.xml` 中的. 甚至你没有使用Maven，我们建议你请求 `pom.xml` 文件来获取第三方依赖和版本。另一个好办法是检查实例应用程序的库。

核心模块 - spring-security-core.jar

包含核心的验证和访问控制类和接口，远程支持和基本的配置API。任何使用Spring Security的应用程序都需要这个模块。支持独立应用程序、远程客户端、服务层方法安全和JDBC用户配置。包含以下顶层包:

- `org.springframework.security.core`
- `org.springframework.security.access`
- `org.springframework.security.authentication`
- `org.springframework.security.provisioning`

远程调用 - spring-security-remoting.jar

提供与Spring Remoting的集成，通常你不需要这个模块，除非你挣使用Spring Remoting编写远程客户端。主要的包是 `org.springframework.security.remoting` .

网页 - spring-security-web.jar

包括锅炉汽和网站安全相关的基础代码。使用Servlet API的任何应用依赖他。如果你需要Spring Security网页验证服务和基于URL的访问控制你需要这个模块。主包名为 `org.springframework.security.web`.

配置 - spring-security-config.jar

包含安全命令空间的解析代码。如果你使用Spring Security XML命令空间进行配置你需要包含这个模块。 主包名为`org.springframework.security.config`. 没有类需要被应用程序直接使用.

LDAP - spring-security-ldap.jar

LDAP验证和配置代码，如果你需要使用LDAP验证和管理LDAP用户实体，你需要这个模块。主包名为`org.springframework.security.ldap`.

ACL访问控制表 - spring-security-acl.jar

ACL专门的领域对象的实现。用来在你的应用程序中应用安全特定的领域对象实例。主包名为`org.springframework.security.acls`.

CAS - spring-security-cas.jar

Spring Security的CAS客户端集成。如果你想用CAS的SSO服务器使用Spring Security网页验证需要该模块。顶层的包是`org.springframework.security.cas`.

OpenID - spring-security-openid.jar

OpenID 网页验证支持。使用外部的OpenID服务器验证用户。 `org.springframework.security.openid`. 需要OpenID4Java.

Test - spring-security-test.jar

支持Spring Security的测试。

签出源代码

因为Spring Security是一个开源项目，我们强烈建议您使用Git签出源代码。这可以让你完全访问所有势力应用程序，你可以轻松地构建最先进的最新版本的项目。有了项目的源代码对debug有非常大的帮助。异常堆栈跟踪不再是模糊的黑盒问题，你可以沿着他找出到底发生了什么。源代码是中级文档，通常也是最简答的项目实际是如何工作的地方。

要获取项目源代码，请使用一下git命令:

```
git clone https://github.com/spring-projects/spring-security.git
```

这可以让你在本地机器上访问到整个项目的历史记录（包括所有版本和分支）.

Spring Security 4.1新特性

Spring Security 4.1 新特性有关的链接 100+ RC1 issues and 60+ RC2 issues 这里是新特性的列表:

Java 配置提升

- Simplified UserDetailsService Java Configuration
- Simplified AuthenticationProvider Java Configuration
- 通过 `LogoutConfigurer` 配置内容谈判 `LogoutSuccessHandler` (s)
- 通过 `SessionManagementConfigurer` 可配置 `InvalidSessionStrategy`
- 使用 `HttpSecurity.addFilterAt` 在链中的特定位置添加一个 `Filter` 的能力

Web应用程序安全性提升

- MvcRequestMatcher
- Content Security Policy (CSP)
- HTTP Public Key Pinning (HPKP)
- CORS
- CookieCsrfTokenRepository 提供简单的 AngularJS 与 CSRF 整合
- 添加 `ForwardAuthenticationFailureHandler` 和 `ForwardAuthenticationSuccessHandler`
- AuthenticationPrincipal 支持表达式属性从而支持转换 `Authentication.getPrincipal()` 对象 (即处理不可变的自定义的 `User` 域对象)

授权改进

- Path Variables in Web Security Expressions
- Method Security Meta Annotations

密码模块的改进

- SCrypt 支持 `SCryptPasswordEncoder`
- PBKDF2 支持 Pbkdf2PasswordEncoder
- BouncyCastle 中新改进的 `BytesEncryptor` 使用 *AES/CBC/PKCS5Padding* 和 *AES/GCM/NoPadding* 算法

测试的改进

- @WithAnonymousUser
- @WithUserDetails 允许指定 `UserDetailsService` bean 名称
- Test Meta Annotations
- 模拟一个列表的能力 `GrantedAuthority` 使用 `SecurityMockMvcResultMatchers.withAuthorities`

一般的改进

- 重新组织样本项目
- 移动到GitHub的问题

样品和指南 (Start Here)

如果您想从Spring Security开始，最好的从我们的示例应用程序开始。

Table 1. Sample Applications

源	描述	指南
{gh-samples-url}/javaconfig/helloworld[Hello Spring Security]	演示如何整合Spring Security和基于java的配置的现有应用程序。	链接:../../guides/html5/helloworld-javaconfig.html[Hello Spring Security Guide]
{gh-samples-url}/boot/helloworld[Hello Spring Security Boot]	演示如何整合Spring Security和 Spring Boot应用程序。	链接:../../guides/html5/helloworld-boot.html[Hello Spring Security Boot Guide]
{gh-samples-url}/xml/helloworld[Hello Spring Security XML]	演示如何整合Spring Security和基于XML配置使用的现有应用程序。	链接:../../guides/html5/helloworld-xml.html[Hello Spring Security XML Guide]
{gh-samples-url}/javaconfig/hellomvc[Hello Spring MVC Security]	演示如何整合Spring Security和Spring MVC 应用程序。	链接:../../guides/html5/hellomvc-javaconfig.html[Hello Spring MVC Security Guide]
{gh-samples-url}/javaconfig/form[Custom Login Form]	演示如何创建自定义表单。	链接:../../guides/html5/form-javaconfig.html[Custom Login Form Guide]

Java 配置

java配置的支持主要在Spring框架的3.1加入. Java Configuration. Spring Security 从3.2开始加入java配置支持。这让用户不使用任何XML用更简单方式配置Spring Security。

如果你熟悉第四章，安全命名空间配置，安全命名空间配置 你应该找到它和java的安全配置支持之间的相当多的相似之处.

Spring Security 提供大量的示例应用 lots of sample applications 使用 `-jc` 参数演示使用java配置Spring Security。

基础的网站安全java配置

第一步是创建我们的java配置。这个配置在你的应用程序中创建一个springSecurityFilterChain 的Servlet的过滤器 `springSecurityFilterChain` 负责所有安全（例如 保护应用程序的URL，验证提交的用户名和密码，重定向到登陆的表单等等）。你可以在下面找到大部分java配置项的例子:

```
  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 extends WebSecurityConfigurerAdapter {

 @Autowired
 public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
  auth
    .inMemoryAuthentication()
     .withUser("user").password("password").roles("USER");
 }
}
```

configureGlobal方法的名字不重要，然而，重要的是只在一个被 `@EnableWebSecurity`，`@EnableGlobalMethodSecurity` ，或者 `@EnableGlobalAuthentication` 注解的类中配置AuthenticationManagerBuilder，否则会有不可预知的后果。

真的是没有太多的配置，但它确实有很多功能，你可以在下面找到功能摘要:

- 在你的应用程序中对每个URL进行验证
- 为你生成一个登陆表单
- 允许使用用户名 **Username** *user* 和密码 **Password** *password* 使用验证表单进行验证。
- 允许用户登出
- CSRF attack CSPF攻击防范
- Session Fixation Session保护
- 安全 Header 集成
  - HTTP Strict Transport Security 对安全要求严格的HTTP传输安全
  - X-Content-Type-Options X-Content-Type-Options集成
  - 缓存控制（稍后可以允许你缓存静态资源）
  - X-XSS-Protection X-XSS-Protection集成
  - X-Frame-Options 集成防止点击劫持 Clickjacking
- 和以下 Servlet API 方法集成
  - HttpServletRequest#getRemoteUser()
  - HttpServletRequest.html#getUserPrincipal()
  - HttpServletRequest.html#isUserInRole(java.lang.String)
  - HttpServletRequest.html#login(java.lang.String, java.lang.String)
  - HttpServletRequest.html#logout()

AbstractSecurityWebApplicationInitializer

下一步是在war里注册 `springSecurityFilterChain` 。这可以通过Spring在Servlet 3.0+环境中对 Spring's WebApplicationInitializer supportWebApplicationInitializer的支持进行了java配置，Spring Security提供了基本的抽象类 `AbstractSecurityWebApplicationInitializer` 这可以确保 `springSecurityFilterChain` 已经被注册。我们使用 `AbstractSecurityWebApplicationInitializer` 的不同方式取决于你是已经在使用Spring框架还是只是用了Spring Security。

- [abstractsecuritywebapplicationinitializer-without-existing-spring] - 如果你没有使用框架就使用这个说明
- [abstractsecuritywebapplicationinitializer-with-spring-mvc] - 如果你正在使用Spring框架使用这个说明

AbstractSecurityWebApplicationInitializer 不与Spring一起使用

如果你没有使用Spring MVC 或Spring，你需要传递 `WebSecurityConfig` 到超类来确保配置被使用，你可以参考下面的例子:

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

```
public class SecurityWebApplicationInitializer
 extends AbstractSecurityWebApplicationInitializer {

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

`SecurityWebApplicationInitializer` 将会做下面的事情:

- 自动为你的应用程序的每个URL注册 `springSecurityFilterChain` 过滤器
- 添加一个 `ContextLoadListener` 用来载入 WebSecurityConfig.

AbstractSecurityWebApplicationInitializer 与 Spring MVC一起使用

如果我们在应用程序的其他地方已经使用了Spring，那么我们已经有了一个 `WebApplicationInitializer` 用来载入Spring的配置。如果我们使用上面的配置将会得到一个错误。所以我们应该使用已经存在的 `ApplicationContext` 注册Spring Security。举个例子，如果我们使用Spring MVC我们的 `SecurityWebApplicationInitializer` 应该看起来和下面差不多:

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

public class SecurityWebApplicationInitializer
 extends AbstractSecurityWebApplicationInitializer {

}
```

这回简单的只为你的应用程序的所有URL注册springSecurityChain过滤器。然后我们需要确保这个Security配置被载入到我们已经存在的 `WebSecurityConfig` 。例如，如果你使用Spring MVC 它应该被加入到 `getRootConfigClasses()`

```
 public class MvcWebApplicationInitializer extends
  AbstractAnnotationConfigDispatcherServletInitializer {

 @Override
 protected Class<?>[] getRootConfigClasses() {
  return new Class[] { WebSecurityConfig.class };
 }

 // ... other overrides ...
}
```

HttpSecurity

到目前为止我们的WebSecurityConfig只包含了关于如何验证我们的用户的信息。Spring Security怎么知道我们相对所有的用户进行验证？Spring Securityn怎么知道我们需要支持基于表单的验证？原因
是 `WebSecurityConfigurerAdapter` 在 `configure(HttpSecurity http)` 方法中提供了一个默认的配置，看起来和下面类似:

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

上面的默认配置:

- 确保我们应用中的所有请求都需要用户被认证
- 允许用户进行基于表单的认证
- 允许用户使用HTTP基于验证进行认证

你可以看到这个配置和下面的XML命名配置相似:

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

java配置使用 `and()` 方法相当于XML标签的关闭。 这样允许我们继续配置父类节点。如果你阅读代码很合理，想配置请求验证，并使用表单和HTTP基本身份验证进行登录。

然而，java配置有不同的默认URL和参数，当你自定义用户登录页是需要牢记这一点。让我们的URL冯家RESTful，另外不要那么明显的观察出我么你在使用Spring Security这样帮助我们避免信息泄露。 information leaks。比如：

Java配置和表单登录

你可能会想知道系统提示您登录表单从哪里来的，因为我们都没有提供任何的HTML或JSP文件。由于Spring Security的默认配置并没有明确设定一个登录页面的URL，Spring Security自动生成一个，基于这个功能被启用，使用默认URL处理登录的提交内容，登录后跳转的URL等等。

自动生成的登录页面可以方便应用的快速启动和运行，大多数应用程序都需要提供自己的登录页面。要做到这一点，我们可以更新我们的配置，如下所示：

```
 protected void configure(HttpSecurity http) throws Exception {
http
 .authorizeRequests()
  .anyRequest().authenticated()
  .and()
 .formLogin()
  .loginPage("/login") (1)
  .permitAll();        (2)
}
```

(1) 指定登录页的路径

(2) 我们必须允许所有用户访问我们的登录页（例如为验证的用户），这个 `formLogin().permitAll()` 方法允许基于表单登录的所有的URL的所有用户的访问。

一个我么当前配置使用的JSP实现的页面如下：

下面这个登陆页是我们的当前配置，如果不符合我们的要求我们可以很容易的更新我们的配置。

```
 <c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">       (1)
 <c:if test="${param.error != null}">           (2)
  <p>
   Invalid username and password.
  </p>
 </c:if>
 <c:if test="${param.logout != null}">          (3)
  <p>
   You have been logged out.
  </p>
 </c:if>
 <p>
  <label for="username">Username</label>
  <input type="text" id="username" name="username"/> (4)
 </p>
 <p>
  <label for="password">Password</label>
  <input type="password" id="password" name="password"/> (5)
 </p>
 <input type="hidden"                           (6)
  name="${_csrf.parameterName}"
  value="${_csrf.token}"/>
 <button type="submit" class="btn">Log in</button>
</form>
```

(1) 一个POST请求到 `/login` 用来验证用户

(2) 如果参数有 `error`，验证尝试失败

(3) 如果请求蚕食 `logout` 存在则登出

(4) 登录名参数必须被命名为*username*

(5) 密码参数必须被命名为*password*

(6) CSRF参数，了解更多查阅后续 包括CSRF令牌 和Cross Site Request Forgery (CSRF) 相关章节

验证请求

我们的例子中要求用户进行身份验证并且在我们应用程序的每个URL这样做。我么你可以通过给 `http.authorizeRequests()` 添加多个子节点来指定多个定制需求到我们的URL。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)
  .anyRequest().authenticated()                                                (5)
  .and()
 // ...
 .formLogin();
```

```
}
```

1. `http.authorizeRequests()` 方法有多个子节点，每个macher按照他们的声明顺序执行。
2. 我们指定任何用户都可以通过访问的多个URL模式。任何用户都可以访问URL以"/resources/", equals "/signup", 或者 "/about"开头的URL。
3. 以 "/admin/" 开头的URL只能由拥有 "ROLE_ADMIN"角色的用户访问。请注意我们使用 `hasRole` 方法，没有使用 "ROLE_" 前缀.
4. 任何以"/db/" 开头的URL需要用户同时具有 "ROLE_ADMIN" 和 "ROLE_DBA"。和上面一样我们的 `hasRole` 方法也没有使用 "ROLE_" 前缀.
5. 尚未匹配的任何URL要求用户进行身份验证

处理登出

当使用 `WebSecurityConfigurerAdapter`，注销功能会自动应用。默认是访问URL`/logout`将注销登陆的用户：

- 使HTTP Session 无效
- 清楚所有已经配置的 RememberMe 认证
- 清除 `SecurityContextHolder`
- 跳转到 `/login?logout`

和登录功能类似，你也有不同的选项来定制你的注销功能：

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

1. 提供注销支持，使用 `WebSecurityConfigurerAdapter` 会自动被应用。
2. 设置触发注销操作的URL (默认是 `/logout`). 如果CSRF内启用（默认是启用的）的话这个请求的方式被限定为POST。 请查阅相关信息 JavaDoc相关信息.
3. 注销之后跳转的URL。默认是 `/login?logout` 。具体查看 the JavaDoc文档.
4. 让你设置定制的 `LogoutSuccessHandler` 。如果指定了这个选项那么 `logoutSuccessUrl()` 的设置会被忽略。请查阅 JavaDoc文档.
5. 指定是否在注销时让 `HttpSession` 无效。 默认设置为 **true**。 在内部配置 `SecurityContextLogoutHandler` 选项。 请查阅 JavaDoc文档.
6. 添加一个 `LogoutHandler` .默认 `SecurityContextLogoutHandler` 会被添加为最后一个 `LogoutHandler` 。
7. 允许指定在注销成功时将移除的cookie。这是一个现实的添加一个 `CookieClearingLogoutHandler` 的快捷方式。

> 注销也可以通过XML命名空间进行配置，请参阅Spring Security XML命名空间相关文档获取更多细节logout element。

一般来说，为了定制注销功能，你可以添加 `LogoutHandler` 和 `LogoutSuccessHandler` 的实现。 对于许多常见场景，当使用流食式API时，这些处理器会在幕后进行添加。

LogoutHandler

一般来说， `LogoutHandler` 的实现类可以参阅到注销处理中。他们被用来执行必要的清理，因而他们不应该抛出错误，我们提供您各种实现：

- PersistentTokenBasedRememberMeServices
- TokenBasedRememberMeServices
- CookieClearingLogoutHandler
- CsrfLogoutHandler
- SecurityContextLogoutHandler

请查看 Remember-Me的接口和实现 获取详情。

流式API提供了调用相应的 `LogoutHandler` 实现的快捷方式, 而不是直接提供 `LogoutHandler` 的实现。例如：`deleteCookies()` 允许指定注销成功时要删除一个或者多个cookie。这是一个添加 `CookieClearingLogoutHandler` 的快捷方式。

LogoutSuccessHandler

`LogoutSuccessHandler` 被 `LogoutFilter` 在成功注销后调用，用来进行重定向或者转发相应的目的地。注意这个借口与 `LogoutHandler` 几乎一样，但是可以抛出异常。

下面是提供的一些实现：

- SimpleUrlLogoutSuccessHandler
- HttpStatusReturningLogoutSuccessHandler

和前面提到的一样，你不需要直接指定 `SimpleUrlLogoutSuccessHandler`。而使用流式API通过设置 `logoutSuccessUrl()` 快捷的进行设置 `SimpleUrlLogoutSuccessHandler`。注销成功 后将重定向到设置的URL地址。默认的地址是 `/login?logout`.

在REST API场景中 `HttpStatusReturningLogoutSuccessHandler` 会进行一些有趣的改变。 `LogoutSuccessHandler` 允许你设置一个返回给客户端的HTTP状态码（默认返回200）来替换重定向到URL这个动作。

进一步的注销相关的参考（TODO）

- 处理注销
- 测试注销
- HttpServletRequest.logout()
- 章节"Remember-Me接口和实现"
- 注销的CSRF说明
- 点点注销（CAS协议）
- 注销的XML命名空间章节

验证

到现在为止我们只看了一下基本的验证配置，让我们看看一些稍微高级点的身份验证配置选项。

内存中的身份验证

我们已经看到了一个单用户配置到内存验证的示例，下面是配置多个用户的例子：

```
  @Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
 auth
  .inMemoryAuthentication()
   .withUser("user").password("password").roles("USER").and()
   .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

JDBC 验证

你可以找一些更新来支持JDBC的验证。下面的例子假设你已经在应用程序中定义好了 `DataSource`， jdbc-javaconfig 示例提供了一个完整的基于JDBC的验证。

```
  @Autowired
private DataSource dataSource;

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

LDAP 验证

你可以找一些更新来支持LDAP的身份验证， ldap-javaconfig 提供了一个完成的使用基于LDAP的身份验证的示例。

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

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

上面的例子中使用一下LDIF和前如何Apache DS LDAP示例。

users.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
```

AuthenticationProvider

您可以通过一个自定义的`AuthenticationProvider`为bean定义自定义身份验证。 例如， 下面这个例子假设自定义身份验证`SpringAuthenticationProvider`实现了`AuthenticationProvider`：

NOTE:`AuthenticationManagerBuilder`如果还不密集这将仅被使用。

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

UserDetailsService

你可以通过一个自定义的`UserDetailsService`为bean定义自定义身份验证。 例如，下面这个例子假设自定义身份验证`SpringDataUserDetailsService`实现了`UserDetailsService`：

NOTE:`AuthenticationManagerBuilder`如果还不密集这将被仅被使用并且没有`AuthenticationProviderBean`申明。

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

你也可以通过让`passwordencoder`为bean自定义密码如何编码。 例如，如果你使用BCrypt，你可以添加一个bean定义如下图所示：

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

LDAP 验证

多个HttpSecurity

我们可以配置多个HttpSecurity实例，就像我们可以有多个 `<http>` 块. 关键在于对 `WebSecurityConfigurationAdapter` 进行多次扩展。例如下面是一个对 `/api/` 开头的URL进行的不同的设置。

```
 @EnableWebSecurity
public class MultiHttpSecurityConfig {
 @Autowired
 public void configureGlobal(AuthenticationManagerBuilder auth) { ❶
  auth
   .inMemoryAuthentication()
    .withUser("user").password("password").roles("USER").and()
    .withUser("admin").password("password").roles("USER", "ADMIN");
 }

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

 @Configuration                                                   ❹
 public static class FormLoginWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

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

❶ 配置正常的验证。

❷ 创建一个 `WebSecurityConfigurerAdapter` ，包含一个 `@Order` 注解，用来指定个哪一个 `WebSecurityConfigurerAdapter` 更优先。

❸ `http.antMatcher` 指出，这个 `HttpSecurity` 只应用到以 `/api/` 开头的URL上。

❹ 创建另外一个 `WebSecurityConfigurerAdapter` 实例。用于不以 `/api/` 开头的URL，这个配置的顺序在 `ApiWebSecurityConfigurationAdapter` 之后，因为他没有指定 `@Order` 值为 `1` (没有指定 `@Order` 默认会被放到最后).

方法安全

从2.0 开始Spring Security对服务层的方法的安全有了实质性的改善。他提供对JSR-250的注解安全支持像框架原生 `@Secured` 注解一样好。从3.0开始你也可以使用新的基于表达式的注解 expression-based annotations。你可以应用安全到单独的bean，使用拦截方法元素去修饰Bean声明，或者你可以在整个服务层使用 AspectJ风格的切入点保护多个bean。

EnableGlobalMethodSecurity

我们可以在任何使用 `@Configuration` 的实例上，使用 `@EnableGlobalMethodSecurity` 注解来启用基于注解的安全性。例如下面会启用Spring的 `@Secured` 注解。

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

添加一个注解到一个方法（或者一个类或者接机）会限制对相应方法的访问。Spring Security的原生注解支持定义了一套用于该方法的属性。这些将被传递 `AccessDecisionManager` 到来做实际的决定：

```
 public interface 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);
}
```

使用如下代码启用JSR-250注解的支持

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

这些都是基于标准的，并允许应有个你简单的基于角色的约束，但是没有Spring Security的本地注解的能力。要使用基于表达书的语法，你可以使用：

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

和响应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);
}
```

GlobalMethodSecurityConfiguration

有时候你可能需要执行一些比`@EnableGlobalMethodSecurity`注解允许的更复杂的操作。对于这些情况，你可以扩展`GlobalMethodSecurityConfiguration`确保`@EnableGlobalMethodSecurity`注解出现在你的子类。例如如果你想提供一个定制的`MethodSecurityExpressionHandler`，你可以使用下面的配置：

```
 @EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
 @Override
 protected MethodSecurityExpressionHandler createExpressionHandler() {
  // ... create and return custom MethodSecurityExpressionHandler ...
  return expressionHandler;
 }
}
```

关于可以被重写的方法的更多信息，请参考`GlobalMethodSecurityConfiguration`的java文档。

已配置对象的后续处理

Spring Security的Java配置没有公开每个配置对象的每一个属性，这简化了广大用户的配置。毕竟如果要配置每一个属性，用户可以使用标准的Bean配置。

虽然有一些很好的理由不直接暴露所有属性，用户可能仍然需要更多高级配置，为了解决这个Spring Security引入了`ObjectPostProcessor`概念，用来替换java配置的对象实例。例如：如果你想在`filterSecurityPublishAuthorizationSuccess`里配置`FilterSecurityInterceptor`属性，你可以想下面一样：

```
 @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.setPublishAuthorizationSuccess(true);
     return fsi;
    }
   });
}
```

安全命名空间配置

简介

命名空间配置在Spring框架的2.0版本就可以使用了，他允许你通过额外的XML架构元素补充传统的Spring bean应用程序上下文。你可以从Spring的参考文档找到更多信息 Reference Documentation.。命名空间元素可以简单的允许配置单个bean，或者更强大的，定义一个可选的配置语法，这样更贴近问题域并且对用户隐藏背后的复杂性。一个简单的元素可以隐藏多个bean和添加到应用程序上下文的多个处理步骤。例如：从安全命名空间添加后面的元素到应用程序上下文将开始一个LDAP服务到应用程序内用于测试：

```
<security:ldap-server />
```

这比配置一个Apache木服务器bean简单的多。最常见的替代配置需求是`ldap-server`元素的属性支持，用户不用担心他们要创建的bean属名称:[你可以从:specialcharacters,macros[LDAP 身份验证]找到如何使用`ldap-server`元素的更多的信息.]。当编译应用的Context文件时良好的XML编译器应该可以提供您可用的属性和元素的信息。我们建议你尝试使用 Spring Tool Suite，它具有与标准Spring命名空间工作的特别的功能。

为了在你的应用程序上下文中使用安全命名空间，你需要将`spring-security-config`包含到你的classpath中。然后在你的上下文文件中加入以下的结构声明：

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:security="http://www.springframework.org/schema/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security.xsd">
 ...
</beans>
```

在很多示例（包括示例应用程序）中你将会看到，我们经常使用security作为默认的命名空间而不是使用beans，这样我们可以在所有安全命名空间中忽略前缀，使得内容更加容易阅读。如果你的应用程序上下文被分割成单独的文件，大部分的安全配置被放到一个文件中，你可能也想这样做。你的安全应用上下文文件应该像下面的一样：

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security.xsd">
 ...
</beans:beans>
```

我们假设在这一章节我们都使用这种语法。

命名空间的设计

命名空间是设计用来捕捉框架最常见用途和提供一个简化和简介的语法用来在应用程序中打开他们。设计师基于框架中的大规模依赖，并且可以划分为以下几个方面：

- *Web/HTTP 安全* - 最复杂的部分，设置过滤器和应用框架验证机制的相关服务bean，渲染登录和错误页面等等。
- *业务对象（方法）安全* - 业务层安全选项。
- *AuthenticationManager* - 处理来自框架其他部分的认证请求。
- *AccessDecisionManager* - 为网页和方法安全提供访问决策，会注册一个默认的但是你可以使用一个定制的来取代他，使用一般的Spring bean语法即可定义。
- *AuthenticationProvider*- 认证管理器认证用户的机制，命名空间提供多种标准选项的支持，同时使用传统语法添加自定义bean的方法。
- *UserDetailsService* - 和AuthenticationProviders密切相关，但往往也被其他bean需要。

我们将在后续章节查看怎么配置他们。

开始使用安全命名空间配置

在这一章，我们将看看怎么创建一个命名空间配置来使用框架的主要功能。让我们假设你想要快速的使用命名空间配置添加验证支持和访问控制和一些测试的登录到一个已经存在的网站应用程序。然后我们看看如何验证数据库和其他的安全仓库。在后续章节我们将介绍更多高级的命名空间配置选项。

web.xml 配置

你需要做的第一件事情是添加下面的过滤器定义到你的`web.xml`文件：

```
<filter>
```

```
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

这提供了一个钩子到Spring Security的网页基础设施。`DelegatingFilterProxy`是一个委托了在应用的Context中定义为bean的过滤器实现的一个Spring框架类，在这种情况下，bean的名字是`springSecurityFilterChain`，这是网络安全命名空间创建的一个基础类。注意你不能自己使用这个bean名字。一旦你添加这个到你的`web.xml`，你就可以开始编辑你的应用程序上下文文件，网络安全服务被配置到`<http>`元素。

最小<http>配置

开始开启网页安全你只需要：

```
  <http>
<intercept-url pattern="/**" access="hasRole('USER')" />
<form-login />
<logout />
</http>
```

这表示，我们想要对我们应用程序中的所有URL进行性检测。需要角色`ROLE_USER`访问他们，我们使用一个包含username和password的表单登录到应用程序，我们希望有一个注销的URL，让我们可以登出应用程序。`<http>`元素是所有网站相关的命名空间功能的父元类。`<intercept-url>`元素定义了一个规则使用ant语法去匹配传入的请求的URL。（在网站应用基础设施章节，查看 Request Matching and HttpFirewall来了解匹配是如何执行的） 你也可以环衬正则表达式语法（查看附录了解更多）。这个`access`属性定义了这个匹配的规则需要的访问需求。默认是一个典型的用逗号分隔的角色列表，有其中一个符合则允许执行这个请求。 前缀"ROLE_" 是一个标记，只是一个和用户的授权进行简单的比较。换句话说，一个简单的基于角色的检查会被应用。Spring Security的访问控制不仅限于简单角色(因此使用前缀来区分不同类型的)。

> 你可以使用多个`<intercept-url>`元素来为不同的URL定于不同的访问需求，他是他们将按照顺序计算，第一个匹配的会被使用，说以你必须将最特别的匹配放到最上面，你也可以添加一个`method`属性 用来限制HTTP方法(`GET`，`POST`，`PUT`等等).

为了添加一些用户，你可以直接在命名空间直接定义一组测试数据。

```
  <authentication-manager>
<authentication-provider>
 <user-service>
 <user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
 <user name="bob" password="bobspassword" authorities="ROLE_USER" />
 </user-service>
</authentication-provider>
</authentication-manager>
```

> 如果你熟悉框架的预命名空间版本，你很可能已经猜到这里怎么回事了。`<http>`元素负责创建`FilterChainProxy`和它使用的过滤器bean。像不正确过滤排序常见的问题是不再是一个问题，因为过滤器的位置都是预定义的。
>
> `<authentication-provider>`元素创建`DaoAuthenticationProvider` bean并且`<user-service>`元素创建了一个`InMemoryDaoImpl`。所有的验证供应商元素必须是`<authentication-manager>`元素的子元素。他创建一个`ProviderManager`并注册为验证供应商。你可以在 namespace appendix找到更多信息。如果你想了解框架的重要的一些类了解他们如何使用特别是后面你想定制一些事情，你值得去查看一下。

上面的配置定义了两个用户，他们的密码和在这个应用程序里的角色（这将用于访问控制）。也可以使用用户服务从标准的属性文件`properties`载入用户信息`user-service`。查看内存认证部分文件格式的更多细节in-memory authentication 。使用`<authentication-provider>`元素意思是这些用户信息将被认证管理器使用到请求验证。你可以设置多个`<authentication-provider>`元素来定义不同的验证源每一个都会被一次访问到。

此时你应该可以开始你的应用程序，你将被要求登录。尝试一下或者尝试使用附带的项目教程实例。

表单和基本登录选项

你可能会想当系统提示你登录这些登录表单哪里来的，因为我们都没有提供任何HTML和JSP文件。事实上我们并没有明确的设定一个登陆页的URL，Spring Security自动生成了一个，基于处理登录的URL标准值。以及登录后跳转的URL然后命名空间提供了大量的支持，让你可以自定义这些选项，例如如果你想自己设计登录页面，你可以使用：

```
  <http>
<intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
<intercept-url pattern="/**" access="ROLE_USER" />
```

```
<form-login login-page='/login.jsp'/>
</http>
```

另外请注意一个额外的`intercept-url`元素，来指定登录页的请求可以被任何匿名用户访问:[AuthenticatedVoter 类可以提供更多 匿名身份验证] AuthenticatedVoter `IS_AUTHENTICATED_ANONYMOUSLY`被处理的细节]. 否则请求会被/**规则匹配这将是不可能访问到登录页面。这是一个常见的配置错误，将导致应用无限循环。Spring Security会产生一条经过到日志中如果你的登录页被保护了。它也可以设置让所有请求完全匹配特定模式而绕过安全过滤器，通过定义像这样单独`http`元素的规则：

```
  <http pattern="/css/**" security="none"/>
<http pattern="/login.jsp*" security="none"/>

<http use-expressions="false">
<intercept-url pattern="/**" access="ROLE_USER" />
<form-login login-page='/login.jsp'/>
</http>
```

从Spring Security 3.1开始允许使用多个`http`元素来为不用的请求规则分别定义Security Filter Chain配置。如果`http`元素的`pattern`属性为设置则匹配所有请求。创建不进行安全检查的规则是这个语法的一个简答的例子，这个规则会被映射到一个空的过滤链。我们将会在后面的安全过滤链章节找到更多细节。[多个`<http>`元素的使用时一个非常重要的功能，例如它允许命名空间在一个应用程序中同时支持有状态和无状态的相同URL之前的语法，使用`filters="none"`属性到一个`intercept-url`元素与这一变化相抵触，在3.1将不再支持 。

重要的一点需要认识到，这些不安全的请求将完全无视任何Spring Security 的web相关的配置或附加属性，例如`requires-channel`通道的限定，所以在请求期间你将无法访问当前用户信息，或调用需要进行安全验证的方法。如果你想让安全过滤链被应用另外也可以使用 `access='IS_AUTHENTICATED_ANONYMOUSLY'`。

如果你想使用基本验证来替换掉表单的登录，你可以将配置修改如下:

```
  <http use-expressions="false">
<intercept-url pattern="/**" access="ROLE_USER" />
<http-basic />
</http>
```

当用户访问被保护的资源时，基本身份验证将优先弹出提示框来进行登录。在这总配置如果想使用基于表单的登录，他仍然有效。例如，将登录嵌入另外一个网页。

设置默认的登录后地址

如果不是由试图访问受保护的资源促成的登录，默认目标URL就会起作用。这是一个当用户成功登录后会被带去的URL默认是"/"，你也可以通过配置`default-target-target`为`true`让用户一直跳转到这个地址而无论用户是按需还是明确的选择登录。这在你的应用程序需要用户一直从主页开始时很有用。例如：

```
  <http pattern="/login.htm*" security="none"/>
<http use-expressions="false">
<intercept-url pattern='/**' access='ROLE_USER' />
<form-login login-page='/login.htm' default-target-url='/home.htm'
  always-use-default-target='true' />
</http>
```

为了对跳转目标进行更多控制，你可以使用`authentication-success-handler-ref`来替代`default-target-url`。这个bean的引用应该是`AuthenticationSuccessHandler`的一个实例。你可以在 Core Filters 章节和命名空间附录找到如何定制认证失败的流程的更多信息。

处理登出

`logout`元素，增加了一个登出导航到一个特定URL的支持，默认的登出URL是`/logout`。但是你可以使用`logout-url`属性指定为其他的URL。更多的属性你可以从附录找到。

使用其他的验证供应商

在实践中，你将需要将一些名字添加到应用程序上下文文件中更具扩展性的数据源。你有可能想把用户信息存放在注入数据库或者LDAP服务器中。LDAP命名空间的配置在LDAP chapter章节，这里就不说了。,如果你的应用程序上下文中有一个Spring Security的`UserDetailsService`的实现叫做"myUserDetailsService"，你可以这样使用：

```
  <authentication-manager>
 <authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>
```

如果你想用一个数据库，你可以这样用：

```
  <authentication-manager>
<authentication-provider>
 <jdbc-user-service data-source-ref="securityDataSource"/>
</authentication-provider>
</authentication-manager>
```

在应用程序上下文中，"securityDataSource"是`DataSource` bean的名字，指明一个数据库包含标准的 Spring Security user data tables. 另外， 你可以配置一个Spring Security `JdbcDaoImpl` bean 并且指明它使用`user-service-ref` 属性:

```
  <authentication-manager>
<authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>

<beans:bean id="myUserDetailsService"
 class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
<beans:property name="dataSource" ref="dataSource"/>
</beans:bean>
```

你也可以像下面一样使用标准的`AuthenticationProvider`

```
  <authentication-manager>
 <authentication-provider ref='myAuthenticationProvider'/>
</authentication-manager>
```

`myAuthenticationProvider`是应用程序上下文的一个bean名字 。他实现了`AuthenticationProvider`接口。 你可以设置多个验证供应商，这样这些供应商就会按照定义的顺序被查询。看看 验证管理器和命名空间 章节“验证管理器和命名空间”连接`AuthenticationManager`在命名空间中怎么被配置。

添加密码编码器

密码应该一直使用一个尽可能安全的哈希算法进行编码（非标准的算法，例如SHA或者MD5）。这在`<password-encoder>`中进行支持，使用`bcrypt`编码密码，原始的供应商配置应该类似这样：

```
  <beans:bean name="bcryptEncoder"
 class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>

<authentication-manager>
<authentication-provider>
 <password-encoder ref="bcryptEncoder"/>
 <user-service>
 <user name="jimi" password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f"
   authorities="ROLE_USER, ROLE_ADMIN" />
 <user name="bob" password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"
   authorities="ROLE_USER" />
 </user-service>
</authentication-provider>
</authentication-manager>
```

Bcrypt 在大部分时候是一个好的选择。除非你有一个旧系统强迫你使用一个不用的算法。如果你正在使用一个简单的哈希算法或者更糟糕存储了明文密码，你应该考虑迁移到`bcrypt`这个更安全的选项。

高级web功能

Remember-Me 验证

查看独立的Remember-Me chapter章节来查看该功能的配置。

添加 HTTP/HTTPS 通道安全

如果你的应用程序同时支持HTTP和HTTPS，你要求特定的URL只能使用HTTPS，这是可以直接使用`<intercept-url>`的`requires-channel`属性:

```
  <http>
<intercept-url pattern="/secure/**" access="ROLE_USER" requires-channel="https"/>
<intercept-url pattern="/**" access="ROLE_USER" requires-channel="any"/>
...
</http>
```

有了这个配置,如果用户试图访问任何匹配的"/secure/**"模式使用HTTP,他们将首先被重定向到一个HTTPS URL脚注:[想要了解channel-processing 如何实现，请查阅java文档`ChannelProcessingFilter`以及相关的类]。可用的选项是“http”、“https”或“任何”。使用价值“任何”意味着可以使用HTTP或HTTPS。

该选项可选值为"http","https"或者"any"，使用"any"意思是使用HTTP或者HTTPS均可。如果应用程序使用HTTP和HTTPS非标准的端口，你可以指定如下端口映射的列表：

```
  <http>
...
<port-mappings>
 <port-mapping http="9080" https="9443"/>
</port-mappings>
</http>
```

请注意,为了真正的安全,应用程序不应该使用HTTP或HTTP和HTTPS之间切换。它应该开始在HTTPS(与用户进入一个HTTPS URL)使用安全连接,以避免任何中间人攻击的可能性。

Session 管理

检测超时

你可以配置Spring Security检测无效的Session ID提交并且将用户重定向到一个适当的URL，这通过`session-management`元素来达到：

```
  <http>
...
<session-management invalid-session-url="/invalidSession.htm" />
</http>
```

请注意，如果你使用这个机制来检测会话超时，如果用户注销，然后重新登录，但不关闭浏览器它可能会错误地抛出一个错误。这是因为当你让Session失效时，Cookie没有被清理干净，就算用户已注销还是会重新提交session的cookie而不会清除。你可以在注销时明确的删除JSESSIONID的cookie，例如：通过在注销处理程序使用一下语法：

```
  <http>
<logout delete-cookies="JSESSIONID" />
</http>
```

不幸的是这个不保证在所有servlet容器都会正常工作，所以你需要在你的环境下进行测试。

> 如果你的应用程序运行在代理后面，你还可以配置代理服务器去删除session的cookie。例如，使用Apache httpd的mod_headers模块。下面的指令可以通过在注销请求的响应头删除`JSESSIONID`的cookie。(假设应用程序部署在`/tutorial`路径下):
>
> ```
>  <LocationMatch "/tutorial/logout">
> Header always set Set-Cookie "JSESSIONID=;Path=/tutorial;Expires=Thu, 01 Jan 1970 00:00:00 GMT"
> </LocationMatch>
> ```

并发Session控制

如果你想限制单个用户访问你的应用程序的能力。Spring Security通过后面简单的配置马上启用。首先你需要添加后面的监听器到你的`web.xml`文件。让Spring Security获得session的生存事件：

```
  <listener>
<listener-class>
 org.springframework.security.web.session.HttpSessionEventPublisher
</listener-class>
</listener>
```

然后添加后面的行到你的应用程序上下文：

```
  <http>
...
<session-management>
 <concurrency-control max-sessions="1" />
</session-management>
</http>
```

这将放置用户登录多次。第二次登录将导致第一次登录变成无效。通常我们更想放置第二次登录，在这种情况下，你可以使用：

```
  <http>
...
<session-management>
 <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
</session-management>
</http>
```

第二次登录将被拒绝，如果基于表单的验证被启用这个用户将会被发送到`authentication-failure-url`。如果第二次登录是通过其他非交互的机制，比如“记住我”功能，进行登录的。那么一个“unauthorized”(401)错误将会被发送给客户端。如果你想替换成一个错误页面，你可以为`session-management`添加一个`session-authentication-error-url`属性。

如果你正在使用基于表单的登录定制验证过滤器，那么你必须明确地配置同步会话控制的支持。更多的细节可以在 Session Management chapter章节中找到。

Session Fixation 攻击保护

Session fixation完成攻击是一个潜在的威胁。攻击者访问网站生成一个Session，然后诱使其他用户用同一个会话登录（例如：通过发送包含会话标识符作为一个参数链接）。Spring Security通过在登录时创建新的Session或者修改Session ID来应对这种情况。如果你不需要这个保护或者与一些其他需求冲入你可以通过`<session-management>`的`session-fixation-protection`属性来控制这个行为。它有4个选项：

- `none` - 什么都不做，原来的会话将会保留
- `newSession` - 创建一个新的干净的Session，不会复制已经存在的Session属性到新的Session，这是Servlet3.0及之前的

容器的默认设置

- `migrateSession` - 创建一个新的Session并且拷贝所有已经存在的Session属性到新的Session，这是Servlet3.0及之前的容器的默认设置。
- `changeSessionId` - 不创建新的Session，使用Servlet容器提供的Session完成攻击保护（`HttpServletRequest#changeSessionId()`）。这个选项只有在Servlet3.1（java EE 7）和更新的容器下可用。在旧的容器设置这个选项会产生一个异常。在Servlet3.0和更新的容器就默认该选项。

当会话完成保护发生时，它会产生`SessionFixationProtectionEvent`发布到应用程序上下文，如果使用`changeSessionId`，这种保护也将导致 任何`javax.servlet.http.HttpSessionIdListener`被通知,所以如果你的代码监听这两个事件要特别小心。查看Session Management 管理章节查看更多信息。

OpenID 支持

通过一个简单的改变，命名空间可支持用 OpenID 替换或者添加到普通的基于表单登录：

```
  <http>
<intercept-url pattern="/**" access="ROLE_USER" />
<openid-login />
</http>
```

然后，你应该将自己注册为一个OpenID供应商（如myopenid.com），并添加用户信息到你的基于内存的`<user-service>`：

```
  <user name="http://jimi.hendrix.myopenid.com/" authorities="ROLE_USER" />
```

你应该能够使用`myopenid.com`网站登录来进行验证。另外，也可以通过设置 `openid-login`元素的`user-service-ref`属性来指定一个 `UserDetailsService` bean来使用OpenID。查看前面的authentication providers章节了解更多信息请注意，我们省略从上述用户配置中的密码属性，由于该组的用户数据只被用于加载当前的用户。内部会产生一个随机密码，放置你意外的将这个用户数据用作验证源到你的配置中的其他地方。

属性交换

OpenID 的属性交换支持 attribute exchange。下面的例子尝试接受从OpenID供应商接收邮件和全名，用于应用程序中：

```
  <openid-login>
<attribute-exchange>
 <openid-attribute name="email" type="http://axschema.org/contact/email" required="true"/>
 <openid-attribute name="name" type="http://axschema.org/namePerson"/>
</attribute-exchange>
</openid-login>
```

每个OpenID属性的“type”是一个URI，有一种特定的模式来确定，这个例子中是 http://axschema.org/。如果属性必须在成功认证后接收，可以设置 `required` 属性。确切的模式和属性的支持将取决于你的OpenID提供商。该属性值返回作为认证过程的一部分，随后可以使用下面的代码访问：

```
  OpenIDAuthenticationToken token =
 (OpenIDAuthenticationToken)SecurityContextHolder.getContext().getAuthentication();
List<OpenIDAttribute> attributes = token.getAttributes();
```

`OpenIDAttribute`包含了属性类型和接收到的值（在有多个属性值的情况下包含多个值），通过查看Spring Security核心组件的technical overview章节我们可以了解更多的`SecurityContextHolder`类的使用方法。如果希望使用多个身份提供者方，多重属性交换配置也被支持。你可以提供多个`attribute-exchange`元素。在每个上面使用`identifier-matcher`属性。它包含一个正则表达式，会匹配由用户提供的OpenID标识符。查看代码库的OpenID示例应用的一个示例配置。对 Google, Yahoo 和 MyOpenID 提供了不同的属性列表。

相应头

查看如何定制头元素的更多信息请查看安全HTTP响应头章节。

添加自己的过滤器

如果你以前使用过Spring Security，你就会知道，这个 框架维护一个过滤器链，以便应用它的服务。你可能想要添加自己的过滤器到过滤器堆栈的特定位置，或者使用一个Spring Security还没有一个命名空间配置的选项的过滤器（比如CAS）。或者你想使用一个标准命名空间过滤器的定制化版本，比如`UsernamePasswordAuthenticationFilter`是由`<form-login>`元素显式的使用Bean来获取一些额外的高级配置选项，在过滤器链不直接暴露的情况下，你怎么使用命名空间配置这一点？

使用命名空间时过滤器的顺序始终严格执行，当创建应用程序上下文，过滤器Bean被命名空间处理代码进行排序，标准的Spring Security过滤器都具有的命名空间和一个众所周知的位置的别名。

在以前的版本中，排序发生在过滤器实例创建之后，在应用程序上下文后处理中。语法在3.0有一些轻微的改变，这会影响到解析`<http>`元素时，你自己的过滤器如何被添加到整个过滤器列表中

标准过滤器别名和顺序，别名和创建的命名空间元素/属性在下表列出，按照过滤器在链中出现的顺序列出：

Table 2. 标准过滤器别名和顺序

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	`ChannelProcessingFilter`	`http/intercept-url@requires-channel`
SECURITY_CONTEXT_FILTER	`SecurityContextPersistenceFilter`	`http`
CONCURRENT_SESSION_FILTER	`ConcurrentSessionFilter`	`session-management/concurrency-control`
HEADERS_FILTER	`HeaderWriterFilter`	`http/headers`
CSRF_FILTER	`CsrfFilter`	`http/csrf`
LOGOUT_FILTER	`LogoutFilter`	`http/logout`
X509_FILTER	`X509AuthenticationFilter`	`http/x509`
PRE_AUTH_FILTER	`AbstractPreAuthenticatedProcessingFilter` Subclasses	N/A
CAS_FILTER	`CasAuthenticationFilter`	N/A
FORM_LOGIN_FILTER	`UsernamePasswordAuthenticationFilter`	`http/form-login`
BASIC_AUTH_FILTER	`BasicAuthenticationFilter`	`http/http-basic`
SERVLET_API_SUPPORT_FILTER	`SecurityContextHolderAwareRequestFilter`	`http/@servlet-api-provision`
JAAS_API_SUPPORT_FILTER	`JaasApiIntegrationFilter`	`http/@jaas-api-provision`
REMEMBER_ME_FILTER	`RememberMeAuthenticationFilter`	`http/remember-me`
ANONYMOUS_FILTER	`AnonymousAuthenticationFilter`	`http/anonymous`
SESSION_MANAGEMENT_FILTER	`SessionManagementFilter`	`session-management`
EXCEPTION_TRANSLATION_FILTER	`ExceptionTranslationFilter`	`http`
FILTER_SECURITY_INTERCEPTOR	`FilterSecurityInterceptor`	`http`
SWITCH_USER_FILTER	`SwitchUserFilter`	N/A

你可以添加自己的过滤器到列表中，使用`custom-filter`过滤元件和这些名字来指定你的过滤器应该出现在的位置之一：

```
  <http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</http>

<beans:bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

你也可以使用`after`和`before`属性来让你的过滤器插入到列表中的其他过滤器的前面和后面。`FIRST`和`LAST`可以用在`position`属性来设置你希望将你的过滤器插入到整个列表的前面或者后面。

Avoiding filter position conflicts

如果你插入的定制的过滤器可能会占用由命名空间创建的标准过滤器相同的位置，很重要的一点是不要错误的包含了命名空间的版本。移除所有你想替换的功能的过滤器元素。注意，你不能移除有`<http>`本身创建的过滤器，`SecurityContextPersistenceFilter`，`ExceptionTranslationFilter`和`FilterSecurityInterceptor`，一些其他的过滤器是默认被添加的，但是你可以禁止他们。除非你禁用session-fixation protection，一个`AnonymousAuthenticationFilter`会默认被添加，一个`SessionManagementFilter`也会被添加到过滤器链。

如果你替换的命名空间过滤器需要一个验证入口点（例如：在认证过程是通过尝试触发未认证用户访问以受保护资源）你也需要添加一个定制的入口点Bean。

创建自定义的 AuthenticationEntryPoint

如果你没有使用表单登录，OpenID或基本验证，你可能想像我们之前看到的一样，使用传统的bean语法，定义一个验证过滤器和入口点链接到命名空间。相应的 `AuthenticationEntryPoint` 可以使用 `<http>` 元素的 `entry-point-ref` 属性进行设置。

CAS示例应用是一个很好的示例，展示命名空间的定制bean的使用。如果你不熟悉认证的入口点，可以在technical overview章节看到相关讨论。

方法安全

从2.0版本开始Spring Security改进了对服务层方法的安全支持，它提供了对JSR-250注解安全支持以及框架的原生 `@Secured` 注解支持。从3.0开始你也可以使用新的expression-based annotations。你可以将安全应用到单个bean。使用 `intercept-methods` 元素装饰Bean的声明。或者你可以在使用AspectJ风格的切入点应用安全到整个服务层的多个Bean类。

The <global-method-security> 元素

这个元素用来在你的应用程序中启用基于安全性（通过设置元素的 `appropriate` 属性），同时分组将应用到整个应用程序上下文的安全切入点声明。你应该只定义一个 `<global-method-security>` 元素。下面的定义可以开启Spring Security的 `@Secured` 支持:

```
<global-method-security secured-annotations="enabled" />
```

添加一个注解到类或者接口的方法中可以限制对相应方法的访问。Spring Security的原生注解支持定义了一套用于该方法的属性。这些将被传递到 `AccessDecisionManager` 用来做实际的决定：

```
public interface 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);
}
```

启用 JSR-250 注解使用

```
<global-method-security jsr250-annotations="enabled" />
```

这些都是基于标准的，并允许应用简单的基于角色的约束，但是没有Spring Security的原生注解强大。要使用新的基于表达式的语法，你可以使用

```
<global-method-security pre-post-annotations="enabled" />
```

等价的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);
}
```

基于表达式的注解是一个很好的选择，如果你需要定义超过一个检查当前用户列表中的角色名称的简单的规则。

> 被注解的方法将仅在被定义为Spring 的Bean的实例时才能确保安全（在相同的应用程序的上下文中该方法-启用方法安全检查）。如果你想确保非Sprign创建的实例的安全性（比如使用 `new` 操作符创建的）那么你需要使用AspectJ.

> 你可以在同一个应用程序中启用多种注解，但是在一个借口或者类中只能使用一种类型的注解，否则会出现不明确的行为。如果对特定的方法使用了两个注解，只有其中的一个会被应用。

使用 protect-pointcut

使用 `protect-pointcut` 非常有用, 因为它允许你只用简单的声明就能对很多bean添加安全性。看看下面的例子：

```
<global-method-security>
<protect-pointcut expression="execution(* com.mycompany.*Service.*(..))"
```

```
 access="ROLE_USER"/>
</global-method-security>
```

这将保护在应用程序上下文定义的所有在 `com.mycompany` 包下类名以“service”结尾的类的方法。只有拥有 `ROLE_USER` 角色用户才能执行这些方法。和URL匹配一样，列表中多个匹配的话将会使用第一个匹配的安全注解，比切入点有更高的优点级。

默认的 AccessDecisionManager

这一章节假设你有一些Spring Security中访问控制的底层架构的知识。如果没有，你可以跳过它，后面再来看，这部分针对那些真正需要进行一些定制而不是简单的基于角色的安全用户。

当你使用命名空间配置时，一个 `AccessDecisionManager` 实例将会自动创建并注册用来按照你在 `intercept-url` 和 `protect-pointcut` （还有如果你使用了方法注解安全也包含在内）定义的访问属性进行访问决策。

默认的策略是使用一个 `AffirmativeBased` 、 `AccessDecisionManager` 和 `RoleVoter``AuthenticatedVoter` ，你也可以从authorization章节找到更多信息.

自定义 AccessDecisionManager

如果你需要使用一个更复杂的访问控制策略，那么很容易的为方法和Web安全设置替代方案。

对于方法安全，通过在 `access-decision-manager-ref` 上设置 `global-method-security` 属性来为应用程序指导适当的 `AccessDecisionManager` 的Bean `ID` 。

```
 <global-method-security access-decision-manager-ref="myAccessDecisionManagerBean">
...
</global-method-security>
```

网站安全的语法一样，但是是在 `http` 元素上。

```
 <http access-decision-manager-ref="myAccessDecisionManagerBean">
...
</http>
```

验证管理器和命名空间

Spring Security中主要的提高验证服务的借口是 `AuthenticationManager` ，这通常是一个Spring Security 的 `ProviderManager` 类的实例。如果你以前用过框架你可能已经熟悉了。如果不是后面的 technical overview chapter章节会讲到。这个bean是通过 `authentication-manager` 命名空间来注册。你不能使用自定义的 `AuthenticationManager` 如果通过命名空间使用HTTP或方法安全，但是这不应该是一个问题，因为你可以完全控制所使用的好的 `AuthenticationProvider` 。

你可能需要使用 `ProviderManager` 注册其他的 `AuthenticationProvider` Bean，你可以使用 `<authentication-provider>` 元素的 `ref` 属性，属性的值是你要添加的bean的名字，例如：

```
 <authentication-manager>
<authentication-provider ref="casAuthenticationProvider"/>
</authentication-manager>

<bean id="casAuthenticationProvider"
 class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
</bean>
```

另一个常见的需求是，上下文中的另一个bean可能需要引用 `AuthenticationManager` ，你可以注册一个 `AuthenticationManager` 的别名在应用程序上下文中的其他地方使用这个名字。

```
 <security:authentication-manager alias="authenticationManager">
...
</security:authentication-manager>

<bean id="customizedFormLoginFilter"
 class="com.somecompany.security.web.CustomFormLoginFilter">
<property name="authenticationManager" ref="authenticationManager"/>
...
</bean>
```

应用程序示例

这里有几个可用的网站应用程序示例。为了避免大量的下载。只有"totorial""contacts"示例包含到了分发的zip文件。其他的可用按照the introduction从源代码构建。你可以很容易的自己构建项，通过 http://spring.io/spring-security/网站可以获取更多信息。本章中提到的所有路径是相对于项目的源目录。

Tutorial 示例

本教程样本是一个让你入门的很好的基础的例子。它完全使用而来简单命名空间配置。编译好的应用程序包含在分发的zip包中。随时可以部署到你的Web容器(`spring-security-samples-tutorial-3.1.x.war`)。 form-based的认证机制结合remember-me 功能，验证供应商提供使用cookie自动基础登录。

我们建议你从本示例开始，因为XML非常小，易于遵循。最重要的是，你可以把这个XML文件（和它对应的 `web.xml` 入口）轻松地添加到现有的应用程序中。在这个基本集成成功的时候，我们建议你试着添加方法验证和领域对象安全。

Contacts 示例

该示例是一个高级的例子，它展示除了基本的应用程序安全领域对象的访问控制列表（ACL）的更强大的功能。本申请提供了一个接口让用户能够管理简单的联系人数据库（域对象）。

拷贝WAR问价你从Spring Security 分发包到你自己的容器的 `webapps` 目录来部署它。war名字是 `spring-security-samples-contacts-3.1.x.war` (扩展的版本号取决于你的版本号)。

开始你的容器后检查应用程序是否可用载入，访问 http://localhost:8080/contacts (或者其他适合你的容器URL)。

接下来，单击"Debug"。系统将提示你进行身份验证，以及页面提示的一系列用户名和密码。简单验证，显示一个结果页面。它应该包含类似于一下成功信息：

```
Security Debug Information

Authentication object is of type:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken

Authentication object as a String:

org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1f127853:
Principal: org.springframework.security.core.userdetails.User@b07ed00: Username: rod; \
Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER; \
Password: [PROTECTED]; Authenticated: true; \
Details: org.springframework.security.web.authentication.WebAuthenticationDetails@0: \
RemoteIpAddress: 127.0.0.1; SessionId: 8fkp8t83ohar; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER

Authentication object holds the following granted authorities:

ROLE_SUPERVISOR (getAuthority(): ROLE_SUPERVISOR)
ROLE_USER (getAuthority(): ROLE_USER)

Success! Your web filters appear to be properly configured!
```

一旦你成功地收到上述消息，返回到示例应用程序的主页，点击"Manage"。然后，你可以使用这个应用程序。注意只有当前用户的联系人会被显示，并且只有拥有 `ROLE_SUPERVISOR` 可用授权去删除联系人，在幕后 `MethodSecurityInterceptor` 保护业务对象。

这应用程序允许你修改与不同的联系人相关联的访问控制列表。一定要试试这个，并了解它是如何工作通过检查应用程序上下文XML文件。

LDAP 示例

LDAP例子程序提供了一个基本的配置和使用命名空间和使用传统的bean同等设置在同一个应用程序上下文文件。这意味着，其实是在这个应用程序中配置了两个相同的身份验证提供者。

OpenID 示例

OpenID的示例演示了如何使用命名空间来配置OpenID和如何设置为Google，Yahoo和OpenID设置 attribute exchange配置（如果你愿意的话，也可以添加其他的供应商）。它使用一个基于JQery的 openid-selector，以提供用户友好的登录页面，它允许用户很容易的选择一个验证提供者，而不是输入OpenID标识符。

应用程序的不同之处在于它允许任何用户访问该站点认证情景（只要他们的OpenID身份验证是成功的）。当你第一次登录时，你会得到一个"Welcome [your name]"的消息。如果你注销并重新登录（以相同的身份），那么这应更改为"欢迎回来"，这是通过使用一个实现自定义的 `UserDetailsService` 它分配一个标准角色给任何用户，并在内部存储身份的map。显然，一个真正的应用程序将使用一个数据库替换它。看一看源代码了解更多信息，此类还考虑到，不同供应商返回不同属性，并建立相应用户名。

CAS 示例

CAS示例要求你同时运行CAS服务器和CAS客户端。 它不包含在发行包，所以你应该按照 the introduction描述签出目标代码。你会发现 `sample/cas` 目录下的相关文件。还有在那里一个 `Readme.txt` 文件，这也解释了如何直接从源代码运行服务器和客户端，完全支持SSL。

JAAS 示例

该JAAS是很简单的如何用Spring Security 使用JAAS LoginModule 的例子。所提供的`LoginModule`将成功地验证用户，如果用户名和密码相等，否则抛出一个`LoginException`异常。在本示例中使用的`AuthorityGranter`总是授予角色ROLEUSER。示例应用程序还演示了如何通过设置`LoginModule`的jaas-api-provision等于`true`返回JAAS Subjec来运行。

Pre-Authentication 示例

此示例应用程序演示了如何绑定bean从pre-authentication 框架从Java EE容器使用登录信息。用户名和角色是由容器设置的。

代码是`samples/preauth`.

Spring Security 社区

问题跟踪

Spring Security 使用JIRA来管理bug报告和改进请求，如果你发现错误，请使用JIRA记录报告。不要在支持论坛，邮件列表，或通过电子邮件项目的记录。这些方法是临时的，我们更喜欢使用正式的流程。

如果可能的话，在你的问题的报告摸清提供一个JUnit测试，演示任何不正确的行为。或者，更好的是，提供了一个可以解决这个问题的补丁。同样，非常欢迎的提出改进需求，虽然我们只接受、有对应的单元测试的改进请求。这是必要的，以确保保持项目的测试覆盖率。

问题跟踪系统你可以通过这里访问： https://github.com/spring-projects/spring-security/issues.

成为参与者

我们欢迎你参与Spring Security项目。有贡献的方法很多，包括阅读论坛，并响应来自其他人的问题，编写新的代码，改进现有的代码，协助文档，开发样品或教程，或则干脆提出建议。

更多信息

欢迎为Spring Security 提出问题和意见。你可以使用Spring社区 论坛网站 http://spring.io/questions[http://spring.io/questions] 和其他Spring Security框架的用户进行讨论。请记住，正如前面说的使用JIRA提交bug报告。

架构与实现

一旦你熟悉了设置和运行一些基于命名空间配置的应用程序，你可能希望开发更多的框架去理解命名空间门面后面实际上是如何运转。类似大部分软件， Spring Security有一定的中央接口，以及通常在整个框架中使用的概念抽象类。在参考指南的这一部分，我们将看看其中的一些，看看它们如何协同工作去支持Spring Security中的身份验证和访问控制。

技术概述

运行环境

Spring Security 3.0需要Java 5.0的运行环境或者更高的版本. 由于 Spring Security 是以独立的方式运作, 就不需要什么特殊的配置文件到你的Java运行环境。特别是, 不需要配置专门的Java认证和授权服务(JAAS)策略文件或将Spring Security的位置放到普通路径中。

同样，如果你使用的是EJB容器或者Servlet容器也没有必要把任何特殊的配置文件放到任何地方，也不包括Spring Security的服务器类加载器。所有必须的文件都将包含在你的应用程序中。

这种设计给部署时间提供了最大的灵活性，你可以简单的复制你的目标文件(可以是JAR, WAR或者EAR)从一个系统到另一个系统，它会立即开始工作。

核心组件

在Spring Security 3.0的版本中，`spring-security-core`中的内容被精简到了最低限度。它不再包含web应用安全,LDAP或命名空间配置的任何代码。我们来看看一些Java类型，你会在核心模块中找到。它们代表了框架的基石，所有如果你需要越过一个简单的命名空间配置，那么最重要的是你要明白它们是什么，即使你不需要直接与它们进行交互。

SecurityContextHolder, SecurityContext和Authentication 对象

最根本的对象是`SecurityContextHolder`。我们把当前应用程序的当前安全环境的细节存储到它里边了， 它也包含了应用当前使用的主体细节。默认情况下`SecurityContextHolder`使用`ThreadLocal`存储这些信息， 这意味着，安全环境在同一个线程执行的方法一直是有效的， 即使这个安全环境没有作为一个方法参数传递到那些方法里。这种情况下使用`ThreadLocal`是非常安全的，只要记得在处理完当前主体的请求以后，把这个线程清除就行了。当然，Spring Security自动帮你管理这一切了， 你就不用担心什么了。

有些程序并不适合使用`ThreadLocal`，因为它们处理线程的特殊方法。比如Swing客户端也许希望Java Virtual Machine里所有的线程 都使用同一个安全环境。`SecurityContextHolder`可以配置启动策略来指定你希望上下文怎么被存储。对于一个独立的应用程序，你会使用`SecurityContextHolder.MODE_GLOBAL`策略。其他程序可能也想由安全线程产生的线程也承担同样的安全标识。这是通过使用`SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`实现。你可以通过两种方式更改默认的`SecurityContextHolder.MODE_THREADLOCAL`模式。第一个是设置系统属性，第二个是调用`SecurityContextHolder`的静态方法。大多数应用程序不需要修改默认值，但是如果你想要修改，可以看一下`SecurityContextHolder`的JavaDocs中的详细信息了解更多。

当前用户获取信息

我们在`SecurityContextHolder`内存储目前与应用程序交互的主要细节。Spring Security使用一个`Authentication`对象来表示这些信息。 你通常不需要创建一个自我认证的对象,但它是很常见的用户查询的`Authentication`对象。你可以使用以下代码块-从你的应用程序的任何部分-获得当前身份验证的用户的名称，例如:

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

if (principal instanceof UserDetails) {
String username = ((UserDetails)principal).getUsername();
} else {
String username = principal.toString();
}
```

通过调用`getContext()`返回的对象是`SecurityContext`接口的实例。这是保存在线程本地存储中的对象。我们将在下面看到，大多数的认证机制以Spring Security返回`UserDetails`实例为主。

The UserDetailsService

从上面的代码片段中还可以看出一件事，就是你可以从`Authentication`对象中获得安全主体。这个安全主体就是一个`Object`。大多数情况下，可以强制转换成`UserDetails`对象 。 `UserDetails`是一个Spring Security的核心接口。它代表一个主体，是扩展的，而且是为特定程序服务的。 想一下`UserDetails`章节，在你自己的用户数据库和如何把Spring Security需要的数据放到`SecurityContextHolder`里。为了让你自己的用户数据库起作用，我们常常把`UserDetails`转换成你系统提供的类，这样你就可以直接调用业务相关的方法了(比如 `getEmail()`, `getEmployeeNumber()`等等)。

现在，你可能想知道，我应该什么时候提供这个`UserDetails`对象呢?我怎么做呢?我想你说这个东西是声明式的，我不需要写任何代码，怎么办?简单的回答是，这里有一个特殊的接口叫`UserDetailsService`。这个接口里的唯一的一个方法，接收`String`类型的用户名参数，返回`UserDetails`:

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

这是Spring Security用户加载信息的最常用的方法并且每当需对用户的信息时你会看到它使用的整个框架。

成功认证后， `UserDetails`用于构建存储在`SecurityContextHolder`(详见 以下)的`Authentication`对象。好消息是，我们提供了一些`UserDetailsService`的实现，包括一个使用内存映射(`InMemoryDaoImpl`)而另一个使用JDBC(`JdbcDaoImpl`)。大多数用户倾向于写自己的，常常放到已有的数据访问对象(DAO)上使用这些实现，表示他们的雇员，客户或其他企业应用中的用户。记住这个优势，无论你用`UserDetailsService`返回的什么数据都可以通过`SecurityContextHolder`获得，就像上面的代码片段讲的一样。

> 关于`UserDetailsService`常常有一些混乱。它纯粹是用于用户数据的DAO并没有其它功能，除了提供该数据到其他组件的框架内。特别是，它*不会*对用户进行身份验证,这是由`AuthenticationManager`完成。在许多情况下，如果你需要自定义身份验证过程，直接实现 `AuthenticationProvider`更有意义 。

GrantedAuthority

除了主体，另一个`Authentication`提供的重要方法是`getAuthorities()`。这个方法提供了`GrantedAuthority`对象数组。毫无疑问， `GrantedAuthority`是赋予到主体的权限。这些权限通常使用角色表示，比如`ROLE_ADMINISTRATOR`或`ROLE_HR_SUPERVISOR`。这些角色会在后面，对web验证，方法验证和领域对象验证进行配置。Spring Security的其他部分用来拦截这些权限，期望他们被表现出现。 `GrantedAuthority`对象通常是使用`UserDetailsService`读取的。

通常情况下， `GrantedAuthority`对象是应用程序范围下的授权。它们不会特意分配给一个特定的领域对象。因此，你不能设置一个`GrantedAuthority` ，让他有权限展示编号54的`Employee`对象，因为如果有成千上万的这种授权，你会很快用光内存(或者，至少，导致程序花费大量时间去验证一个用户)。当然，Spring Security被明确设计成处理常见的需求，但是你最好别因为

这个目的使用项目领域模型安全功能。

小结

简单回顾一下，Spring Security主要由以下几部分组成的:

- `SecurityContextHolder`，提供几种访问 `SecurityContext` 的方式。
- `SecurityContext`，保存 `Authentication` 信息和请求对应的安全信息。
- `Authentication`，展示Spring Security特定的主体。
- `GrantedAuthority`，反应，在应用程序范围你，赋予主体的权限。
- `UserDetails`,通过你的应用DAO，提供必要的信息，构建Authentication对象。
- `UserDetailsService`，创建一个`UserDetails`，传递一个 `String` 类型的用户名(或者证书ID或其他).

现在，你应该对这种重复使用的组件有一些了解了。 让我们贴近看一下验证的过程。

验证

Spring Security可以在很多不同的认证环境下使用。虽然我们推荐人们使用Spring Security，不与已存在的容器管理认证系统结合，但它也是支持的-使用你自己的属性验证系统进行整合。

什么是Spring Security验证?

让我们考虑一个大家都很熟悉的标准的验证场景。

1. 提示用户输入用户名和密码进行登录。
2. 该系统 (成功) 验证该用户名的密码正确。
3. 获取该用户的环境信息 (他们的角色列表等).
4. 为用户建立安全的环境。
5. 用户进行，可能执行一些操作，这是潜在的保护的访问控制机制，检查所需权限，对当前的安全的环境信息的操作。

前三个项目构成的验证过程，所以我们将看看这些是如何发生在Spring Security中的。

1. 用户名和密码进行组合成一个实例`UsernamePasswordAuthenticationToken` (一个`Authentication`接口的实例, 我们之前看到的).
2. 令牌传递到`AuthenticationManager`实例进行验证。
3. 该`AuthenticationManager`完全填充`Authentication`实例返回成功验证。
4. 安全环境是通过调用`SecurityContextHolder.getContext().setAuthentication(…)`，传递到返回的验证对象建立的。

从这一点上来看，用户被认为是被验证的。让我们看看一些代码作为一个例子:

```
  import org.springframework.security.authentication.*;
import org.springframework.security.core.*;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;

public class AuthenticationExample {
private static AuthenticationManager am = new SampleAuthenticationManager();

public static void main(String[] args) throws Exception {
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

 while(true) {
 System.out.println("Please enter your username:");
 String name = in.readLine();
 System.out.println("Please enter your password:");
 String password = in.readLine();
 try {
  Authentication request = new UsernamePasswordAuthenticationToken(name, password);
  Authentication result = am.authenticate(request);
  SecurityContextHolder.getContext().setAuthentication(result);
  break;
 } catch(AuthenticationException e) {
  System.out.println("Authentication failed: " + e.getMessage());
 }
 }
 System.out.println("Successfully authenticated. Security context contains: " +
   SecurityContextHolder.getContext().getAuthentication());
}
}
```

```
class SampleAuthenticationManager implements AuthenticationManager {
static final List<GrantedAuthority> AUTHORITIES = new ArrayList<GrantedAuthority>();

static {
 AUTHORITIES.add(new SimpleGrantedAuthority("ROLE_USER"));
}

public Authentication authenticate(Authentication auth) throws AuthenticationException {
 if (auth.getName().equals(auth.getCredentials())) {
 return new UsernamePasswordAuthenticationToken(auth.getName(),
  auth.getCredentials(), AUTHORITIES);
 }
 throw new BadCredentialsException("Bad Credentials");
}
}
```

在这里我们已经写了一个小程序，要求用户输入一个用户名和密码并执行上述序列。这个 `AuthenticationManager` 我们这里将验证用户的用户名和密码将其设置成一样的，它给每一个用户分配一个单一的角色。从上面输出的将是类似的东西:

```
  Please enter your username:
bob
Please enter your password:
password
Authentication failed: Bad Credentials
Please enter your username:
bob
Please enter your password:
bob
Successfully authenticated. Security context contains: \
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@441d0230: \
Principal: bob; Password: [PROTECTED]; \
Authenticated: true; Details: null; \
Granted Authorities: ROLE_USER
```

请注意，你通常不需要写任何这样的代码。这个过程通常会发生在内部，以一个web认证过滤器为例，我们刚刚在这里的代码显示，在Spring Security中究竟是什么构成了验证的问题，有一个相对简单的答案。用户验证时， `SecurityContextHolder` 包含一个完全填充的 `Authentication` 对象的用户进行身份验证。

直接设置SecurityContextHolder的内容

事实上，Spring Security不介意你如何把 `Authentication` 对象包含在 `SecurityContextHolder` 内。唯一的关键要求是 `SecurityContextHolder` 包含 `Authentication` 在 `AbstractSecurityInterceptor` 之前(我们会看到更多的版本)需要用户授权操作。

你可以(很多用户都这样做)写一个自己的过滤器或MVC控制器来提供验证系统的交互，这些都不是基于Spring Security的。比如，你也许使用容器管理认证，从ThreadLocal或JNDI里获得当前用户信息。或者，你的公司可能有一个遗留系统，它是一个企业标准，你不能控制它。这种情况下，很容易让Spring Security工作，也能提供验证能力。你所需要的就是写一个过滤器(或等价物)从指定位置读取第三方用户信息，把它放到 `SecurityContextHolder` 里。在这种情况下，你还需要考虑的事情通常是由内置的认证基础设施自动照顾。例如，缓存请求的情况下你可能需要事先创建一个HTTP会话，在编写客户端响应之前 [1]。

如果你想知道 `AuthenticationManager` 是如何以现实世界的例子来实现，我们可以来看看 核心服务一章.

在Web应用程序中的身份验证

现在让我们来看看你在Web应用程序中使用Spring Security的情况(不启用 `web.xml` 安全性)。用户如何进行身份验证和建立安全环境?

考虑一个典型的Web应用程序的身份验证过程:

1. 你访问首页，点击一个链接。
2. 向服务器发送一个请求，服务器判断你是否在访问一个受保护的资源。
3. 如果你还没有进行过认证，服务器发回一个响应，提示你必须进行认证。响应可能是HTTP响应代码，或者是重新定向到一个特定的web页面。
4. 依据验证机制，你的浏览器将重定向到特定的web页面，这样你可以添加表单，或者浏览器使用其他方式校验你的身份（比如，一个基本校验对话框，cookie，或者X509证书，或者其他）。
5. 浏览器会发回一个响应给服务器。 这将是HTTP POST包含你填写的表单内容，或者是HTTP头部，包含你的验证信息。
6. 下一步，服务器会判断当前的证书是否是有效的， 如果他们是有效的，下一步会执行。 如果他们是非法的，通常你的浏览器会再尝试一次（所以你返回的步骤二）。
7. 你发送的原始请求，会导致重新尝试验证过程。有希望的是，你会通过验证，得到足够的授权，访问被保护的资源。如果你有足够的权限，请求会成功。否则，你会收到一个HTTP错误代码403，意思是访问被拒绝。

Spring Security使用鲜明的类负责上面提到的每个步骤。主要的部分是（为了使用他们） `ExceptionTranslationFilter`，一个 `AuthenticationEntryPoint` 一个验证机制, 我们在上一节看到它负责调用`AuthenticationManager`。

ExceptionTranslationFilter

`ExceptionTranslationFilter`是一个Spring Security过滤器，用来检测是否抛出了Spring Security异常。这些异常会被`AbstractSecurityInterceptor`抛出，它主要用来提供验证服务。我们会在下一节讨论`AbstractSecurityInterceptor`，但是现在，我们只需要知道，它是用来生成Java，并且要知道和HTTP没什么关系，或者如何验证一个主体。而`ExceptionTranslationFilter`提供这些服务，使用特点那个的响应，返回错误代码403(如果主体被验证了，但是权限不足-在上边的步骤七),或者启动一个`AuthenticationEntryPoint`(如果主体没有被认证，然后我们需要进入步骤三)。

AuthenticationEntryPoint

`AuthenticationEntryPoint`对应上面列表中的步骤三。如你所想的，每个web应用程序都有默认的验证策略(好的，这可以在Spring Security里配置一切，但是让我们现在保持简单)。每个主要验证系统会有它自己的`AuthenticationEntryPoint`实现， 会执行动作，如同步骤三里的描述一样。

验证机制

在你的浏览器决定提交你的认证证书之后(使用HTTP表单发送或者是HTTP头),服务器部分需要有一些东西来"收集"这些验证信息。现在我们到了上述的第六步。 在Spring Security里，我们需要一个特定的名字，来描述从用户代码（通常是浏览器）收集验证信息的功能，这个名字就是"验证机制"。实例是窗体的基本登录和基本的身份验证。一旦认证细节已从用户代理收集,建立一个`Authentication` "request"对象，然后提交给`AuthenticationManager`。

验证机制重新获得了组装好的`Authentication`对象时，它会认为请求有效，把`Authentication`放到`SecurityContextHolder`里的，然后导致原始请求重审(第七步)。另一方面,如果`AuthenticationManager`驳回了请求,验证机制会让用户代码重试(第二步)。

Storing the SecurityContext between requests

根据不同的应用程序类型，在用户操作的过程中需要有合适的策略来保存security信息。在一个典型的web应用中，一个用户登录系统之后就会被一个特有的session Id所唯一标识,服务器会将session作用期间的principal数据保存在缓存中。在Spring Security中,保存`SecurityContext`的任务落在了`SecurityContextPersistenceFilter`身上，它默认将上下文当做`HttpSession`属性保存在HTTP请求中,并且将每一个请求的上下文保存在`SecurityContextHolder`中，最重要的功能,是在请求结束之后,清理`SecurityContextHolder`。你不需要处于安全的目的直接和`HttpSession`打交道。在这里仅仅只是不需要那样做-总是使用`SecurityContextHolder`来代替`HttpSession`。

许多其他的应用（举个例子：一个无状态的RESTful风格web服务）不使用Http Session并且每次请求过来都会进行验证。然而比较重要的是:`SecurityContextPersistenceFilter`被包含在过滤器链中,并确保每次请求完毕之后清理`SecurityContextHolder`。

> 其中有一个应用程序接收一个会话的并发请求,同样的`SecurityContext`实例将线程之间共享。即使正在使用`ThreadLocal`,它是相同的实例,从每个线程的`HttpSession`检索。如果你希望暂时改变一个线程正在运行的上下文这很有意义。如果你只是使用`SecurityContextHolder.getContext()`,和调用`setAuthentication(anAuthentication)`返回的上下文对象,那么`Authentication`对象将在*全部*并发线程共享相同的`SecurityContext`情况的变化。 你可以自定义`SecurityContextPersistenceFilter`的行为,为每一个请求创建一个完全新的`SecurityContext`,防止在一个线程的变化影响另一个。或者,你可以创建一个新的实例,只是在这个点上,你暂时改变了背景。方法`SecurityContextHolder.createEmptyContext()`总是返回一个新的上下文实例。

Spring Security的访问控制(授权)

负责Spring Security访问控制决策的主要接口是`AccessDecisionManager`。它有一个`decide`方法，它需要一个`Authentication`对象请求访问,一个"secure object"(见下文)和安全元数据属性的列表适用的对象(如一个列表哪些角色需要被访问授权)。

安全和AOP建议

如果你熟悉AOP的话，就会知道有几种不同的拦截方式：之前，之后，抛异常和环绕。 其中环绕是非常有用的，因为advisor可以决定是否执行这个方法，是否修改返回的结果，是否抛出异常。 Spring Security为方法调用提供了一个环绕advice，就像web请求一样。 我们使用Spring的标准AOP支持制作了一个处理方法调用的环绕advice，我们使用标准Filter建立了对web请求的环绕advice。

对那些不熟悉AOP的人，需要理解的关键问题是Spring Security可以帮助你保护方法的调用，就像保护web请求一样。大多数人对保护服务层里的安全方法非常按兴趣。这是因为在目前这一代J2EE程序里，服务器放了更多业务相关的逻辑。如果你只是需要保护服务层的方法调用，Spring标准AOP平台就够了。如果你想直接保护领域对象，你会发现AspectJ非常值得考虑。

可以选择使用AspectJ还是SpringAOP处理方法验证，或者你可以选择使用filters处理web请求验证。 你可以不选，选择其中一个，选择两个，或者三个都选。主流的应用是处理一些web请求验证，再结合一些在服务层里的Spring AOP方法调用验证。

安全对象和AbstractSecurityInterceptor

那么什么*是*一个"安全对象"呢?Spring Security使用术语是指可以有安全性的任何对象(如授权决策)应用于它。最常见的例子就是方法调用和web请求。

Spring Security支持的每个安全对象类型都有它自己的类型,他们都是`AbstractSecurityInterceptor`的子类。很重要的是,如果主体是已经通过了验证,在`AbstractSecurityInterceptor`被调用的时候,`SecurityContextHolder`将会包含一个有效的`Authentication`。

`AbstractSecurityInterceptor`提供了一套一致的工作流程，来处理对安全对象的请求，通常是:

1. 查找当前请求里分配的"配置属性"。
2. 把安全对象，当前的`Authentication`和配置属性,提交给`AccessDecisionManager`来进行以此认证决定。
3. 有可能在调用的过程中,对`Authentication`进行修改。
4. 允许安全对象进行处理（假设访问被允许了）。
5. 在调用返回的时候执行配置的`AfterInvocationManager`。如果调用引发异常,`AfterInvocationManager`将不会被调用。

配置属性是什么?

一个"配置属性"可以看做是一个字符串,它对于`AbstractSecurityInterceptor`使用的类是有特殊含义的。它们由框架内接口`ConfigAttribute`表示。它们可能是简单的角色名称或拥有更复杂的含义,这就与`AccessDecisionManager`实现的先进程度有关了。`AbstractSecurityInterceptor`和配置在一起的`SecurityMetadataSource`用来为一个安全对象搜索属性。通常这个属性对用户是不可见的。配置属性将以注解的方式设置在受保护方法上，或者作为受保护URLs的访问属性。例如,当我们看到像`<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/>`命名空间中的介绍,这是说配置属性`ROLE_A`和`ROLE_B`适用于匹配Web请求的特定模式。在实践中,使用默认的`AccessDecisionManager`配置, 这意味着,任何人谁拥有`GrantedAuthority`只要符合这两个属性将被允许访问。严格来说,它们只是依赖于`AccessDecisionManager`实施的属性和解释。使用前缀`ROLE_`是一个标记,以表明这些属性是角色,应该由Spring Security的`RoleVoter`前缀被消耗掉。这只是使用`AccessDecisionManager`的选择基础。我们将在授权章看到`AccessDecisionManager`是如何实现的。

RunAsManager

假设`AccessDecisionManager`决定允许执行这个请求,`AbstractSecurityInterceptor`会正常执行这个请求。话虽如此，罕见情况下，用户可能需要把`SecurityContext`的`Authentication`换成另一个`Authentication`，这是由`AccessDecisionManager`调用`RunAsManager`。这也许在,有原因,不常见的情况下有用,比如服务层方法需要调用远程系统表现不同的身份。 因为Spring Security自动传播安全身份，从一个服务器到另一个（假设你使用了配置好的RMI或者HttpInvoker远程调用协议客户端），就可以用到它了。

AfterInvocationManager

按照下面安全对象执行和返回的方式-可能意味着完全的方法调用或过滤器链的执行-在`AbstractSecurityInterceptor`得到一个最后的机会来处理调用。这种状态下`AbstractSecurityInterceptor`对有可能修改返回对象感兴趣。你可能想让它发生，因为验证决定不能“关于如何在”一个安全对象调用。高可插拔性,`AbstractSecurityInterceptor`通过控制`AfterInvocationManager`在实际需要的时候修改对象。这里类实际上可能替换对象，或者抛出异常，或者什么也不做。如果调用成功后，检查调用才会执行。如果出现异常，额外的检查将被跳过。

`AbstractSecurityInterceptor`和它的相关对象 Security interceptors and the "secure object" model

Figure 1. Security interceptors and the "secure object" model

扩展安全对象模型

只有当开发人员考虑一个全新的拦截方法和授权请求时才需要直接使用安全对象。例如，为了确保对消息系统的调用，它有可能建立建立一个新的安全对象。任何东西都需要安全，并且还提供了一种方法去调用（如建议语义的AOP）能够被做成一个安全对象。不得不说的是，大多数Spring应用程序将只使用三种目前完全支持的安全对象类型(AOP Alliance `MethodInvocation`, AspectJ `JoinPoint`和web请求`FilterInvocation`)。

Localization

Spring Security支持终端用户看到异常消息的本地化。如果你的应用程序是专为讲英语的用户设计的，你不需要做任何事情，因为默认所有的安全信息都是英文的，如果你需要支持其他地方，你需要知道的一切都被包含在这部分。

所有异常消息都可以本地化，包括有关验证失败和访问被拒绝（授权失败）的消息。这主要集中在开发者和系统发布（包括不正确的属性，接口违反合同，使用不正确的构造器，开始验证，日志调试等级）异常和日志消息没有本地化，而是使用硬编码的

Spring Security的英文代码。

在 `spring-security-core-xx.jar` 的运输中你会发现一个 `org.springframework.security` 包含了 `messages.properties` 文件,以及一些常用版本的本地化语言。这应该是你的 `ApplicationContext`,因为Spring Security实现了Spring的 `MessageSourceAware` 界面，希望这些消息是依赖于应用程序上下文启动的时候注入。通常你需要做的是创建你的应用程序上下文参考消息里面的bean。一个例子如下所示：

```
  <bean id="messageSource"
 class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
<property name="basename" value="classpath:org/springframework/security/messages"/>
</bean>
```

该 `messages.properties` 是按照标准的资源束命名方式，为Spring Security的消息所支持的默认语言。这个默认的文件是英文的。

如果您希望自定义 `messages.properties` 文件，或支持其他语言，您应该复制该文件，相应地重命名它，并在上面的bean定义中注释它。在这个文件中没有大量的消息密钥，因此本地化不应该被认为是一个重大举措。如果你对这个文件执行定为操作，请考虑与社区分享你的工作通过记录JIRA任务和附加被你恰当命名的 `messages.properties` 本地化版本。

Spring Security依赖于Spring"s的本地化支持，以实际查找适当的消息。为了这项工作，你必须确保从传入请求的区域存储在Spring's`org.springframework.context.i18n.LocaleContextHolder`。Spring MVC的` DispatcherServlet 会自动为你的程序做，但因为Spring Security的过滤器在那之前被调用，`localecontextholder` 需在过滤器被呼叫之前建立在包含正确的 `Locale` 里。你也可以在你自己的过滤器里面做这个（必须做完这项在Spring Security的 `web.xml` 过滤之前）或者你可以使用Spring的 `RequestContextFilter` 。请参阅Spring Framework文档，以进一步详细说明使用Spring定位。

“联系人”示例应用程序设置为使用本地化消息。

核心服务

现在，我们对Spring Security的架构和核心类进行高级别的概述，让我们在一个或两个核心接口及其实现的仔细看看，尤其是 `AuthenticationManager` ， `UserDetailsService` 和 `AccessDecisionManager` 这些东西的信息都在这个文档的里面，所以这一点很重要，你要知道他们是如何配置如何操作的。

The AuthenticationManager, ProviderManager and AuthenticationProvider

该 `AuthenticationManager` 只是一个接口，这样的实现可以是我们选择的任何东西，但它是如何在实践中运作的?如果我们需要检查多个授权数据库或者将不同的授权服务结合起来，类似数据库和LDAP服务器?

Spring Security的默认实现被称为 `ProviderManager` 而非处理身份验证请求本身，它委托给一个列表去配置 `AuthenticationProvider` ，其中每个查询反过来，看它是否能进行认证。每个提供程序都将抛出一个异常或返回一个完全填充的身份验证对象。还记得我们的好朋友， `UserDetails` 和 `UserDetailsService` 吗?如果没有，回到前面的章节刷新你的记忆。到验证的认证请求的最常见的方法是加载相应 `UserDetails` 并针对已经由用户输入所述一个检查加载密码。这是由 `DaoAuthenticationProvider` 所使用的方法（见下文）。加载的 `UserDetails` 对象-尤其是 `GrantedAuthority` 的IT包含-建设是返回一个成功验证，并存储在 `SecurityContext` 完全填充 `Authentication` 对象时，将被使用。

如果你使用的命名空间，创建并在内部进行维护 `ProviderManager` 的一个实例，您可以通过使用命名空间身份验证提供元素添加提供商。(see 命名空间章节)。在这种情况下，你不应该声明在应用程序上下文中的 `ProviderManager` bean。但是，如果你没有使用命名空间，那么你会这样声明:

```
  <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>
```

在上面的例子中，我们有三个提供者。它们试图在顺序显示(它是通过使用一个 `List` 的暗示)，每个提供者都能尝试验证，或者通过简单的返回 `null` 跳过认证。如果所有的实现都返回 `null` ，则 `ProviderManager` 将抛出一个 `ProviderNotFoundException` 。如果你有兴趣了解更多的有关提供者，请参考 `ProviderManager` 的JavaDocs。

身份验证机，如Web表单登录处理过滤器被注入到 `ProviderManager` 的引用，将调用它来处理自己的身份验证请求。你需要的供应商有时可以与认证机制互换，而在其他时间，他们将依赖于特定的认证机制。例如， `DaoAuthenticationProvider` 和 `LdapAuthenticationProvider` 给它提交一个简单的用户名/密码验证请求，并因此将与基于表单登录或HTTP基本验证工作的机制兼容。另一方面，一些认证机制创建只能由单一类型 `AuthenticationProvider` 解

释的认证请求对象。这一方面的一个例子是JA-SIG CAS，它使用一个服务票据的概念，因此可以仅通过一个 `CasAuthenticationProvider` 进行认证。你不必太在意这一点，因为如果你忘记注册一个合适的供应商，你会简单地收到一个 `ProviderNotFoundException` 不进行认证的尝试。

清楚成功认证的凭据

默认情况下(从Spring Security 3.1开始)的 `ProviderManager` 将试图清除它返回一个成功的认证请求的Authentication`对象的任何敏感的身份验证信息。这可以防止密码等个人资料超过保留时间。

当使用用户对象的高速缓存时，例如，改善在无状态情况下应用程序的性能，这可能导致问题。如果 `Authentication` 包含在高速缓存（诸如 `UserDetails` 实例）的对象的引用中，将其凭证移除，则它将不再能够进行对缓存的值进行验证。你需要考虑到这一点，如果你使用的是高速缓存。一个显而易见的解决方案是让一个对象的副本，无论是在高速缓存中执行或在 `AuthenticationProvider` 它创建返回 `Authentication` 对象。另外，你可以在 `ProviderManager` 中禁用 `eraseCredentialsAfterAuthentication` 。查看Javadoc了解更多信息。

DaoAuthenticationProvider

Spring Security中实现最简单的 `AuthenticationProvider` 是 `DaoAuthenticationProvider` ，也是最早支持的框架。它利用了 `UserDetailsService` （作为DAO）去查找用户名和密码。它的用户进行身份验证通过 `userdetailsservice` 加载 `usernamepasswordauthenticationtoken `提交密码进行一对一的比较。配置提供程序是非常简单的:

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

这个 `PasswordEncoder` 是可选的。一个 `PasswordEncoder` 提供编码以及 `UserDetails` 对象提出的密码是从配置 `UserDetailsService` 返回的解码。 这将更加详细 如下。

UserDetailsService实现

本参考指南早些时候提到的，大多数的认证供应商利用的` userdetails 和 userdetailsservice 接口。回想一下, `UserDetailsService` 是一个方法:

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

返回的 `UserDetails` 是提供给getters的一个接口，以保证非空的认证信息，例如，用户名，密码，授权和用户帐户是否被启用或禁用。大多数认证供应商将使用 `UserDetailsService` ，即使用户名和密码不作为认证决定的一部分。他们可以使用返回的 `UserDetails` 对象为其 `GrantedAuthority` 信息对象，因为其他的一些系统（如LDAP或X.509或CAS等）承担了实际验证凭证的的责任。

鉴于 `UserDetailsService` 就是这么简单实现的，它应该便于用户检索使用自己选择的持久化策略的认证信息。话虽如此，Spring Security确实包括了许多有用的基本实现，我们将在下面看到。

在内存认证

简单的使用去创建一个自定义的 `UserDetailsService` 实现选择从一个持久性引擎中提取信息，但许多应用程序不需要这么复杂。尤其是如果你正在建设一个原型应用或刚刚开始结合Spring Security当你真的不想花时间配置数据库或写作 `userdetailsservice` 实现。对于这种情况，一个简单的选项是使用安全性 命名空间的 `user-service` 元素:

```
 <user-service id="userDetailsService">
<user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
<user name="bob" password="bobspassword" authorities="ROLE_USER" />
</user-service>
```

这也支持一个外部属性文件的使用:

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

属性文件应包含在表单条目

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

例如

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

JdbcDaoImpl

Spring Security还包括 `UserDetailsService` ，它可以从一个JDBC数据源获得认证信息。内部Spring JDBC的使用，避免了一个全功能对象关系映射（ORM）的复杂性来存储用户信息。如果你的应用程序不使用ORM工具，你可以写一个自定义 `UserDetailsService` 重用在你可能已经创建好的映射文件上。回到 `JdbcDaoImpl` ，实例的配置如下：:

```
  <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>
```

您可以通过修改上面的`DriverManagerDataSource`使用不同的关系型数据库管理系统。你也可以从JNDI获得，与任何其他的Spring配置使用一个全球性的数据源。

Authority Groups

默认情况下，`JdbcDaoImpl`加载权限直接映射到用户的角色(见 数据库架构附录)。另一种方法是将权限分成组并分配组给用户。有些人喜欢这种方式作为管理用户权限的一种手段。见 `JdbcDaoImpl` Javadoc获得如何能够使用权限组的更多信息。该组架构也包括在附录中。

Password Encoding

Spring Security的 `PasswordEncoder`接口用于支持密码以某种方式在持久存储中进行编码。你不应该在纯文本中存储密码。总是使用单向密码算法如BCrypt使用内置的混淆值，对于每个存储的密码都是不同的。不要使用普通的哈希函数，如MD5或SHA，甚至是一个混淆的版本。BCrypt是故意设计成慢，用于阻碍离线密码破解，而标准的散列算法是快速和能轻易地被用来测试在并行密码定制的硬件上。你可能会认为这并不适用于你，因为你的密码数据库是安全的，和离线攻击不是一个风险线上的。如果是这样的话，做一些研究并阅读所有的高知名度、一直被嘲笑存储密码不安全并已妥协的网站。最好是在安全的一边。对于安全性使用`org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"`是一个不错的选择。在其他常见的编程语言中也有兼容的实现，所以对于互相操作性它也是一个很好的选择。

如果你使用的是已经有哈希密码的遗留系统，那么你需要使用一个解码器来匹配你当前的算法，至少要等到你可以将用户迁移到一个更安全的方案（通常这将涉及到要求用户设置一个新的密码，因为哈希值是不可逆的）。Spring Security具有包含传统的密码编码功能的实现，即`org.springframework.security.authentication.encoding`包。
该`DaoAuthenticationProvider`可与新的或旧的`PasswordEncoder`类型注入。

什么是hash?

密码散列对于Spring Security不是唯一的，但是一个共同的来源是对于不熟悉概念的用户。哈希（或摘要）算法是一个单向函数，它由一些输入数据，如密码产生一块固定长度的输出数据（散列）的。作为一个例子，字符串"密码"（十六进制）的MD5哈希。

```
5f4dcc3b5aa765d61d8327deb882cf99
```

从这一方面来说hash的是"单向"，要获得给定的散列值这是很困难的(实际上是不可能的)，或者说实际上任何可能的输出都会产生散列值的原始输入。这个属性使得哈希值对于进行身份验证是非常有用的。它们可以被存储在用户数据库作为替代明文口令，即使值受到损害它们也不会立刻露出可用于登录的密码。请注意，这也意味着一旦它被编码，你就没有办法恢复密码。

在Hash中添加Salt

随着密码的哈希值的使用有一个潜在的问题就是如果一个普通的词用于输入，则相对容易得到哈希的单向属性。人们倾向于选择类似的密码和巨大的字典，这些以前被黑客攻击的网站都可以在线。例如，如果你使用谷歌搜索哈希值`5f4dcc3b5aa765d61d8327deb882cf99`，你会很快发现原来的"密码"。以类似的方式，攻击者可以构建散列的字典从标准单词列表中查找原来的密码。有一种方法帮助防止这种情况就是有一个合适的强密码策略，以防止常用的单词被使用。另一个是计算散列何时使用"salt"。这是它与计算散列之前密码组合的每个用户的已知数据的附加字符串。理想情况下，数据应该是随机的，但在实践中，任何salt值通常是最好没有。使用的salt意味着攻击者必须建立散列每个salt值的一个单独的字典，使得攻击更加复杂（但不是不可能）。

Bcrypt自动生成每个密码随机salt值时，它被编码，并存储在一个标准格式的BCrypt的字符串。

传统的方法来处理salt是注入一个`SaltSource`到`DaoAuthenticationProvider`，这将获得一个特定的用户salt值并将其传递到`PasswordEncoder`。使用BCrypt意味着你不用担心salt处理的细节（如值存储在什么地方），因为它是在内部完成的。所以，我们强烈建议你使用BCrypt，除非你已经有一个系统分开其中所存储的salt。

Hashing and Authentication

当一个认证供应商（如Spring Security的`DaoAuthenticationProvider`）需要对某个用户的已知值提交的认证请求来检查密码和存储的密码以某种方式进行编码，然后提交的值必须准确使用编码相同的算法。这个取决于你的检查，这些都是兼容的，因为Spring Security没有对持久值的控制。如果添加的密码散列以你的身份验证配置Spring Security的，和你的数据库中包含明文

密码，那么就没有办法验证成功。即使你知道你的数据库是使用MD5编码的密码，例如，你的应用程序被配置为使用Spring Security的`Md5PasswordEncoder`，那么有很多事情会是错误的。数据库可以具有在Base 64编码的密码，例如当编码器使用十六进制字符串（缺省值）。替代地数据库可以是使用大写，而从编码器的输出为小写。为了确保，你需要写一个测试，以检查输出从你配置的密码编码器与一个已知的密码和salt的组合，并检查它相匹配的数据库值之前，进一步尝试通过您的应用程序进行验证。使用诸如BCrypt标准将避免这些问题。

如果你想生成您的用户数据库编码密码，直接在Java中进行存储，那么你可以使用`PasswordEncoder`上的`encode`方法。

Testing

This section describes the testing support provided by Spring Security.

To use the Spring Security test support, you must include `spring-security-test-{spring-security-version}.jar` as a dependency of your project.

Testing Method Security

This section demonstrates how to use Spring Security's Test support to test method based security. We first introduce a `MessageService` that requires the user to be authenticated in order to access it.

```
 public class HelloMessageService implements MessageService {

@PreAuthorize("authenticated")
public String getMessage() {
 Authentication authentication = SecurityContextHolder.getContext()
             .getAuthentication();
  return "Hello " + authentication;
 }
}
```

The result of `getMessage` is a String saying "Hello" to the current Spring Security `Authentication`. An example of the output is displayed below.

```
 Hello org.springframework.security.authentication.UsernamePasswordAuthenticationToken@ca25360: Principal:
org.springframework.security.core.userdetails.User@36ebcb: Username: user; Password: [PROTECTED]; Enabled: true;
AccountNonExpired: true; credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities: ROLE_USER; Credentials:
[PROTECTED]; Authenticated: true; Details: null; Granted Authorities: ROLE_USER
```

Security Test Setup

Before we can use Spring Security Test support, we must perform some setup. An example can be seen below:

```
 @RunWith(SpringJUnit4ClassRunner.class) (1)
@ContextConfiguration (2)
public class WithMockUserTests {
```

This is a basic example of how to setup Spring Security Test. The highlights are:

(1) `@RunWith` instructs the spring-test module that it should create an ApplicationContext This is no different than using the existing Spring Test support. For additional information, refer to the Spring Reference

(2) `@ContextConfiguration` instructs the spring-test the configuration to use to create the `ApplicationContext`. Since no configuration is specified, the default configuration locations will be tried. This is no different than using the existing Spring Test support. For additional information, refer to the Spring Reference

Spring Security hooks into Spring Test support using the `WithSecurityContextTestExecutionListener` which will ensure our tests are ran with the correct user. It does this by populating the `SecurityContextHolder` prior to running our tests. After the test is done, it will clear out the `SecurityContextHolder`. If you only need Spring Security related support, you can replace `@ContextConfiguration` with `@SecurityExecutionListeners`.

Remember we added the `@PreAuthorize` annotation to our `HelloMessageService` and so it requires an authenticated user to invoke it. If we ran the following test, we would expect the following test will pass:

```
 @Test(expected = AuthenticationCredentialsNotFoundException.class)
public void getMessageUnauthenticated() {
 messageService.getMessage();
}
```

@WithMockUser

The question is "How could we most easily run the test as a specific user?" The answer is to use `@WithMockUser`. The following test will be ran as a user with the username "user", the password "password", and the roles "ROLE_USER".

```
 @Test
@WithMockUser
public void getMessageWithMockUser() {
String message = messageService.getMessage();
...
}
```

Specifically the following is true:

- The user with the username "user" does not have to exist since we are mocking the user
- The `Authentication` that is populated in the `SecurityContext` is of type `UsernamePasswordAuthenticationToken`
- The principal on the `Authentication` is Spring Security's `User` object
- The `User` will have the username of "user", the password "password", and a single `GrantedAuthority` named "ROLE_USER" is used.

Our example is nice because we are able to leverage a lot of defaults. What if we wanted to run the test with a different username? The following test would run with the username "customUser". Again, the user does not need to actually exist.

```
 @Test
@WithMockUser("customUsername")
public void getMessageWithMockUserCustomUsername() {
 String message = messageService.getMessage();
...
}
```

We can also easily customize the roles. For example, this test will be invoked with the username "admin" and the roles "ROLE_USER" and "ROLE_ADMIN".

```
 @Test
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public void getMessageWithMockUserCustomUser() {
 String message = messageService.getMessage();
 ...
}
```

If we do not want the value to automatically be prefixed with ROLE_ we can leverage the authorities attribute. For example, this test will be invoked with the username "admin" and the authorities "USER" and "ADMIN".

```
 @Test
@WithMockUser(username = "admin", authorities = { "ADMIN", "USER" })
public void getMessageWithMockUserCustomAuthorities() {
 String message = messageService.getMessage();
 ...
}
```

Of course it can be a bit tedious placing the annotation on every test method. Instead, we can place the annotation at the class level and every test will use the specified user. For example, the following would run every test with a user with the username "admin", the password "password", and the roles "ROLE_USER" and "ROLE_ADMIN".

```
 @RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public class WithMockUserTests {
```

@WithAnonymousUser

Using `@WithAnonymousUser` allows running as an anonymous user. This is especially convenient when you wish to run most of your tests with a specific user, but want to run a few tests as an anonymous user. For example, the following will run withMockUser1 and withMockUser2 using @WithMockUser and anonymous as an anonymous user.

```
 @RunWith(SpringJUnit4ClassRunner.class)
@WithMockUser
public class WithUserClassLevelAuthenticationTests {
```

```
 @Test
 public void withMockUser1() {
 }

 @Test
 public void withMockUser2() {
 }

 @Test
 @WithAnonymousUser
 public void anonymous() throws Exception {
  // override default to run as anonymous user
 }
}
```

@WithUserDetails

While `@WithMockUser` is a very convenient way to get started, it may not work in all instances. For example, it is common for applications to expect that the `Authentication` principal be of a specific type. This is done so that the application can refer to the principal as the custom type and reduce coupling on Spring Security.

The custom principal is often times returned by a custom `UserDetailsService` that returns an object that implements both `UserDetails` and the custom type. For situations like this, it is useful to create the test user using the custom `UserDetailsService`. That is exactly what `@WithUserDetails` does.

Assuming we have a `UserDetailsService` exposed as a bean, the following test will be invoked with an `Authentication` of type `UsernamePasswordAuthenticationToken` and a principal that is returned from the `UserDetailsService` with the username of "user".

```
  @Test
@WithUserDetails
public void getMessageWithUserDetails() {
 String message = messageService.getMessage();
 ...
}
```

We can also customize the username used to lookup the user from our `UserDetailsService`. For example, this test would be executed with a principal that is returned from the `UserDetailsService` with the username of "customUsername".

```
  @Test
@WithUserDetails("customUsername")
public void getMessageWithUserDetailsCustomUsername() {
 String message = messageService.getMessage();
 ...
}
```

We can also provide an explicit bean name to look up the `UserDetailsService`. For example, this test would look up the username of "customUsername" using the `UserDetailsService` with the bean name "myUserDetailsService".

```
  @Test
@WithUserDetails(value="customUsername", userDetailsServiceBeanName="myUserDetailsService")
public void getMessageWithUserDetailsServiceBeanName() {
 String message = messageService.getMessage();
 ...
}
```

Like `@WithMockUser` we can also place our annotation at the class level so that every test uses the same user. However unlike `@WithMockUser`, `@WithUserDetails` requires the user to exist.

@WithSecurityContext

We have seen that `@WithMockUser` is an excellent choice if we are not using a custom `Authentication` principal. Next we discovered that `@WithUserDetails` would allow us to use a custom `UserDetailsService` to create our `Authentication` principal but required the user to exist. We will now see an option that allows the most flexibility.

We can create our own annotation that uses the `@WithSecurityContext` to create any `SecurityContext` we want. For example, we might create an annotation named `@WithMockCustomUser` as shown below:

```
  @Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class)
public @interface WithMockCustomUser {

 String username() default "rob";
```

```
 String name() default "Rob Winch";
}
```

You can see that `@WithMockCustomUser` is annotated with the `@WithSecurityContext` annotation. This is what signals to Spring Security Test support that we intend to create a `SecurityContext` for the test. The `@WithSecurityContext` annotation requires we specify a `SecurityContextFactory` that will create a new `SecurityContext` given our `@WithMockCustomUser` annotation. You can find our `WithMockCustomUserSecurityContextFactory` implementation below:

```
 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;
 }
}
```

We can now annotate a test class or a test method with our new annotation and Spring Security's `WithSecurityContextTestExecutionListener` will ensure that our `SecurityContext` is populated appropriately.

When creating your own `WithSecurityContextFactory` implementations, it is nice to know that they can be annotated with standard Spring annotations. For example, the `WithUserDetailsSecurityContextFactory` uses the `@Autowired` annotation to acquire the `UserDetailsService`:

```
 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;
 }
}
```

Test Meta Annotations

If you reuse the same user within your tests often, it is not ideal to have to repeatedly specify the attributes. For example, if there are many tests related to an administrative user with the username "admin" and the roles `ROLE_USER` and `ROLE_ADMIN` you would have to write:

```
 @WithMockUser(username="admin",roles={"USER","ADMIN"})
```

Rather than repeating this everywhere, we can use a meta annotation. For example, we could create a meta annotation named `WithMockAdmin`:

```
 @Retention(RetentionPolicy.RUNTIME)
@WithMockUser(value="rob",roles="ADMIN")
public @interface WithMockAdmin { }
```

Now we can use `@WithMockAdmin` in the same way as the more verbose `@WithMockUser`.

Meta annotations work with any of the testing annotations described above. For example, this means we could create a meta annotation for `@WithUserDetails("admin")` as well.

Spring MVC Test Integration

Spring Security provides comprehensive integration with Spring MVC Test

Setting Up MockMvc and Spring Security

In order to use Spring Security with Spring MVC Test it is necessary to add the Spring Security `FilterChainProxy` as a `Filter`. It is also necessary to add Spring Security's `TestSecurityContextHolderPostProcessor` to support Running as a User in Spring MVC Test with Annotations. This can be done using Spring Security's `SecurityMockMvcConfigurers.springSecurity()`. For example:

Spring Security's testing support requires spring-test-4.1.3.RELEASE or greater.

```
 import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class CsrfShowcaseTests {

 @Autowired
 private WebApplicationContext context;

 private MockMvc mvc;

 @Before
 public void setup() {
  mvc = MockMvcBuilders
    .webAppContextSetup(context)
    .apply(springSecurity()) (1)
    .build();
 }

...
```

(1) `SecurityMockMvcConfigurers.springSecurity()` will perform all of the initial setup we need to integrate Spring Security with Spring MVC Test

SecurityMockMvcRequestPostProcessors

Spring MVC Test provides a convenient interface called a `RequestPostProcessor` that can be used to modify a request. Spring Security provides a number of `RequestPostProcessor` implementations that make testing easier. In order to use Spring Security's `RequestPostProcessor` implementations ensure the following static import is used:

```
 import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

Testing with CSRF Protection

When testing any non safe HTTP methods and using Spring Security's CSRF protection, you must be sure to include a valid CSRF Token in the request. To specify a valid CSRF token as a request parameter using the following:

```
 mvc
 .perform(post("/").with(csrf()))
```

If you like you can include CSRF token in the header instead:

```
 mvc
 .perform(post("/").with(csrf().asHeader()))
```

You can also test providing an invalid CSRF token using the following:

```
 mvc
 .perform(post("/").with(csrf().useInvalidToken()))
```

Running a Test as a User in Spring MVC Test

It is often desirable to run tests as a specific user. There are two simple ways of populating the user:

- Running as a User in Spring MVC Test with RequestPostProcessor
- Running as a User in Spring MVC Test with Annotations

Running as a User in Spring MVC Test with RequestPostProcessor

There are a number of options available to associate a user to the current `HttpServletRequest`. For example, the following will run as a user (which does not need to exist) with the username "user", the password

"password", and the role "ROLE_USER":

> The support works by associating the user to the `HttpServletRequest`. To associate the request to the `SecurityContextHolder` you need to ensure that the `SecurityContextPersistenceFilter` is associated with the `MockMvc` instance. A few ways to do this are:
>
> - Invoking apply(springSecurity())
> - Adding Spring Security's `FilterChainProxy` to `MockMvc`
> - Manually adding `SecurityContextPersistenceFilter` to the `MockMvc` instance may make sense when using `MockMvcBuilders.standaloneSetup`

```
mvc
.perform(get("/").with(user("user")))
```

You can easily make customizations. For example, the following will run as a user (which does not need to exist) with the username "admin", the password "pass", and the roles "ROLE_USER" and "ROLE_ADMIN".

```
mvc
.perform(get("/admin").with(user("admin").password("pass").roles("USER","ADMIN")))
```

If you have a custom `UserDetails` that you would like to use, you can easily specify that as well. For example, the following will use the specified `UserDetails` (which does not need to exist) to run with a `UsernamePasswordAuthenticationToken` that has a principal of the specified `UserDetails`:

```
mvc
.perform(get("/").with(user(userDetails)))
```

You can run as anonymous user using the following:

```
mvc
.perform(get("/").with(anonymous()))
```

This is especially useful if you are running with a default user and wish to execute a few requests as an anonymous user.

If you want a custom `Authentication` (which does not need to exist) you can do so using the following:

```
mvc
.perform(get("/").with(authentication(authentication)))
```

You can even customize the `SecurityContext` using the following:

```
mvc
.perform(get("/").with(securityContext(securityContext)))
```

We can also ensure to run as a specific user for every request by using `MockMvcBuilders`'s default request. For example, the following will run as a user (which does not need to exist) with the username "admin", the password "password", and the role "ROLE_ADMIN":

```
mvc = MockMvcBuilders
 .webAppContextSetup(context)
 .defaultRequest(get("/").with(user("user").roles("ADMIN")))
 .apply(springSecurity())
 .build();
```

If you find you are using the same user in many of your tests, it is recommended to move the user to a method. For example, you can specify the following in your own class named `CustomSecurityMockMvcRequestPostProcessors`:

```
 public static RequestPostProcessor rob() {
return user("rob").roles("ADMIN");
}
```

Now you can perform a static import on `SecurityMockMvcRequestPostProcessors` and use that within your tests:

```
 import static sample.CustomSecurityMockMvcRequestPostProcessors.*;

...

mvc
 .perform(get("/").with(rob()))
```

Running as a User in Spring MVC Test with Annotations

As an alternative to using a `RequestPostProcessor` to create your user, you can use annotations described in Testing Method Security. For example, the following will run the test with the user with username "user", password "password", and role "ROLE_USER":

```
  @Test
@WithMockUser
public void requestProtectedUrlWithUser() throws Exception {
mvc
  .perform(get("/"))
  ...
}
```

Alternatively, the following will run the test with the user with username "user", password "password", and role "ROLE_ADMIN":

```
  @Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
mvc
  .perform(get("/"))
  ...
}
```

Testing HTTP Basic Authentication

While it has always been possible to authenticate with HTTP Basic, it was a bit tedious to remember the header name, format, and encode the values. Now this can be done using Spring Security's `httpBasic` `RequestPostProcessor`. For example, the snippet below:

```
  mvc
 .perform(get("/").with(httpBasic("user","password")))
```

will attempt to use HTTP Basic to authenticate a user with the username "user" and the password "password" by ensuring the following header is populated on the HTTP Request:

```
  Authorization: Basic dXNlcjpwYXNzd29yZA==
```

SecurityMockMvcRequestBuilders

Spring MVC Test also provides a `RequestBuilder` interface that can be used to create the `MockHttpServletRequest` used in your test. Spring Security provides a few `RequestBuilder` implementations that can be used to make testing easier. In order to use Spring Security's `RequestBuilder` implementations ensure the following static import is used:

```
  import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.*;
```

Testing Form Based Authentication

You can easily create a request to test a form based authentication using Spring Security's testing support. For example, the following will submit a POST to "/login" with the username "user", the password "password", and a valid CSRF token:

```
  mvc
 .perform(formLogin())
```

It is easy to customize the request. For example, the following will submit a POST to "/auth" with the username "admin", the password "pass", and a valid CSRF token:

```
  mvc
 .perform(formLogin("/auth").user("admin").password("pass"))
```

We can also customize the parameters names that the username and password are included on. For example, this is the above request modified to include the username on the HTTP parameter "u" and the password on the HTTP parameter "p".

```
  mvc
 .perform(formLogin("/auth").user("u","admin").password("p","pass"))
```

Testing Logout

While fairly trivial using standard Spring MVC Test, you can use Spring Security's testing support to make testing log out easier. For example, the following will submit a POST to "/logout" with a valid CSRF token:

```
  mvc
 .perform(logout())
```

You can also customize the URL to post to. For example, the snippet below will submit a POST to "/signout" with a valid CSRF token:

```
mvc
.perform(logout("/signout"))
```

SecurityMockMvcResultMatchers

At times it is desirable to make various security related assertions about a request. To accommodate this need, Spring Security Test support implements Spring MVC Test's `ResultMatcher` interface. In order to use Spring Security's `ResultMatcher` implementations ensure the following static import is used:

```
import static org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatchers.*;
```

Unauthenticated Assertion

At times it may be valuable to assert that there is no authenticated user associated with the result of a `MockMvc` invocation. For example, you might want to test submitting an invalid username and password and verify that no user is authenticated. You can easily do this with Spring Security's testing support using something like the following:

```
mvc
.perform(formLogin().password("invalid"))
.andExpect(unauthenticated());
```

Authenticated Assertion

It is often times that we must assert that an authenticated user exists. For example, we may want to verify that we authenticated successfully. We could verify that a form based login was successful with the following snippet of code:

```
mvc
.perform(formLogin())
.andExpect(authenticated());
```

If we wanted to assert the roles of the user, we could refine our previous code as shown below:

```
mvc
.perform(formLogin().user("admin"))
.andExpect(authenticated().withRoles("USER","ADMIN"));
```

Alternatively, we could verify the username:

```
mvc
.perform(formLogin().user("admin"))
.andExpect(authenticated().withUsername("admin"));
```

We can also combine the assertions:

```
mvc
.perform(formLogin().user("admin").roles("USER","ADMIN"))
.andExpect(authenticated().withUsername("admin"));
```

Web应用程序的安全性

大多数Spring Security的用户将使用在这使得HTTP和Servlet API的用户应用程序的框架。在这一部分中,我们将看看Spring Security提供身份验证和访问控制特性的应用程序的web层。我们会看看该命名空间的门面后面,看看哪些类和接口实际上是组装提供web层安全。在某些情况下需要使用传统的bean配置提供完全控制配置,所以我们也将看到怎样的情况下直接配置这些类的命名空间。

The Security Filter Chain

Spring Security的web基础设施是完全基于标准的servlet过滤器。它不使用servlet或任何其他基于servlet框架(比如Spring MVC)在内部,所以它与任何特定的web技术没有紧密的联系。它涉及到`HttpServletRequest`和`HttpServletResponse`并不在乎请求是否来自一个浏览器,一个web服务客户端,一个`HttpInvoker`或一个AJAX应用程序。

Spring Security维护过滤器链内部,每个过滤器都有一个特定的责任和过滤器中添加或删除的配置取决于哪些服务是必需的。过滤器的顺序很重要,因为它们之间有依赖关系。如果你一直使用命名空间配置,那么过滤器会为你自动配置和你不需要明确定义任何Spring bean但可能有时你想要完全控制安全过滤器链,因为您正在使用的功能中不支持在命名空间中,或你使用你自己的自定义版本的类。

DelegatingFilterProxy

当使用Servlet过滤器时，你显然需要声明他们的`web.xml`，否则将被servlet容器忽略。在Spring Security，过滤器类也都在Spring bean中定义应用上下文，从而能够利用Spring的丰富的依赖注入的设施和生命周期接口的优势。Spring的`DelegatingFilterProxy`提供`web.xml`和应用程序上下文之间的链接

当使用`DelegatingFilterProxy`，你会看到这样的事情在`web.xml`文件中:

```
  <filter>
<filter-name>myFilter</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>myFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

请注意，该过滤器实际上是一个`DelegatingFilterProxy`，而不是类，会实现过滤器的逻辑。什么`DelegatingFilterProxy`所做的是通过向其从Spring应用程序上下文中的bean委托`Filter`的方法。这使得bean来从Spring Web应用程序上下文的生命周期支持和配置灵活性中受益。bean必须实现`javax.servlet.Filter`并且必须具有相同的名称，在`filter-name`元素。阅读`DelegatingFilterProxy`的Javadoc的更多信息

FilterChainProxy

Spring Security的网络基础设施，只能通过委托给`FilterChainProxy`的一个实例使用。安全过滤器不应该由自己来使用。理论上你可以声明每个弹簧安全过滤器bean，你需要在你的应用程序的上下文文件并添加相应的`DelegatingFilterProxy`到`web.xml`每个过滤器中，确保他们正确排序，但是这是很麻烦的并且如果你有很多的过滤器会很快弄乱了`web.xml`文件。`FilterChainProxy`让我们将一个条目添加到`web.xml`和完全处理应用程序上下文文件来管理我们的网络安全bean。它是使用`DelegatingFilterProxy`连接的，就像在上面的例子，随着`filter-name`设为"filterChainProxy"bean的名称。然后过滤器链中声明应用程序上下文相同的bean的名称。这里有一个例子:

```
  <bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">
<constructor-arg>
 <list>
 <sec:filter-chain pattern="/restful/**" filters="
  securityContextPersistenceFilterWithASCFalse,
  basicAuthenticationFilter,
  exceptionTranslationFilter,
  filterSecurityInterceptor" />
 <sec:filter-chain pattern="/**" filters="
  securityContextPersistenceFilterWithASCTrue,
  formLoginFilter,
  exceptionTranslationFilter,
  filterSecurityInterceptor" />
 </list>
</constructor-arg>
</bean>
```

命名空间元素`filter-chain`是用来方便设置应用程序中所需的安全过滤器链。[2]它映射一个特定的URL模式的从`filters`元素指定的bean的名字建立了过滤器列表，并结合`SecurityFilterChain`类型的bean。该`pattern`属性需要一个Ant路径和最具体的URI应该首先出现。[3]在运行时，`FilterChainProxy`将定位相匹配的当前Web请求，并由`filters`属性指定的过滤器bean列表将被应用到该请求的第一个URI模式。过滤器会按照它们定义的顺序调用，所以你必须在这是适用于一个特定的URL过滤器链的完全控制。

您可能已经注意到，我们已宣布两个`SecurityContextPersistenceFilter`在过滤器链（`ASC`的简称`allowSessionCreation`，`SecurityContextPersistenceFilter`的属性）。因为web服务从来不会在请求`jsessionid`，创造`HttpSession`表示这样的用户代理是一种浪费。如果你这需要这最大的扩展能力高容量的应用程序，我们建议您使用上面的方法。对于较小的应用程序，使用`SecurityContextPersistenceFilter`（其默认`allowSessionCreation`为`true`）就足够了。

需要注意的是`FilterChainProxy`不会调用它配置了过滤器标准过滤器生命周期方法。我们建议你使用Spring的应用程序上下文的生命周期接口作为替代，就像你对任何其他的Spring bean一样。

当我们看到如何使用命名空间配置建立网络安全,我们使用了`DelegatingFilterProxy`和"springSecurityFilterChain"这个名字。您现在应该能够看到，这是由`FilterChainProxy`的名称创建的命名空间。

绕过过滤器链

你可以使用属性`filters = "none"`替代供应一个过滤器bean列表。这将完全忽略安全筛选器链中的请求模式。请注意，任何匹配此路径将没有任何身份验证或授权服务应用，将可自由访问。如果你想在一个请求中使用的`SecurityContext`内容的内容，那么它必须通过保安过滤器链。否则，`SecurityContextHolder`不会被填充且内容也将空。

Filter Ordering

过滤器链中定义的顺序是非常重要的。不论你使用的是哪一种过滤器，顺序应该如下:

- `ChannelProcessingFilter`,因为它可能需要重定向到其他协议。
- `SecurityContextPersistenceFilter`,所以`SecurityContext`可在`SecurityContextHolder`在web请求的开始设立，并且当web请求结束时`SecurityContext`任何改变可以被复制到`HttpSession`（准备下一个使用Web请求）
- `ConcurrentSessionFilter`,因为它使用了`SecurityContextHolder`功能和需要更新`SessionRegistry`以反映反映主要正在处理的请求。
- 认证处理机制 - `UsernamePasswordAuthenticationFilter`，`CasAuthenticationFilter`，`BasicAuthenticationFilter`等 - 使得`SecurityContextHolder`可以被修饰以包含有效的`Authentication`请求令牌
- `SecurityContextHolderAwareRequestFilter`，如果你使用它来安装一个Spring Security意识`HttpServletRequestWrapper`到你的servlet容器
- `JaasApiIntegrationFilter`，如果`JaasAuthenticationToken`是在`SecurityContextHolder`这将处理`F`RememberMeAuthenticationFilter, so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`, and the request presents a cookie that enables remember-me services to take place, a suitable remembered `Authentication` object will be put thereilterChain`作为`Subject`在`JaasAuthenticationToken`
- `RememberMeAuthenticationFilter`,所以如果早期的认证处理机制没有更新`SecurityContextHolder`并且请求给出一个Cookie使remember-me服务发生，一个合适的记忆`Authentication`对象将被放在那里
- `AnonymousAuthenticationFilter`,这样如果之前的验证执行机制没有更新`SecurityContextHolder`，一个匿名`Authentication`对象将被放在那里
- `ExceptionTranslationFilter`,捕获任何Spring安全异常,以便响应可以返回一个HTTP错误或适当的`AuthenticationEntryPoint`可以启动
- `FilterSecurityInterceptor`,保护网络的URI，当访问被拒绝引发异常

Request Matching and HttpFirewall

Spring Security有几个方面，你已经定义模式对传入的请求，以决定该请求应如何处理测试。这发生在`FilterChainProxy`决定的请求应通过传递该过滤器链，也当`FilterSecurityInterceptor`决定哪些安全约束适用于请求。要了解什么机制和针对你定义的模式进行测试时使用什么URL值是非常重要的。

Servlet规范定义了`HttpServletRequest`其中有几个属性是通过getter方法访问，以及我们可能要匹配。这些都是`contextPath`，`servletPath`，`pathInfo`和`queryString`。 Spring Security唯一感兴趣的仅仅是在确保应用程序中的路径，所以`contextPath`被忽略。不幸的是，servlet规范并没有定义到底什么是`servletPath`和`pathInfo`的值将包含特定请求URI。例如，一个URL的每个路径段可包含参数, 定义在 RFC 2396 [4]. 该规范并没有明确说明是否这些都应该被包含在`servletPath和`pathInfo`的价值观和行为的不同servlet容器之间变化。有一种危险，当一个应用程序被部署在一个容器中，而不是从这些值中的路径参数，攻击者可以将它们添加到请求的URL,以意外导致模式匹配成功或失败。 [5]. 在传入的URL的其他变化也可能。它可能包含路径遍历序列（如`/../`）或多个斜杠（`//`），这也可能导致匹配失败。一些容器标准化这些之前执行servlet映射,但其他人没有。为了防止类似这些问题，`FilterChainProxy`采用了`HttpFirewall`战略，检查和包装的要求。未规范化请求默认自动拒绝和路径参数和复制斜线匹配的目的被删除。 [6]. 因此，至关重要的一个`FilterChainProxy`用于管理安全过滤器链。需要注意的是`servletPath`和`pathInfo`值由容器解码，所以你的应用程序不应该有任何有效的路径包含分号，这些部分将被删除匹配的目的。

正如上面提到的，默认的策略是使用Ant风格的路径进行匹配，这可能是大多数用户的最佳选择。该策略在使用Spring的`AntPathMatcher`执行对级联`servletPath`和`pathInfo`的格局不区分大小写的匹配类`AntPathRequestMatcher`实现，忽视了`queryString`。

如果由于某种原因，你需要一个更强大的匹配策略，您可以使用正则表达式。然后`RegexRequestMatcher`战略的实现。看到这个类的Javadoc获取更多信息。

在实践中我们建议你使用方法安全服务层,控制您的应用程序访问,并不完全依赖于在Web应用程序级别定义的安全约束的使用。URL改变,很难考虑所有可能的应用程序的URL可能支持和如何处理请求。你应该尝试限制自己使用一些简单的容易理解的路径。总是试图用"deny-by-default"的方法,全方位通配符(`/` **or**) 定义和拒绝访问。

安全在服务层定义更健壮且难以绕过,所以你应该利用Spring Security的方法安全选项。

与其它过滤器-基于框架使用

如果您使用的过滤器也基于其他一些框架，那么你需要确保Spring Security过滤器是第一位的。这使

得 SecurityContextHolder 到时由其他过滤器被填充以供使用。例子是使用SiteMesh的来装饰您的网页或类似Wicket的Web框架，它使用一个过滤器来处理其请求。

高级的命名空间配置

正如我们在命名空间章节前面所看到的，它可以使用多个 http 元素来定义不同的URL模式来进行不同的安全配置。每个元素创建内部 FilterChainProxy 内的过滤器链和应该映射的URL模式。这些元素将声明它们的添加顺序，所以最具体的模式必须再次先声明。这里的另一个例子，与上面有一个类似的情况，在应用程序支持一个无国籍的RESTful API和一个正常的网络应用，用户可以登录使用的一种形式。

```
  <!-- Stateless RESTful service using Basic authentication -->
<http pattern="/restful/**" create-session="stateless">
<intercept-url pattern='/**' access="hasRole('REMOTE')" />
<http-basic />
</http>

<!-- Empty filter chain for the login page -->
<http pattern="/login.htm*" security="none"/>

<!-- Additional filter chain for normal users, matching all other requests -->
<http>
<intercept-url pattern='/**' access="hasRole('USER')" />
<form-login login-page='/login.htm' default-target-url="/home.htm"/>
<logout />
</http>
```

核心安全过滤器

总是会有一些关键的过滤器在web应用程序中使用Spring Security,所以我们来看看这些和他们的支持类和接口。我们不会覆盖所有功能，所以如果你想获得完整的信息一定要看看它们的Javadoc。

FilterSecurityInterceptor

我们已经看到了 FilterSecurityInterceptor 简要讨论访问控制 ，我们已经与其中 <intercept-url> 元素相结合，在内部配置的命名空间中使用它。现在，我们将看到如何明确将其 FilterChainProxy 其配置为使用，其配套过滤器 ExceptionTranslationFilter 一起。典型的配置示例如下所示:

```
  <bean id="filterSecurityInterceptor"
 class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="securityMetadataSource">
 <security:filter-security-metadata-source>
 <security:intercept-url pattern="/secure/super/**" access="ROLE_WE_DONT_HAVE"/>
 <security:intercept-url pattern="/secure/**" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
 </security:filter-security-metadata-source>
</property>
</bean>
```

FilterSecurityInterceptor 负责处理HTTP资源的安全性。它需要一个 AuthenticationManager 和 AccessDecisionManager 参考。它也有提供适用于不同的HTTP URL请求的配置属性。回头参考技术介绍中这些原始的讨论。

在 FilterSecurityInterceptor 配置属性可以两种方式进行配置。上面显示的第一个，使用 <filter-security-metadata-source> 命名空间元素。这类似于 <http> 元素的命名空间，但 <intercept-url> 的子元素只使用 pattern 和 access 的属性。使用逗号分隔不同的配置属性,适用于每个HTTP URL。第二个选择是编写自己的‘SecurityMetadataSource’,但这超出了本文的范围。不管采用哪种方式，在 SecurityMetadataSource 负责返回一个 List<ConfigAttribute> 包含所有用一个安全的HTTP URL相关联的配置属性。

应当指出的是， FilterSecurityInterceptor.setSecurityMetadataSource（） 方法实际上是 FilterInvocationSecurityMetadataSource 的一个实例。这是一个标记接口，它的子类是 SecurityMetadataSource 。它只是表示 SecurityMetadataSource 明白 FilterInvocation 。在简单的利益，我们将继续参考 FilterInvocationSecurityMetadataSource 作为 SecurityMetadataSource ，因为区别是无关紧要大多数用户。为了简单起见，我们将继续参考 FilterInvocationSecurityMetadataSource 作为 SecurityMetadataSource ，因为对于大多数用户区别是无关紧要。

通过命名空间的语法创建的 SecurityMetadataSource 获得通过匹配在配置的 pattern 属性请求的URL特定 FilterInvocation 配置属性。这表现在，它确实为命名空间配置相同的方式。默认的是处理所有表达式作为Apache Ant的路径和正则表达式还支持更复杂的情况。 request-matcher 属性用于指定模式的类型被使用。它是不可能的相同的定义内混合表达式语法。作为一个例子，使用正则表达式，而不是Ant paths的先前的配置将按如下表示:

```
 <bean id="filterInvocationInterceptor"
 class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="runAsManager" ref="runAsManager"/>
<property name="securityMetadataSource">
 <security:filter-security-metadata-source request-matcher="regex">
 <security:intercept-url pattern="\A/secure/super/.*\Z" access="ROLE_WE_DONT_HAVE"/>
 <security:intercept-url pattern="\A/secure/.*\" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
 </security:filter-security-metadata-source>
</property>
</bean>
```

模式总是按照它们被定义的顺序进行。因此，重要的是，更具体的模式被定义在列表中比不太具体的模式更高。这是反映在我们的例子上面，在更具体的 `/secure/super/` 模式似乎比 `/secure/` 模式更高。如果它们被逆转， `/secure/` 模式会一直匹配，并且 `/secure/super/` 模式将永远不会被评估。

ExceptionTranslationFilter

该 `ExceptionTranslationFilter` 在 `FilterSecurityInterceptor` 安全过滤器堆栈的上面。它没有做任何实际的安全执法本身，而是处理由安全拦截器抛出的异常，并提供合适的HTTP响应。

```
 <bean id="exceptionTranslationFilter"
class="org.springframework.security.web.access.ExceptionTranslationFilter">
<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
<property name="accessDeniedHandler" ref="accessDeniedHandler"/>
</bean>

<bean id="authenticationEntryPoint"
class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">
<property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"
 class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
<property name="errorPage" value="/accessDenied.htm"/>
</bean>
```

AuthenticationEntryPoint

如果用户请求一个安全HTTP资源这个 `AuthenticationEntryPoint` 将被调用，但他们不被认证。适当的 `AuthenticationException` 或 `AccessDeniedException` 将被安全拦截器进一步拆毁了调用堆栈，触发入口点的 `commence` 方法。这确实呈现给用户的适当的反应，使认证可以开始工作。我们这里使用的是 `LoginUrlAuthenticationEntryPoint` ，这将请求重定向到一个不同的URL（一般是一个登录页面）。使用将取决于你想要的认证机制的实际实现在应用程序中使用。

AccessDeniedHandler

如果用户已经通过身份验证，他们试图访问受保护的资源会发生什么?在正常使用情况下，这不应该发生，因为应用程序的工作流程应该仅限于用户访问的操作。例如，HTML链接到一个管理页面可能隐藏唉一个没有管理员角色的用户中。你不能依赖隐藏链接的安全,总是有一个可能性,用户只会直接输入URL,以绕过限制。或者，他们可能会修改一个RESTful URL来改变一些参数值。您的应用程序必须对这些方案进行保护，或者它绝对会是不安全的。您通常会使用简单的网络层安全性约束适用于基本的网址，并使用更具体的方法，基于安全上的服务层接口真正明确什么是允许的。

如果一个 `AccessDeniedException` 被抛出并且用户已经被认证，那么这意味着一个操作已经尝试了它们不具有足够的权限。在这种情况下， `ExceptionTranslationFilter` 将调用第二策略， `AccessDeniedHandler` 。默认情况下, `AccessDeniedHandlerImpl` 被使用，这只是发送一个403（禁止）响应于客户端。此外，还可以配置明确的实例（如在上面的例子），并设置一个错误页面的URL，它会请求转发 [7]. 这可以是一个简单的“拒绝访问”页上，如一个JSP，或者它可以是更复杂的处理程序，如一个MVC的控制器。当然，你可以自己实现接口，并使用自己的实现。

它也可以提供当你自定义你的应用程序时使用命名空间来配置 `AccessDeniedHandler` 。详见 命名空间附录 了解更多详情。

SavedRequest s and the RequestCache Interface

`ExceptionTranslationFilter` 另一个责任是调用 `AuthenticationEntryPoint` 之前保存当前的请求。这允许使用已经验证后要恢复的请求(看到之前的概述 web 认证).一个典型示例是在用户登录表单,然后重定向到默认的原始URL `SavedRequestAwareAuthenticationSuccessHandler` (见 下方).

该 `RequestCache` 封装用于存储和检索 `HttpServletRequest` 实例所需的功能。默认 `HttpSessionRequestCache` 被使用，其中该请求存储在 `HttpSession` 。该 `RequestCacheFilter` 具有实际恢复从当用户被重定向到原始URL缓存中保存的请求工作。

在正常情况下，你不需要修改任何此功能，但保存的请求处理是一个“尽力而为”的做法，有可能是其默认的配置是无法处理的情况。使用这些接口，使得它完全可插入从Spring Security 3.0起。

SecurityContextPersistenceFilter

我们涵盖所有重要的过滤器在技术概述这一章，所以你可能想在这一点上重新阅读该部分。让我们先来看看，你会如何配置它与一个 `FilterChainProxy` 使用。一个基本的配置只需要bean本身。

```
  <bean id="securityContextPersistenceFilter"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter"/>
```

正如我们之前看到的,这个过滤器有两个主要的任务。它负责存储` securitycontext 内容之间的HTTP请求和当请求完成结算 securitycontextholder`。清除 `ThreadLocal` 其中存储的上下文是必不可少的，因为它可能会以其他方式可能是一个线程被替换成servlet容器的线程池，仍连接一个特定的用户的安全上下文。这个线程可能在稍后的阶段中使用，与错误的凭证执行操作。

SecurityContextRepository

从Spring Security 3.0起,加载和存储安全上下文的工作现在是委托给一个单独的策略接口:

```
  public interface SecurityContextRepository {

SecurityContext loadContext(HttpRequestResponseHolder requestResponseHolder);

void saveContext(SecurityContext context, HttpServletRequest request,
  HttpServletResponse response);
}
```

该 `HttpRequestResponseHolder` 只是一个容器传入的请求和响应对象，允许实现替换这些包装类。返回的内容将被传递给过滤器链。

默认实现 `HttpSessionSecurityContextRepository` ,存储安全上下文作为HttpSession的属性 [8]. 对于此实现最重要的配置参数是 `allowSessionCreation` 属性，它默认为 `true` ,从而使类来创建一个会话如果它需要一个存储身份验证的用户的安全上下文(它不会创建一个除非认证发生和安全上下文的内容改变了)。如果你不希望创建一个会话,则可以将此属性设置为 `false` :

```
  <bean id="securityContextPersistenceFilter"
 class="org.springframework.security.web.context.SecurityContextPersistenceFilter">
<property name='securityContextRepository'>
 <bean class='org.springframework.security.web.context.HttpSessionSecurityContextRepository'>
 <property name='allowSessionCreation' value='false' />
 </bean>
</property>
</bean>
```

或者你可以提供 `NullSecurityContextRepository` 的一个实例, 一个 null object 的实现, 这将防止安全上下文被存储，即使已经在会话期间创建一个该实例请求。

UsernamePasswordAuthenticationFilter

我们现在看到的三个主要的过滤器总是存在在一个Spring Security web配置。这些也都是这是由命名空间 `<http>` 元素自动创建，并且不能被替代被取代的三种。现在唯一缺少的是一个实际的认证机制，而这将允许用户进行身份验证。该过滤器是最常用的身份验证过滤器，也是最常用的一种认证过滤器 [9]. 另外，它还提供了 `<form-login>` 元素的命名空间的实现。有三个阶段需要进行配置。

- 配置一个 `LoginUrlAuthenticationEntryPoint` 登录页面的URL,就像我们上面,并设置 `ExceptionTranslationFilter` 。
- 实现登录页面(使用JSP或MVC控制器)。
- 配置一个 `UsernamePasswordAuthenticationFilter` 应用程序上下文的实例
- 将过滤器bean添加到您的过滤器链代理（确保你要注意顺序）。

登录表单包含 `username` 和 `password` 输入字段，并发布到由过滤器（默认情况下这是 `/login` ）监测的URL。基本的过滤器配置看起来是这样的：

```
  <bean id="authenticationFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>
```

认证成功和失败的应用程序流程

该过滤器调用配置 `AuthenticationManager` 处理每个认证请求。继成功认证或认证失败的目的地是由 `AuthenticationSuccessHandler` 和 `AuthenticationFailureHandler` 策略接口分别控制。过滤器属性允许你设置这些，所以你可以完全自定义行为 [10]. 一些标准的实现，如 `SimpleUrlAuthenticationSuccessHandler` ， `SavedRequestAwareAuthenticationSuccessHandler` ， `SimpleUrlAuthentica` 供。看一看这些类的Javadoc和 `AbstractAuthenticationProcessingFilter` 来获取它们是如何工作的概述和所支持的特性。

如果认证成功，将所得 `Authentication` 对象将被放置到 `SecurityContextHolder` 。配置的 `AuthenticationSuccessHandler` 将被重定向或用户转发给适当的目的地。默认情况下使用 `SavedRequestAwareAuthenticationSuccessHandler` ,这意味着用户将被重定向到他们要求的原始目的地之前，他们被要

求登录。

`ExceptionTranslationFilter`缓存原始请求用户。当用户进行身份验证时,这种缓存的请求处理程序利用请求获取原始URL和重定向到它。然后原始请求被重建和使用作为替代。

如果身份验证失败,则将调用`AuthenticationFailureHandler`配置。

Servlet API的集成

本节描述如何与Servlet API集成Spring Security。 servletapi-xml示例应用程序演示了使用这些方法。

Servlet 2.5+ 集成

HttpServletRequest.getRemoteUser()

HttpServletRequest.getRemoteUser() 将返回`SecurityContextHolder.getContext().getAuthentication().getName()`的结果。 这通常是当前的用户名。这可能是有用的,如果你想在应用程序中显示当前用户名。此外,检查如果这是空可以用来表示如果一个用户已经通过身份验证或者是匿名的。知道用户是否通过身份验证可以用于确定特定UI元素应该显示（即注销链接应该只显示如果用户身份验证）。

HttpServletRequest.getUserPrincipal()

HttpServletRequest.getUserPrincipal()将返回`SecurityContextHolder.getContext().getAuthentication()`的结果。这意味着它是一个`Authentication`通常是`UsernamePasswordAuthenticationToken`的一个实例使用基于用户名和密码的身份验证。这可能是有用的,如果你需要更多关于用户的信息。例如,您可能已经创建了一个自定义`UserDetailsService`,返回一个自定义的`UserDetails`包含您的用户的姓名。你可以用以下获得这些信息:

```
  Authentication auth = httpServletRequest.getUserPrincipal();
// assume integrated custom UserDetails called MyCustomUserDetails
// by default, typically instance of UserDetails
MyCustomUserDetails userDetails = (MyCustomUserDetails) auth.getPrincipal();
String firstName = userDetails.getFirstName();
String lastName = userDetails.getLastName();
```

应该指出的是，它通常是坏的做法，在你的应用程序执行这么多的逻辑。相反,应该集中它减少任何Spring Security和Servlet API的耦合。

HttpServletRequest.isUserInRole(String)

The HttpServletRequest.isUserInRole(String) 将确定`SecurityContextHolder.getContext().getAuthentication().getAuthorities()`包含一个`GrantedAuthority`角色传入的`isUserInRole(String)`。 一般用户不应通过"ROLE_"前缀到这个方法,因为它是自动添加的。例如,如果你想要确定当前用户有权力"ROLE_ADMIN",你可以使用以下:

```
  boolean isAdmin = httpServletRequest.isUserInRole("ADMIN");
```

这可能是有用的确定应该显示特定的UI组件。例如,您可能只显示管理链接如果当前用户是管理员。

Servlet 3+ 集成

以下部分将介绍Servlet 3集成Spring Security的方法。

HttpServletRequest.authenticate(HttpServletRequest,HttpServletResponse)

The HttpServletRequest.authenticate(HttpServletRequest,HttpServletResponse)方法可用于确保用户身份验证。如果他们没有身份验证,配置AuthenticationEntryPoint将用于请求用户进行身份验证(即重定向到登录页面)。

HttpServletRequest.login(String,String)

The HttpServletRequest.login(String,String) 方法可用于验证用户与当前`AuthenticationManager`。例如,下面将尝试验证用户名 "user"和密码"password":

```
  try {
httpServletRequest.login("user","password");
} catch(ServletException e) {
// fail to authenticate
}
```

没有必要捕获ServletException如果你想让Spring Security来处理失败的身份验证。

HttpServletRequest.logout()

The HttpServletRequest.logout() 方法可以用于记录当前用户。

通常这意味着SecurityContextHolder将被清除,HttpSession将失效,任何“记住我”身份验证将清理干净,等。然而,LogoutHandler配置实现将取决于你的Spring Security配置。需要注意是,在HttpServletRequest.logout()被调用,你要还负责编写一个响应。通常这将涉及一个重定向到欢迎页面。

AsyncContext.start(Runnable)

The AsynchContext.start(Runnable) 的方法,确保您的凭据将传播到新的线程。使用 Spring Security的并发支持Spring Security覆盖AsyncContext.start(可运行),以确保当前SecurityContext处理可运行时使用。例如,以下将输出当前用户的身份验证:

```
  final AsyncContext async = httpServletRequest.startAsync();
async.start(new Runnable() {
 public void run() {
  Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
  try {
   final HttpServletResponse asyncResponse = (HttpServletResponse) async.getResponse();
   asyncResponse.setStatus(HttpServletResponse.SC_OK);
   asyncResponse.getWriter().write(String.valueOf(authentication));
   async.complete();
  } catch(Exception e) {
   throw new RuntimeException(e);
  }
 }
});
```

Async Servlet Support

如果你使用的是基于Java的配置,你已经准备好了。如果你使用XML配置中,有一些是必要的更新。第一步是确保你更新你的web.xml使用至少3.0模式如下所示:

```
  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
version="3.0">

</web-app>
```

接下来你需要确保你的springSecurityFilterChain设置处理异步请求。

```
  <filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
 org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
<async-supported>true</async-supported>
</filter>
<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
<dispatcher>REQUEST</dispatcher>
<dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

就是这样!现在的Spring Security将确保您的SecurityContext是在异步请求传播了。

那么它是怎样工作的？如果你是不是真的感兴趣，请随时跳过这一节的其余部分，否则就继续阅读。这是内置在Servlet规范,但有一点调整,Spring Security确保事情妥善处理异步请求。Spring Security 3.2之前,securitycontextholder的securitycontext自动保存只要httpservletresponse承诺。这可能会导致在异步环境下的问题。例如,考虑以下:

```
  httpServletRequest.startAsync();
new Thread("AsyncThread") {
 @Override
 public void run() {
  try {
   // Do work
   TimeUnit.SECONDS.sleep(1);

   // Write to and commit the httpServletResponse
   httpServletResponse.getOutputStream().flush();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}.start();
```

问题是,这个线程是未知的Spring Security,所以SecurityContext中不会传播到它。这意味着当我们提交HttpServletResponse没有SecuriytContext。当Spring Security的自动保存SecurityContext支持HttpServletResponse

就会失去我们的登录用户。

自从3.2版本开始，Spring Security是足够聪明，不再自动保存SecurityContext，一旦遵守协议 HttpServletResponse的HttpServletRequest.startAsync（）将尽快被调用。

Servlet 3.1+ 集成

以下部分描述了Servlet3.1与Spring Security集成的方法。

HttpServletRequest#changeSessionId()

The HttpServletRequest.changeSessionId() 是默认的方法防止 固定会话 损害Servlet 3.1或更高的版本。

Basic和Digest认证

Basic和digest认证是在web应用中流行的替代身份验证机制。基本身份验证通常是使用无状态的客户对每个请求通过他们的凭证。这是相当常见的结合，其中一个应用程序通过两个基于浏览器的用户界面，并作为Web服务使用基于表单的身份验证中使用它。但是，基本身份验证发送密码以纯文本，所以应该只在真正加密传输层可以使用，例如HTTPS。

BasicAuthenticationFilter

`BasicAuthenticationFilter`负责处理HTTP头部中的基本认证证书。这可以用于验证由Spring远程协议(如Hessian和Burlap)的呼叫以及正常的浏览器的用户代理（如Firefox和Internet Explorer）。 HTTP基本认证的标准是由RFC 1945，第11条规定，以及`BasicAuthenticationFilter`符合这个RFC。基本身份验证是一个有吸引力的认证方法，因为它是非常广泛的，部署在用户代理和实施是非常简单的（它只是一个用户名的Base64编码:密码，在HTTP头中指定）。

Configuration

为了实现HTTP基本身份验证,您需要添加一个`BasicAuthenticationFilter`到你的过滤器链。应用程序上下文应包含`BasicAuthenticationFilter`及其所需的合作者:

```
  <bean id="basicAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
</bean>

<bean id="authenticationEntryPoint"
class="org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">
<property name="realmName" value="Name Of Your Realm"/>
</bean>
```

配置的`AuthenticationManager`处理每个认证请求。如果认证失败，配置`AuthenticationEntryPoint`将被用来重试认证过程。通常你会在组合使用过滤器`BasicAuthenticationEntryPoint` ，它会返回一个合适的头部重试HTTP基本身份验证的401响应。如果认证成功，将所得`Authentication`对象将和平常一样被放入`SecurityContextHolder`。

如果认证事件成功，或者认证不需要执行，因为HTTP头部没有包含支持的认证请求，过滤器链将继续正常工作。过滤器链中断的唯一情况是，如果认证失败了， `Auth`DigestAuthenticationFilter`能够消化处理HTTP头部中的认证证书。摘要式身份验证试图解决许多基本身份验证的弱点，特别是保证凭据从不通过线路明文形式发送。许多用户支持摘要式身份验证，包括Firefox和Internet Explorer。标准管理HTTP摘要认证由RFC 2617定义,更新摘要式身份验证的一个早期版本由RFC 2069标准规定。大多数用户代理实现RFC 2617。Spring Security的`DigestAuthenticationFilter`兼容“auth”质量认证的保护(`qop`)规定的RFC 2617,这也与RFC 2069提供向后兼容性。摘要式身份验证是一个更有吸引力的选择,如果您需要使用未加密的HTTP(即没有TLS / HTTPS)并希望最大化安全的身份验证过程。事实上WebDAV协议摘要式身份验证是一个强制性的要求,指出由RFC 2518 17.1节。 enticationEntryPoint`被调用。

DigestAuthenticationFilter

`DigestAuthenticationFilter`能够消化处理HTTP头部中的认证证书。摘要式身份验证试图解决许多基本身份验证的弱点，特别是保证凭据从不通过线路明文形式发送。许多用户支持摘要式身份验证，包括Firefox和Internet Explorer。标准管理HTTP摘要认证由RFC 2617定义,更新摘要式身份验证的一个早期版本由RFC 2069标准规定。大多数用户代理实现RFC 2617。Spring Security的`DigestAuthenticationFilter`兼容“auth”质量认证的保护(`qop`)规定的RFC 2617,这也与RFC 2069提供向后兼容性。摘要式身份验证是一个更有吸引力的选择,如果您需要使用未加密的HTTP(即没有TLS / HTTPS)并希望最大化安全的身份验证过程。事实上WebDAV协议摘要式身份验证是一个强制性的要求,指出由RFC 2518 17.1节可见。

> 你不应该在现代应用中使用摘要，因为它被认为是不安全的。 最明显的问题是，你必须把您的密码以明文，加密，或一个MD5格式。 所有这些存储格式被认为是不安全的。 相反,你应该使用一种自适应的哈希密码（即BCrypt，PBKDF2，SCrypt等）。

摘要式身份验证的核心是一个"nonce"。这是一个服务器生成的值。Spring Security nonce采用以下格式:

```
  base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))
expirationTime:   The date and time when the nonce expires, expressed in milliseconds
key:              A private key to prevent modification of the nonce token
```

`DigestAuthenticatonEntryPoint`有一个属性指定用于生成nonce令牌的`key`,以及`nonceValiditySeconds`属性确定过期时间(默认300,等于5分钟)。只要nonce是有效的，摘要就会通过串联字符串包括用户名，密码，随机数，被请求的URI，一个客户端产生的随机数（仅该用户代理每个请求生成一个随机值），域名等，然后执行MD5哈希值。如果他们不同意一个包含值(如密码),服务器和用户代理执行此计算消化,就会导致不同的哈希码。在Spring Security实现中，如果服务器生成的nonce已经过期（但是摘要还是有效），在`DigestAuthenticationEntryPoint`将发送一个`"stale=true"`头。这告诉用户代理没有必要打扰用户（如用户名和密码等是正确的），而只是采用了全新的随机数再试一次。

对于`DigestAuthenticationEntryPoint`的参数`nonceValiditySeconds`而言一个适当的值取决于您的应用程序。非常安全的应用程序应该注意一个拦截认证头可以用来模拟本体，直到`expirationTime`中包含的特定场合。这是关键的原则，选择适当的设置，但它是不寻常的，非常安全的应用程序不能运行在第一个实例的TLS/HTTPS。

因为摘要式身份验证的更复杂的实现，经常有用户代理的问题。例如，IE不能在同一个会话的请求提出了一个“opaque”令牌。因此Spring Security封装了所有状态信息进入到“nonce”令牌中。在我们的测试中,Spring Security的实现与FireFox和ie能可靠地工作,正确处理nonce超时等。

Configuration

既然我们已经回顾了理论,让我们看看如何使用它。为了实现HTTP摘要认证，必须在过滤器链定义`DigestAuthenticationFilter`。这个应用上下文将需要定义`DigestAuthenticationFilter`及其所需的合作者：

```
  <bean id="digestFilter" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
<property name="userDetailsService" ref="jdbcDaoImpl"/>
<property name="authenticationEntryPoint" ref="digestEntryPoint"/>
<property name="userCache" ref="userCache"/>
</bean>

<bean id="digestEntryPoint" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">
<property name="realmName" value="Contacts Realm via Digest Authentication"/>
<property name="key" value="acegi"/>
<property name="nonceValiditySeconds" value="10"/>
</bean>
```

配置`UserDetailsService`是必要的因为`DigestAuthenticationFilter`必须有一个用户的明文密码直接访问。如果您正在使用您的DAO中的编码密码，摘要身份验证将无法工作 [11]. 这个DAO的协作者，与`UserCache`一起，通常直接与`DaoAuthenticationProvider`共享。该`authenticationEntryPoint`属性必须是`DigestAuthenticationEntryPoint`，使`DigestAuthenticationFilter`能够获得正确的`realmName`和`key`再进行摘要计算。

像`BasicAuthenticationFilter`，如果认证成功，`Authentication`请求令牌将被放置到了`SecurityContextHolder`。如果认证事件成功，或者认证不需要执行，因为HTTP头部没有包含摘要认证请求，过滤器链将继续正常。过滤器链中断的唯一情况是，如果认证失败了`AuthenticationEntryPoint`被调用，如前一段讨论。

摘要式身份验证的RFC提供了一系列附加功能，以进一步提高安全性。例如，随机数可以在每次请求被改变。尽管如此，Spring Security的实施旨在最大限度地减少执行（和用户代理出现不兼容）的复杂性，并避免保存服务器端的状态。你被邀请审查RFC 2617如果你想更详细地探索这些功能。据我们所知，Spring Security的实现中遵守了RFC的最低标准。

Remember-Me Authentication

Overview

Remember-me或persistent-login身份验证是指网站能够记住一个主体的身份之间的会话。这通常是通过发送cookie给浏览器，以及在未来的会话中发现的cookie，并进行自动登录发生完成的。Spring Security提供了这些操作发生的必要的挂钩，并有两个具体的remember-me实现。其中一个使用散列来保护基于cookie标记的安全性，另一个使用了数据库或其他持久化存储机制来保存生成的标记。

注意，所有实现都需要一个`UserDetailsService`。如果您正在使用身份验证提供者不使用`UserDetailsService`(例如,LDAP提供者),那么它不会工作,除非你也有一个`UserDetailsService`应用程序上下文中的bean。

Simple Hash-Based Token Approach

这种方法使用散列来完成remember-me策略。本质上一个cookie发送到浏览器交互验证成功后,使用的cookie组成结构如下：

```
  base64(username + ":" + expirationTime + ":" +
md5Hex(username + ":" + expirationTime + ":" password + ":" + key))

username:          As identifiable to the UserDetailsService
password:          That matches the one in the retrieved UserDetails
expirationTime:    The date and time when the remember-me token expires, expressed in milliseconds
key:               A private key to prevent modification of the remember-me token
```

因此，remember-me的令牌在指定的期间是有效的，提供的用户名，密码和密钥不会改变。值得注意的是，这有一个潜在的安全问题，任何用户代理在remember-me令牌到期之前捕获到它，令牌将是可用的。这与摘要式身份验证问题是同样的问题。如果本人知道令牌已被抓获，他们可以轻松地更改他们的密码，并立即注销所有的remember-me标记。如果需要更显著的安全性，你应该使用在下一节中描述的方法。另外记得remember-me服务根本不应该被使用。

如果你熟悉在namespace configuration命名空间配置这一章中讨论的主题，你可以启用remember-me认证只需添加`<remember-me>`元素:

```
  <http>
...
<remember-me key="myAppKey"/>
</http>
```

`UserDetailsService`一般会自动选择。如果你有一个以上的应用环境，你需要指定哪一个应该使用`user-service-ref`属性，其中的值在你的`UserDetailsService` bean的名称中设置。

Persistent Token Approach

这种方法是基于文章 http://jaspan.com/improved_persistent_login_cookie_best_practice 有一些小的修改 [12]. 若要使用命名空间配置的这种方法，你应该提供一个数据源引用:

```
  <http>
...
<remember-me data-source-ref="someDataSource"/>
</http>
```

数据库应该包含一个`persistent_logins`表,使用以下SQL创建(或同等的):

```
  create table persistent_logins (username varchar(64) not null,
        series varchar(64) primary key,
        token varchar(64) not null,
        last_used timestamp not null)
```

Remember-Me的接口和实现

Remember-me是用于`UsernamePasswordAuthenticationFilter`,并通过`AbstractAuthenticationProcessingFilter`超类的钩子实现。 它也被运用在`BasicAuthenticationFilter`. 钩子会在合适的时候调用一个具体`RememberMeServices`。 该接口看起来像这样:

```
  Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

void loginFail(HttpServletRequest request, HttpServletResponse response);

void loginSuccess(HttpServletRequest request, HttpServletResponse response,
 Authentication successfulAuthentication);
```

请参考JavaDocs获得什么方法都做了更充分的讨论，不过注意在这个阶段，`AbstractAuthenticationProcessingFilter`仅调用`loginFail()`和`loginSuccess()`方法。当`SecurityContextHolder`不包含`Authentication`时`autoLogin()`方法被`RememberMeAuthenticationFilter`调用。因此，此接口提供了基本的remember-me的实现与认证相关的事件的充分通知，和代表的执行情况时，一个候选Web请求可能包含一个cookie并希望被记住。这种设计允许任何数目的remember-me实现策略。我们已经看到上面Spring Security提供两种实现方法。我们来看看这些。

TokenBasedRememberMeServices

这个实现支持 Simple Hash-Based Token Approach中描述的简单方法。`TokenBasedRememberMeServices`生成一个`RememberMeAuthenticationToken`,由`RememberMeAuthenticationProvider`处理。一个`key`认证提供者和`TokenBasedRememberMeServices`之间共享。此外，`TokenBasedRememberMeServices`需要一个UserDetailsService可以实现检索比较用户名和密码的目的，并生成`RememberMeAuthenticationToken`包含正确`GrantedAuthority`的一个UserDetailsService。如果用户提供无效的cookie，应该应用一些注销命令。`TokenBasedRememberMeServices`还实现了Spring Security的` logouthandler 接口，所有可以用`LogoutFilter`自动清除cookie。

在应用程序环境方面，remember-me服务需要的bean类如下:

```
  <bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter">
<property name="rememberMeServices" ref="rememberMeServices"/>
```

```
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

<bean id="rememberMeServices" class=
"org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices">
<property name="userDetailsService" ref="myUserDetailsService"/>
<property name="key" value="springRocks"/>
</bean>

<bean id="rememberMeAuthenticationProvider" class=
"org.springframework.security.authentication.RememberMeAuthenticationProvider">
<property name="key" value="springRocks"/>
</bean>
```

不要忘记添加你的 `RememberMeServices` 实现 `UsernamePasswordAuthenticationFilter.setRememberMeServices()` 的属性,包括 `RememberMeAuthenticationProvider` 在 `AuthenticationManager.setProviders()` 中的列表，并添加 `RememberMeAuthenticationFilter` 到你的 `FilterChainProxy` (一般在你的 `UsernamePasswordAuthenticationFilter` 之后)。

PersistentTokenBasedRememberMeServices

这个类可以以同样的方式作为 `TokenBasedRememberMeServices` 被使用，但它另外需要与 `PersistentTokenRepository` 存储令牌进行配置。有两种标准实现。

- `InMemoryTokenRepositoryImpl` 这是只用来测试。
- `JdbcTokenRepositoryImpl` 其存储在数据库中的标记。

数据库模式如上面所描述 Persistent Token Approach.

Cross Site Request Forgery (CSRF)

本节讨论Spring Security’s Cross Site Request Forgery (CSRF) 支持。

CSRF攻击

在我们讨论Spring Security如何保护应用程序不受CSRF攻击之前，我们将解释什么是CSRF攻击。让我们来看一个具体的例子来更好地理解。

假设你的银行网站提供允许从当前登录的用户转移钱转到另一个银行账户的形式。例如，HTTP请求可能看起来像:

```
  POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

现在假装你身份验证你的银行网站，然后再不注销，请访问一个恶意的网站。恶意的网站包含以下形式的HTML页面:

```
  <form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
 name="amount"
 value="100.00"/>
<input type="hidden"
 name="routingNumber"
 value="evilsRoutingNumber"/>
<input type="hidden"
 name="account"
 value="evilsAccountNumber"/>
<input type="submit"
 value="Win Money!"/>
</form>
```

你想赢钱,所以你点击submit按钮。在这个过程中，你无意中转移$100到恶意用户。这是因为，虽然恶意的网站不能看到你的cookies，与银行相关的cookies仍然与请求一起发送。

最糟糕的是，这整个过程可能已经使用JavaScript自动化。这意味着你甚至没有需要点击的按钮。那么我们该如何保护自己免受此类攻击?

同步器标记模式

问题是，从银行网站上的HTTP请求，和从恶意的网站的要求是完全一样的。这就意味着没有办法拒绝来自恶意的网站的请求,允许请求来自银行的网站。为了抵御CSRF攻击我们需要确保请求中有一些恶意的网站是无法提供的。

一种解决方案是使用 同步器标记模式. 这个解决方案是保证每个请求除了需要我们的会话cookie，还用随机生成的令牌作为

HTTP参数。提交一个请求时,服务器必须查找参数和比较它的期望值和实际值的请求。如果值不匹配,请求失败。

我们可以轻松的预期，只要更新每个HTTP请求的令牌。这样做可以安全地自同源策略保证了恶意的网站无法读取响应。此外，我们不希望包括HTTP GET随机令牌，因为这可能会导致标记被泄露。

让我们来看看我们的例子将如何改变。假定随机产生的标记存在于一个HTTP参数name_csrf。例如，要转账的要求是这样的:

```
  POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

你会注意到，我们添加了_csrf参数的随机值。现在，恶意网站将无法猜测_csrf参数(必须提供明确的恶意网站)，当服务器将实际的令牌与预期的令牌进行比较时，传输将失败。

何时使用CSRF保护

什么时候应该使用CSRF保护？我们的建议是使用CSRF保护，可以通过浏览器处理普通用户的任何请求。如果你只是创建一个非浏览器客户端使用的服务,你可能会想要禁用CSRF保护。

CSRF保护和JSON

一个常见的问题是"我是否需要保护JSON请求的javascript ?"简短的答案是，这取决于你。但是，你必须非常小心，因为有CSRF攻击可以影响JSON请求。例如，一个恶意用户可以使用以下格式，创建一个 CSRF和JSON使用以下form:

```
  <form action="https://bank.example.com/transfer" method="post" enctype="text/plain">
<input name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber", "ignore_me":"'
value='test"}' type='hidden'>
<input type="submit"
 value="Win Money!"/>
</form>
```

这将产生以下JSON结构

```
  { "amount": 100,
"routingNumber": "evilsRoutingNumber",
"account": "evilsAccountNumber",
"ignore_me": "=test"
}
```

如果一个应用程序没有验证内容类型，那么它会接触到这种攻击。根据设置,验证内容类型的Spring MVC应用程序仍然可以利用更新URL后缀结尾".json"如下所示:

```
  <form action="https://bank.example.com/transfer.json" method="post" enctype="text/plain">
<input name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber", "ignore_me":"'
value='test"}' type='hidden'>
<input type="submit"
 value="Win Money!"/>
</form>
```

CSRF和无状态的浏览器应用程序

如果我的应用程序是无状态的呢?这并不意味着你是受保护的。事实上,如果用户对于一个给定的请求不需要在web浏览器中执行任何操作,他们可能仍然容易受到CSRF攻击。

例如,考虑一个应用程序使用一个定制的cookie,其中包含所有的声明进行身份验证,而不是JSESSIONID。当CSRF是由自定义cookie与在该JSESSIONID cookie在前面的例子中相同的方式发送的，请求被发送。

使用基本身份验证的用户也容易受到CSRF攻击，因为浏览器会自动包括以同样的方式，在我们前面的例子中该JSESSIONID的cookie会发送任何请求的用户名密码。

Using Spring Security CSRF Protection

那么，什么是必要的，使用Spring Security来保护我们的网站不受CSRF攻击的步骤是什么？使用Spring Security的CSRF保护的步骤如下所述:

- 使用适当的HTTP动词
- 配置CSRF保护
- 包括CSRF令牌

使用适当的HTTP动词

防止CSRF攻击的第一步是确保你的网站使用适当的HTTP动词。具体来说，前Spring Security的CSRF支持可以使用，你需要

确定你的应用程序使用PATCH，POST, PUT, and/or DELETE对于任何修改状态。

这不是一个Spring安全限制的支持，而是进行适当的预防CSRF的一般要求。原因是包括在一个HTTP GET私人信息可能导致的信息泄露。看到 RFC 2616 Section 15.1.3在URI的敏感信息编码 一般的指导对于敏感信息使用POST而不是GET。

配置CSRF保护

下一步是要包括你的应用程序中的Spring Security的CSRF保护。有些框架处理无效CSRF无效用户的会话令牌，而这将导致 其自身的问题。相反，在默认情况下Spring Security的CSRF保护将产生一个HTTP 403访问被拒绝。这可以通过配置AccessDeniedHandler以不同方式处理。

Spring Security 4.0,CSRF保护与XML配置默认启用。如果你想禁用CSRF保护,下面可以看到相应的XML配置。

```
 <http>
 <!-- ... -->
 <csrf disabled="true"/>
</http>
```

CSRF保护默认情况下使用Java配置启用。如果您想禁用CSRF，下面可以看到相应的Java配置。请参考Javadoc中的CSRF()在CSRF保护是如何自定义配置的。

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable();
}
}
```

包括CSRF令牌

Form Submissions

最后一步是确保您包括CSRF令牌在所有补丁，POST，PUT和DELETE方法。一个方法是使用`_csrf`请求属性来获取当前`CsrfToken`。用一个JSP这样做的一个示例如下所示:

```
 <c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
 method="post">
<input type="submit"
 value="Log out" />
<input type="hidden"
 name="${_csrf.parameterName}"
 value="${_csrf.token}"/>
</form>
```

一个更容易的方法是从Spring Security的JSP标签库使用CSRF输入标签 。

> 如果你正在使用Spring MVC的`<form:form>`标签或 Thymeleaf 2.1+ 和使用 `@EnableWebSecurity`，`CsrfToken` 是自动包括 (使用 `CsrfRequestDataValueProcessor`).

Ajax和JSON请求

如果使用JSON,那么它是不可能在HTTP参数提交CSRF令牌。相反,你可以提交HTTP报头中的令牌。一个典型的模式是将包括您的meta标签内的CSRF令牌。用一个JSP一个例子如下所示:

```
 <html>
<head>
 <meta name="_csrf" content="${_csrf.token}"/>
 <!-- default header name is X-CSRF-TOKEN -->
 <meta name="_csrf_header" content="${_csrf.headerName}"/>
 <!-- ... -->
</head>
<!-- ... -->
```

而不是手动创建meta标签,您可以从Spring Security JSP标记库中使用简单的csrfMetaTags标签。

然后，您可以包含所有Ajax请求中的令牌。如果你是使用jQuery,这将通过以下:

```
 $(function () {
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
 xhr.setRequestHeader(header, token);
});
});
```

作为一种替代的jQuery，我们建议使用 cujoJS's rest.js. The rest.js模块提供先进的支持以RESTful方式处理HTTP请求和响应。核心能力是能够说明链接所需的HTTP客户端添加行为拦截器客户端。

```
 var client = rest.chain(csrf, {
token: $("meta[name='_csrf']").attr("content"),
name: $("meta[name='_csrf_header']").attr("content")
});
```

所配置的客户端可以与需要一个到CSRF保护资源的请求的应用程序共享任何组件。rest.js和jQuery之间有一个显著的不同是只有配置客户端的请求将包含CSRF令牌,vs jQuery将包括*所有*请求令牌。请求接收令牌能力的范围有助于防止泄露CSRF令牌给第三方。请参阅 rest.js 参考文档获取更多关于rest.js的信息。

CookieCsrfTokenRepository

可能有些情况下，用户想要坚持 `CSRF Token` 在cookie中。 默认情况下 `CookieCsrfTokenRepository` 将编写一个名为 `XSRF-TOKEN` 的cookie和从头部命名 `X-XSRF-TOKEN` 中读取或HTTP参数 `_csrf` 。 这些默认值来自 AngularJS

你可以使用下面的XML配置 `CookieCsrfTokenRepository` :

```
 <http>
<!-- ... -->
<csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
p:cookieHttpOnly="false"/>
```

示例显式地设置 `cookieHttpOnly=false` . 这是必要的，以允许JavaScript（即AngularJS）读取它。 如果你不需要使用JavaScript直接读取cookie的能力， ，建议省略 `cookieHttpOnly=false` 来提高安全性。

你可以配置 `CookieCsrfTokenRepository` 在Java配置中使用:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
  WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
  .csrf()
   .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
}
}
```

示例显式地设置 `cookieHttpOnly=false` . 这是必要的,允许JavaScript(例如AngularJS)读取它。 如果你不需要使用JavaScript直接读取cookie的能力，建议省略 `cookieHttpOnly=false` (通过使用 `new CookieCsrfTokenRepository()` 代替) 提高安全性.

CSRF警告

在实现CSRF时有几个注意事项.

Timeouts

一个问题是,预期CSRF令牌存储在HttpSession中,所以一旦HttpSession到期你的配置将收到一个 InvalidCsrfTokenException `AccessDeniedHandler` 。如果您使用的是默认的 `AccessDeniedHandler` ,浏览器将会得到一个HTTP 403并显示错误消息。

有人可能会问，为什么预期 `CsrfToken` 没有存储在默认的cookie。这是因为在该首部（即，指定的Cookie）可以被另一个域设置公知的漏洞。这是一样的道理Ruby on Rails no longer skips CSRF checks when the header X-Requested-With is present. 见 这个webappsec.org线程 关于如何执行的细节漏洞。另一个缺点是，通过除去状态（即超时）丢失强行终止该令牌，损失它的能力。

一个简单的方法，有一些JavaScript可以减轻一个活跃的用户体验超时，让用户知道他们的会话即将到期。用户可以点击一个按钮继续，并刷新会话。

另外，指定一个自定义的 `AccessDeniedHandler` 允许你以任何你喜欢的方式处理 `InvalidCsrfTokenException` 。 有关如何自定义 `AccessDeniedHandler` 的例子参考提供的链接为 xml 和 Java configuration.

最后，应用程序可以配置为使用 CookieCsrfTokenRepository这将不会过期。 正如前面提到的,这不是安全的使用一个会话,但在许多情况下已经足够好。

Logging In

为了防止 forging登录请求 在登录表单中应该也要防止CSRF攻击。因为`CsrfToken`存储在HttpSession中，这意味着一个HttpSession将尽快创建`CsrfToken`令牌属性进行访问。 虽然这听起来很糟糕，一个RESTful / stateless的体系结构对于实现实际的安全是必要的。如果没有的状态，我们什么都没有，如果令牌被攻破，我们可以做的。实事求是地讲，在CSRF令牌的尺寸非常小，应该对我们的架构可以忽略不计的影响。如果没有状态，如果令牌被攻破，我们没有什么可以做的。实际上,CSRF令牌的规模和体系结构很小应该对我们的架构有一个微不足道的影响。

一种保护登录常用技术是使用一个javascript函数在表单提交之前获得一个有效的CSRF令牌。通过这样做，没有必要去想会话超时因为表单提交前有创建会话（在上一节中讨论）（假设 CookieCsrfTokenRepository 没有配置而不是), 因此，用户当他想提交用户名/密码可以停留在登录页面上为了达到这个目标,你可以采取Spring Security提供的`CsrfTokenArgumentResolver`的优势，并像它在这里描述暴露的端点。

Logging Out

添加CSRF将更新LogoutFilter只使用HTTP POST。这确保了注销需要CSRF令牌和一个恶意的用户不能强制注销用户。

一种方法是使用一种形式的日志。如果你真的想要一个链接，你可以（即也许隐藏表单）使用JavaScript，让链接执行POST。对于禁止使用JavaScript的浏览器,您可以选择有链接的用户注销确认页面执行POST。

如果你真的想使用HTTP GET注销你可以这样做,但请记住这一般不推荐。例如,下面的Java配置将执行任意的HTTP方法请求URL /logout执行注销:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
    .logout()
     .logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
 }
}
```

Multipart (文件上传)

有两个选项来使用CSRF保护multipart/form-data.每个选项都有其权衡。

- Placing MultipartFilter before Spring Security
- Include CSRF token in action

> 之前你和multipart集成Spring Security的CSRF保护文件上传，确保没有CSRF保护你可以第一时间上传。使用multipart表单与Spring的更多信息可以在 17.10 Spring’s multipart (文件上传) 支持 的Spring参考和 MultipartFilter javadoc.

Placing MultipartFilter before Spring Security

第一个选项是确保`MultipartFilter`在Spring Security过滤器之前指定。Spring Security过滤器之前指定`MultipartFilter`意味着没有授权调用`MultipartFilter`这意味着任何人都可以服务器上上传临时文件。不过，只有授权的用户可以将文件提交由您的应用程序处理。总的来说,这是推荐的方法,因为临时文件上传在大多数服务器上应该只有一个微不足道的影响。

为了确保`MultipartFilter`是Spring Security的过滤器，在Java配置前指定，用户可以重写beforespringsecurityfilterchain如下所示:

```
 public class SecurityApplicationInitializer extends AbstractSecurityWebApplicationInitializer {

 @Override
 protected void beforeSpringSecurityFilterChain(ServletContext servletContext) {
  insertFilters(servletContext, new MultipartFilter());
 }
}
```

为了确保`MultipartFilter`是Spring Security的过滤器，在XML配置前指定，用户可以确定`MultipartFilter`的<filter-mapping>元素在springSecurityFilterChain前在web.xml中配置如下所示:

```
 <filter>
 <filter-name>MultipartFilter</filter-name>
 <filter-class>org.springframework.web.multipart.support.MultipartFilter</filter-class>
</filter>
<filter>
 <filter-name>springSecurityFilterChain</filter-name>
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
```

```
</filter>
<filter-mapping>
 <filter-name>MultipartFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
 <filter-name>springSecurityFilterChain</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Include CSRF token in action

如果允许未经授权的用户上传temporariy文件是不能接受的，另一种是将Spring Security过滤后的`MultipartFilter`和CSRF作为查询参数包含在表单的动作属性。在jsp示例如下所示

```
 <form action="./upload?${_csrf.parameterName}=${_csrf.token}" method="post" enctype="multipart/form-data">
```

这种方法的缺点是，查询参数可以被泄露。更普遍的是,最佳的方法是将敏感数据在身体或头部,以确保不泄露。更多信息可以在[RFC 2616 Section 15.1.3 编码URI的敏感信息](#)中找到。

HiddenHttpMethodFilter

该HiddenHttpMethodFilter应放在Spring Security的过滤器之前。一般来说这是事实，但它可能能够对防止CSRF攻击有更多的影响。

请注意，HiddenHttpMethodFilter只覆盖一个POST HTTP方法，所以这实际上是不可能造成任何实际问题。但是，它仍然是最好的做法，以确保它被放置在Spring Security过滤器之前。

重写默认值

Spring Security的目标是提供默认值,保护用户免受攻击。这并不意味着你被迫接受所有的默认设置。

例如，您可以提供一个自定义CsrfTokenRepository覆盖其中`CSRF Token`的存储方式。

您还可以指定一个自定义RequestMatcher确定哪些请求受CSRF保护(即也许你不在乎注销是利用)。简而言之,如果Spring Security CSRF保护没有做出你想要的,你可以自定义行为。 请参阅 [<csrf>将添加元素](#)文档的细节如何自定义使用XML和`CsrfConfigurer` javadoc,了解如何使用Java配置将这些自定义设置。

CORS

Spring框架提供了 [CORS第一级支持](#)。 CORS必须在Spring Security之前处理因为预检要求将不包含任何cookie（即`JSESSIONID`）。 如果第一个请求不包含任何cookie和Spring Security，将决定用户的请求没有被验证（因为有请求没有cookies），并拒绝它。

最简单的方法就是确保CORS首先是使用`CorsFilter`处理。 用户可以把`CorsFilter`和Spring Security 提供一个`CorsConfigurationSource`使用如下:

```
 @EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
   // by default uses a Bean by the name of corsConfigurationSource
   .cors().and()
   ...
 }

 @Bean
 CorsConfigurationSource corsConfigurationSource() {
  CorsConfiguration configuration = new CorsConfiguration();
  configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
  configuration.setAllowedMethods(Arrays.asList("GET","POST"));
  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
  source.registerCorsConfiguration("/**", configuration);
  return source;
 }
}
```

或在XML中

```
 <http>
 <cors configuration-source-ref="corsSource"/>
 ...
</http>
<b:bean id="corsSource" class="org.springframework.web.cors.UrlBasedCorsConfigurationSource">
```

```
 ...
</b:bean>
```

如果你正在使用Spring MVC的CORS支持，你可以省略指定 `CorsConfigurationSource` 和Spring Security将利用CORS配置提供给Spring MVC。

```
 @EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
   // if Spring MVC is on classpath and no CorsConfigurationSource is provided,
   // Spring Security will use CORS configuration provided to Spring MVC
   .cors().and()
   ...
 }
}
```

或在XML中

```
 <http>
 <!-- Default to Spring MVC's CORS configuraiton -->
 <cors />
 ...
</http>
```

安全HTTP响应头

本节讨论Spring Security支持将各种安全头添加到响应。

默认的安全头

Spring Security允许用户轻松地注入的默认安全标头来帮助保护他们的应用程序。Spring Security默认是包括以下头部:

```
 Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

Strict-Transport-Security只有添加HTTPS请求

有关这些标题的更多详细信息，请参见相应的章节:

- Cache Control
- Content Type Options
- HTTP Strict Transport Security
- X-Frame-Options
- X-XSS-Protection

而这些头文件被认为是最佳方法,应该注意的是,并不是所有的客户端利用头部,所以鼓励进行额外的测试。

您可以自定义特定的头文件。 例如,假设希望你的HTTP响应头部为以下的样子:

```
 Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
```

具体而言，你希望自定义所有的默认标题与下列:

- X-Frame-Options允许来自同一个域中的任何请求
- HTTP Strict Transport Security (HSTS) 将不会被添加到响应

你可以很容易地用下面的Java配置做到这一点:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
  WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
   // ...
   .headers()
    .frameOptions().sameOrigin()
    .httpStrictTransportSecurity().disable();
 }
}
```

另外，如果你正在使用Spring Security的XML配置，你可以使用下面的:

```
 <http>
 <!-- ... -->

 <headers>
  <frame-options policy="SAMEORIGIN" />
  <hsts disable="true"/>
 </headers>
</http>
```

如果你不想要添加默认值,需要明确控制应该使用什么,您可以禁用默认值。一个例子提供了Java和基于XML的配置:

如果您正在使用Spring安全的Java配置以下只会增加 Cache Control.

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  // do not use any default headers unless explicitly listed
  .defaultsDisabled()
  .cacheControl();
}
}
```

下面的XML只会增加Cache Control.

```
 <http>
 <!-- ... -->

 <headers defaults-disabled="true">
  <cache-control/>
 </headers>
</http>
```

如果有必要,你可以禁用所有的HTTP安全响应头下面的Java配置:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers().disable();
}
}
```

如果有必要,你可以禁用所有的HTTP安全响应头下面的XML配置:

```
 <http>
 <!-- ... -->

 <headers disabled="true" />
</http>
```

缓存控制

在过去的Spring Security要求你提供自己的缓存控制您的web应用程序。这似乎是合理的,但是浏览器缓存已经进化到包括缓存安全连接。这意味着用户可以查看经过身份验证的页面,注销,然后恶意用户可以使用浏览器历史记录查看缓存页面。为了帮助缓解Spring Security增加了对高速缓存控制的支持，这将插入以下头部作为你的回应。

```
 Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

```
Pragma: no-cache
Expires: 0
```

简单的添加<头部 没有子元素的元素会自动添加缓存控制和不少其他保护措施。 不过，如果你只想要缓存控制，你可以使用Spring Security的XML命名空间与 <cache-control> 元素和headers@defaults-disabled 属性。

```
 <http>
<!-- ... -->

<headers defaults-disable="true">
 <cache-control />
</headers>
</http>
```

同样，您可以在Java配置中启用缓存控制类似以下:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .defaultsDisabled()
  .cacheControl();
}
}
```

如果你真的想要缓存特定的反应,您的应用程序可以选择性地调用 HttpServletResponse.setHeader(String,String) 覆盖头部Spring Security的设置。为了保证CSS，JavaScript之类的东西是用的并且图像正确缓存。

当使用Spring Web MVC,这通常是在你的配置中。例如,下面的配置设置将确保缓存头你所有的资源:

```
 @EnableWebMvc
public class WebMvcConfiguration extends WebMvcConfigurerAdapter {

 @Override
 public void addResourceHandlers(ResourceHandlerRegistry registry) {
  registry
   .addResourceHandler("/resources/**")
   .addResourceLocations("/resources/")
   .setCachePeriod(31556926);
 }

 // ...
}
```

内容类型选项

历史上的浏览器，包括Internet Explorer，试图想请求的内容类型使用 content sniffing。这就使得浏览器通过猜测来改善用户体验的内容类型没有指定内容类型的资源。例如,如果一个浏览器遇到一个JavaScript文件,该文件没有指定的内容类型,它会猜内容类型,然后执行它。

> 有许多额外的一个人应该做的事情(即只显示文档在不同的领域,确保设置内容类型头,清理文件等等)时,从而允许内容上传。然而,这些措施的范围之外的Spring Security提供什么。同样重要的是指出禁用content sniffing的时候，你必须要指定内容类型的东西才能正常工作。

content sniffing的问题是，这允许恶意用户使用polyglots（即一个文件，是作为多种内容类型有效）来执行XSS攻击。例如，某些网站可能会允许用户提交一个有效的PostScript文档到网站，并查看它。恶意用户可能会创建一个 postscript文件，这也是一个有效的JavaScript文件 并用它执行XSS攻击。

通过添加以下content sniffing可以禁用我们的响应头:

```
 X-Content-Type-Options: nosniff
```

就像缓存控制元件,nosniff指令添加默认情况下使用<headers>元素时,没有子元素。 然而,如果你想要更多的控制,添加头文件可以使用 <content-type-options>元素和headers@defaults-disabled属性，如下所示:

```
 <http>
<!-- ... -->

<headers defaults-disabled="true">
 <content-type-options />
</headers>
</http>
```

X-Content-Type-Options头默认情况下使用Spring Security添加Java配置。如果你想要更多的控制头,您可以显式地指定以下内容类型选项:

```
  @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .defaultsDisabled()
  .contentTypeOptions();
}
}
```

HTTP Strict Transport Security (HSTS)

当你输入你的银行的网站,你输入mybank.example.com 或者你输入 https://mybank.example.com? 如果您省略https协议,你可能容易受到 中间人攻击。即使网站执行重定向到 https://mybank.example.com 恶意用户能够拦截最初的HTTP请求和操作响应(即重定向到 https://mibank.example.com 和窃取他们的凭证)。

很多用户忽略了https协议，这就是为什么 HTTPHTTP严格传输安全（HSTS） 已创建。一旦mybank.example.com 添加为 HSTS host,浏览器可以提前知道任何请求 mybank.example.com 应该解读为 https://mybank.example.com.这大大降低了发生中间人攻击的可能性。

> 按照 RFC6797, HSTS头仅注入到HTTPS响应。为了让浏览器承认头，浏览器必须先信任CA签署的用于建立连接的SSL证书（不只是SSL证书）。

一个网站被标记为HSTS主机有主机预装到浏览器的一个方法。另一种是将"Strict-Transport-Security"头添加到响应。例如，以下将指示浏览器把域作为一年的HSTS主机（一年有大约31536000秒）：

```
  Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

可选includeSubDomains指令指示Spring安全子域(即secure.mybank.example.com)也应该被视为一个 HSTS域。

至于其它头部，Spring Security的默认添加HSTS。您可以自定义HSTS头部与<hsts>元素如下所示:

```
  <http>
 <!-- ... -->

 <headers>
  <hsts
   include-subdomains="true"
   max-age-seconds="31536000" />
 </headers>
</http>
```

类似地,您可以只启用HSTS头与Java配置:

```
  @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .httpStrictTransportSecurity()
   .includeSubdomains(true)
   .maxAgeSeconds(31536000);
}
}
```

HTTP Public Key Pinning (HPKP)

HTTP Public Key Pinning (HPKP) 是一个安全特性，它讲述了一个web客户端关联特定加密公钥与一个特定的web服务器以防止中间人(MITM)攻击和伪造的证书。

为了确保在TLS会话中使用服务器的公钥的真实性，这个公钥被包装成一个X。509证书通常由一个证书颁发机构(CA)签名。Web客户端如浏览器信任了很多这些CA，这都可以创建任意域名证书。 如果攻击者能够破坏一个CA，他们可以执行各种TLS连接MITM攻击。 HPKP可以规避HTTPS协议这一威胁,告诉客户端公钥属于一个特定的web服务器。 HPKP是在首次使用(TOFU)技术信托。第一次web服务器经由该公钥属于它的特殊HTTP头告诉客户端，客户端存储此信息的给定时间段。 当客户端访问服务器,它预计包含公钥证书的指纹通过HPKP已知。如果服务器提供一个未知的公共密钥，客户端应呈现警告给用户。

因为用户代理需要验证对SSL证书链的链销,HPKP头只是注入HTTPS反应。

为你的网站启用这个特性很简单,当在HTTPS访问你的网站时返回Public-Key-Pins HTTP头。 例如，以下将指导用户代理只报告密码验证失败的给定URI(via the ***report-uri*** directive) 2 引脚:

```
 Public-Key-Pins-Report-Only: max-age=5184000 ; pin-sha256="d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=" ; pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=" ; report-uri="http://example.net/pkp-report" ; includeSubDomains
```

一个 ***pin 验证失败 report*** 是一个标准的JSON结构,可以捕获通过web应用程序的API或HPKP主办的公开报告服务, 比如, ***REPORT-URI***.

可选的includeSubDomains指令指示浏览器也与给定引脚验证子域。

相对于其他的头部，Spring Security没有默认添加HPKP。你可以定制HPKP头与<hpkp> 元素如下所示:

```
 <http>
<!-- ... -->

<headers>
 <hpkp
  include-subdomains="true"
  report-uri="http://example.net/pkp-report">
  <pins>
    <pin algorithm="sha256">d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=</pin>
    <pin algorithm="sha256">E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=</pin>
  </pins>
 </hpkp>
</headers>
</http>
```

同样，您可以启用Java配置HPKP头:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
    // ...
    .headers()
      .httpPublicKeyPinning()
        .includeSubdomains(true)
        .reportUri("http://example.net/pkp-report")
        .addSha256Pins("d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=", "E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
  }
}
```

X-Frame-Options

允许给你的网站添加框架可能是一个安全问题。例如，使用巧妙的CSS样式用户可能会被欺骗点击的东西，他们不打算 (video demo)。 例如，登录到他们的银行用户可能会点击一个按钮授予其他用户访问。这种攻击被称为 Clickjacking.

另一种现代的方式来处理点击劫持是使用 [headers-content-security-policy].

有很多方法来缓解点击劫持攻击。例如，为了保护您可以使用传统的点击劫持的攻击的浏览器 frame breaking code. 虽然并不完美，但断码的框架是可以为传统的浏览器做的最好的。

解决点击劫持更先进的方法是使用 X-Frame-Options 头:

```
 X-Frame-Options: DENY
```

X-Frame-Options响应头指示浏览器阻止任何网站，这个头在响应中被呈现在一个框架中。 默认情况下,Spring Security禁用iframe内呈现。

你可以定制X-Frame-Options和 frame-options 元素。 例如,以下将指示Spring Security用 "X-Frame-Options: SAMEORIGIN" 允许iframes在同一个域:

```
 <http>
<!-- ... -->

<headers>
 <frame-options
 policy="SAMEORIGIN" />
```

```
 </headers>
</http>
```

同样，您可以自定义框架选项在Java配置中使用相同的起源:

```
  @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .frameOptions()
   .sameOrigin();
}
}
```

X-XSS-Protection

有些浏览器已经内置支持过滤掉 reflected XSS attacks。这决不是完整的证明,但协助XSS的保护。

过滤通常是默认启用,所以添加头部通常只是确保它被启用和指示浏览器检测到XSS攻击时该做什么。例如,过滤器可能试图改变的内容至少入侵方式仍然呈现一切。有时,这种类型的替代可以成为 XSS 漏洞本身。相反，它是最好的阻止内容，而不是试图修复它。要做到这一点，我们可以添加以下头部:

```
  X-XSS-Protection: 1; mode=block
```

在默认情况下已经包括了这个头。然而，我们可以自定义它，如果我们想。 例如:

```
  <http>
 <!-- ... -->

 <headers>
  <xss-protection block="false"/>
 </headers>
</http>
```

同样，您可以在Java配置中自定义XSS保护如下:

```
  @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .xssProtection()
   .block(false);
}
}
```

Content Security Policy (CSP)

Content Security Policy (CSP)是一个web应用程序可以利用的机制来减轻内容注入漏洞,如跨站点脚本（XSS）。 CSP是一种声明性政策,为web应用程序提供了一个工具的作者声明，并最终通知客户端(用户代理)对web应用程序的来源将加载资源。

> 内容安全策略不是为了解决所有的内容注入漏洞。相反，CSP可以利用，以帮助减少因内容注入攻击的危害。作为防御的第一道防线，Web应用程序的作者应该验证它们的输入和编码它们的输出。

web应用程序可以通过使用CSP使用下列其中一个HTTP头的响应:

- ***Content-Security-Policy***
- ***Content-Security-Policy-Report-Only***

每个这些头被用作一种机制，以提供一个 ***security policy*** 到客户端。 安全策略包含了一组 ***security policy directives*** (例如, *script-src* 和 *object-src*), 每个负责宣布限制为特定的资源表示形式。

例如，Web应用程序可以声明，预计加载特定，可信来源的脚本，通过再响应以下头部:

```
  Content-Security-Policy: script-src https://trustedscripts.example.com
```

试图加载脚本以外另一个来源是什么中声明*script-src*指令将被用户代理。 此外，如果 ***report-uri*** 指令在安全政策声明，然后违反将由用户代理来声明的URL进行报告.

例如,如果一个web应用程序违反了安全政策声明,下面的响应头将指导用户代理发送违反政策的报告中指定的URL *report-uri* 指令。

```
Content-Security-Policy: script-src https://trustedscripts.example.com; report-uri /csp-report-endpoint/
```

Violation reports 是标准JSON结构,可以捕获通过web应用程序的API或公开托管CSP违反报告服务,比如, ***REPORT-URI***.

Content-Security-Policy-Report-Only头提供了web应用程序的作者的能力和管理员监控安全策略,而不是执行它们。 这个头通常用在当试验and/or开发一个网站的安全策略。 当一个政策被认为是有效的,它可以执行使用 *Content-Security-Policy* 头字段。

鉴于以下响应头,政策声明脚本可能加载两种可能的来源。

```
Content-Security-Policy-Report-Only: script-src 'self' https://trustedscripts.example.com; report-uri /csp-report-
endpoint/
```

如果违反了这一政策,试图从 *evil.加载脚本, 用户代理将发送一个违反报告宣布 _report-uri* 指令指定的URL,但仍然允许违反资源负载。

Configuring Content Security Policy

重要的是要注意 Spring Security ***不添加***内容默认安全策略。 web应用程序的作者必须声明安全策略执行and/or对受保护的资源进行监测。

例如,下面的安全策略:

```
script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-
report-endpoint/
```

你可以使用XML配置启用CSP头与 <content-security-policy元素如下所示:

```
<http>
<!-- ... -->

<headers>
 <content-security-policy
  policy-directives="script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/" />
</headers>
</http>
```

要启用CSP '*report-only*'头部, 配置元素如下所示:

```
<http>
<!-- ... -->

<headers>
 <content-security-policy
  policy-directives="script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
  report-only="true" />
</headers>
</http>
```

类似地,您可以使用Java配置启用CSP头，如下所示:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .contentSecurityPolicy("script-src 'self' https://trustedscripts.example.com; object-src
https://trustedplugins.example.com; report-uri /csp-report-endpoint/");
}
}
```

要启用CSP '*report-only*' 头部, 提供以下Java配置:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

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

```
 .headers()
  .contentSecurityPolicy("script-src 'self' https://trustedscripts.example.com; object-src
https://trustedplugins.example.com; report-uri /csp-report-endpoint/")
  .reportOnly();
}
}
```

Additional Resources

内容安全策略应用到一个web应用程序通常是一个不简单的事情。 以下资源可以为你的网站发展有效的安全策略提供进一步的援助。

[An Introduction to Content Security Policy]

[CSP Guide - Mozilla Developer Network]

[W3C Candidate Recommendation]

Custom Headers

Spring安全机制,使它更加方便的将共同安全头添加到您的应用程序。并且,它还提供了钩子,能够添加自定义头部。

Static Headers

也许有些时候你希望注入自定义的安全标头不支持您的应用程序的。例如,下面的自定义安全头:

```
 X-Custom-Security-Header: header-value
```

当使用XML命名空间,这些头部可以添加到响应使用 <header>元素如下所示:

```
 <http>
<!-- ... -->

<headers>
 <header name="X-Custom-Security-Header" value="header-value"/>
</headers>
</http>
```

同样，头部可以在Java配置中添加到响应如下所示:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .addHeaderWriter(new StaticHeadersWriter("X-Custom-Security-Header","header-value"));
}
}
```

Headers Writer

当命名空间或Java配置不支持你想要的头部你可以自定义一个`HeadersWriter`实例甚至提供一个自定义实现的`HeadersWriter`。

让我们看看一个例子使用一个自定义的`XFrameOptionsHeaderWriter`的实例。也许你想允许框架内容有着相同的起源。这很容易支持通过设置 policy 属性 "SAMEORIGIN", 但让我们看看一个更明确的例子使用 ref 属性。

```
 <http>
<!-- ... -->

<headers>
 <header ref="frameOptionsWriter"/>
</headers>
</http>
<!-- Requires the c-namespace.
See http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-c-namespace
-->
<beans:bean id="frameOptionsWriter"
 class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsHeaderWriter"
 c:frameOptionsMode="SAMEORIGIN"/>
```

我们还可以限制框架的内容为与Java配置相同的起源:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 // ...
 .headers()
  .addHeaderWriter(new XFrameOptionsHeaderWriter(XFrameOptionsMode.SAMEORIGIN));
}
}
```

DelegatingRequestMatcherHeaderWriter

有时你可能想要只对某些请求写头部。例如,也许你想只保护你的登录页面不被攻击，你可以使用 `DelegatingRequestMatcherHeaderWriter` 。 当使用XML命名空间配置, 这可以如下所示:

```
 <http>
<!-- ... -->

<headers>
 <frame-options disabled="true"/>
 <header ref="headerWriter"/>
</headers>
</http>

<beans:bean id="headerWriter"
 class="org.springframework.security.web.header.writers.DelegatingRequestMatcherHeaderWriter">
 <beans:constructor-arg>
  <bean class="org.springframework.security.web.util.matcher.AntPathRequestMatcher"
   c:pattern="/login"/>
 </beans:constructor-arg>
 <beans:constructor-arg>
  <beans:bean
   class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsHeaderWriter"/>
 </beans:constructor-arg>
</beans:bean>
```

我们还可以使用java配置防止框架的内容到登录页面:

```
 @EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
 RequestMatcher matcher = new AntPathRequestMatcher("/login");
 DelegatingRequestMatcherHeaderWriter headerWriter =
  new DelegatingRequestMatcherHeaderWriter(matcher,new XFrameOptionsHeaderWriter());
 http
 // ...
 .headers()
  .frameOptions().disabled()
  .addHeaderWriter(headerWriter);
}
}
```

会话管理

HTTP会话相关的功能性是通过 `SessionManagementFilter` 和 `SessionAuthenticationStrategy` 接口结合来处理，筛选出一个代表.典型用法包括会话固定保护攻击预防，检测到会话超时并且限制有多少会话经过身份验证的用户可以同时打开。

会话管理过滤器

`SessionManagementFilter` 检测 `SecurityContextRepository` 的内容违背了现在 `SecurityContextHolder` 的内容来决定是否在当前请求用户已经通过身份验证，通常由一个非交互式的身份验证机制，如验证或记脚注:[身份验证机制验证后进行重定向（例如form-login）将不会被 `SessionManagementFilter` 检测到，过滤器将不调用身份验证请求，会话管理功能必须在这些情况下分别处理].如果存储库包含一个安全文档，过滤器什么都不做，如果这没有这么做，局部线程 `SecurityContext` 包含了一个（不是匿名） `Authentication` 对象.

如果用户当前没有经过身份验证,过滤器将检查是否一个无效的请求会话ID（例如由于超时），并将调用 `InvalidSessionStrategy` 的配置，如果一个被设置了.最通常的行为是重定向到一个固定的URL，这是将实现 `SimpleRedirectInvalidSessionStrategy` 进行实施.无效的会话时后者也会使用当通过命名空间配置.as described earlier.

会话的身份验证策略

`SessionAuthenticationStrategy` 通过 `SessionManagementFilter` 和 `AbstractAuthenticationProcessingFilter` 被利用，所以如果你利用了一个定制的form-login例如，你将需要把它注入到这两个，在这种情况下，一个典型的配置，结合命名空间和自定义的beans可能看起来像这样 ：

```
  <http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />
<session-management session-authentication-strategy-ref="sas"/>
</http>

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
 <beans:property name="sessionAuthenticationStrategy" ref="sas" />
 ...
</beans:bean>

<beans:bean id="sas" class=
"org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy" />
```

注意默认的使用，`SessionFixationProtectionStrategy`可能会导致问题，如果你是存储在session中的bean实施`HttpSessionBindingListener`，包括Spring session-scoped beans，看这个类中的javadoc寻找更多信息.

并发控制

Spring Security能防止主体同时访问同一应用程序的多个指定的次数，许多软件公司利用这一强制许可.虽然网络管理员喜欢这个功能，因为它有助于防止人们共享登录名.例如你能停止"Batman"用户从两个不同的会话登录到Web应用程序 ，您可以终止他们的以前的登录，或者当他们尝试再次登录时，你可以报告一个错误 ，预防第二次登陆，请注意，如果您使用的第二种方法，一个没有显式退出的用户（例如谁刚刚关闭了浏览器）将无法再次登录，直到他们的原始会话过期.

并发控制是由命名空间支持的，所以请检查早期的命名空间章节的最简单的配置。虽然有时你需要自定义的东西。

该实现使用了一个`SessionAuthenticationStrategy`的专门的版本，叫做`ConcurrentSessionControlAuthenticationStrategy`.

> 以前并发的身份验证检查通过`ProviderManager`制作，由`ConcurrentSessionController`注射，后者将检查用户是否试图超过允许的会话数.尽管如此，这种方法需要一个HTTP会话提前被创建.这是不受欢迎的，在Spring Security 3, 用户首先经过`AuthenticationManager`进行身份验证，而且一旦他们成功地通过身份验证，就会创建一个会话，并检查是否允许他们有另一个会话打开.

要使用并发会话支持，您需要添加以下`web.xml`:

```
  <listener>
 <listener-class>
 org.springframework.security.web.session.HttpSessionEventPublisher
 </listener-class>
</listener>
```

除此之外，你将需要添加`ConcurrentSessionFilter`到你的`FilterChainProxy`.`ConcurrentSessionFilter`要求两个属性，`sessionRegistry`一般是指一个`SessionRegistryImpl`的实例，`expiredUrl`指向页面时显示会话已过期的页面，一个配置利用命名空间去创建`FilterChainProxy`和不符合要求的beans如下:

```
  <http>
<custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter" />
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />

<session-management session-authentication-strategy-ref="sas"/>
</http>

<beans:bean id="concurrencyFilter"
class="org.springframework.security.web.session.ConcurrentSessionFilter">
<beans:property name="sessionRegistry" ref="sessionRegistry" />
<beans:property name="expiredUrl" value="/session-expired.htm" />
</beans:bean>

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<beans:property name="sessionAuthenticationStrategy" ref="sas" />
<beans:property name="authenticationManager" ref="authenticationManager" />
</beans:bean>

<beans:bean id="sas" class="org.springframework.security.web.authentication.session.CompositeSessionAuthenticationStrategy">
<beans:constructor-arg>
 <beans:list>
 <beans:bean class="org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy">
  <beans:constructor-arg ref="sessionRegistry"/>
  <beans:property name="maximumSessions" value="1" />
  <beans:property name="exceptionIfMaximumExceeded" value="true" />
 </beans:bean>
 <beans:bean class="org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy">
 </beans:bean>
 <beans:bean class="org.springframework.security.web.authentication.session.RegisterSessionAuthenticationStrategy">
  <beans:constructor-arg ref="sessionRegistry"/>
 </beans:bean>
```

```
 </beans:list>
</beans:constructor-arg>
</beans:bean>

<beans:bean id="sessionRegistry"
 class="org.springframework.security.core.session.SessionRegistryImpl" />
```

每一次 `HttpSession` 开始或者结束,添加一个监听到 `web.xml` 导致一个 `ApplicationEvent` 被发布到Spring `ApplicationContext` .这很关键，因为当一个会话结束时这个允许 `SessionRegistryImpl` 被通知，即使他们退出了另一个会话或它超时.

查询sessionregistry目前身份验证的用户和他们的会话

建立并发控制通过命名空间或使用bean具有有益的副作用，为您提供一个参考的` sessionregistry 可以直接应用在在您的应用程序.因此,即使你不想限制用户会话的数量,它可能值得建立基础设施.你可以建立`maximumSession 财产到- 1允许无限的会话.如果你使用的命名空间，您可以利用 `session-registry-alias` 属性为内部创建 `SessionRegistry` 的别名设置.提供一个参考，你可以注射到你自己的bean.

`getAllPrincipals()` 方法为您提供当前身份验证的用户的列表.您可以列出用户会话通过呼叫 `getAllSessions(Object principal, boolean includeExpiredSessions)` 方法，返回 `SessionInformation` 对象列表.您也可以终止用户会话通过在 `SessionInformation` 实例中呼叫 `expireNow()` ，当用户返回到应用程序时，他们将被阻止进行。您可能会发现这些方法在一个管理应用程序中是有用的，例如.看看javadoc的更多信息

匿名身份验证

概述

采取 "deny-by-default"在您显式指定什么是允许和禁止一切，它通常被认为是良好的安全实践.定义什么是未经身份验证的用户访问的是一个类似的情况,特别是用于Web应用程序.许多网站要求用户必须对其他比一些网址的任何东西进行身份验证(例如，主界面和登录页面).在这种情况下，它是最简单的定义访问这些特定的网址的访问配置属性，而不是为每一个安全的资源.不同的,在默认情况下` role_something `要求很好说，但有时候也有例外.例如一个应用的登陆，注销，和主页面.你也可以从过滤器链完全忽略这些页面,从而绕过访问控制检查,但这可能是不受欢迎的其他原因,特别是如果经过身份验证的用户的页面的行为会有所不同.

这就是我们所说的匿名身份验证，请注意,没有真正的概念区别用户"anonymously authenticated"和未经过身份验证的用户.Spring Security的匿名身份验证只给您一个更加方便的方式来配置您的访问控制属性.如 `getCallerPrincipal` 调用servlet API，仍将返回null,尽管实际上 `SecurityContextHolder` 中的一个匿名认证对象.

还有其他的匿名身份验证的情况下是有用的,例如，当一个审计拦截器查询 `SecurityContextHolder` 来识别哪个主要是负责一个给定的操作.类可以创建更强劲,如果他们知道 `SecurityContextHolder` 总是包含一个“身份验证”对象,而且从不 `null` .

配置

匿名认证提供支持自动使用HTTP配置Spring Security 3时可定制（或禁用）使用 `<anonymous>` 元元素.您不需要配置这里所描述的beans，除非你使用传统的bean配置.

三个类 一起提供匿名身份验证功能. `AnonymousAuthenticationToken` 是一个实现 `Authentication` ,商店的` GrantedAuthority 适用于匿名委托,有相应的 anonymousauthenticationprovider ,这是链接到 providermanager ,以至于 `AnonymousAuthenticationToken 能接受.最后，有一个` anonymousauthenticationfilter ， 这是把正常的认证机制后, 自动添加一个 anonymousauthenticationtoken 的 securitycontextholder `,如果没有现有的身份验证在那里举行.筛选器和身份验证提供程序的定义如下：

```
  <bean id="anonymousAuthFilter"
 class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">
<property name="key" value="foobar"/>
<property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>

<bean id="anonymousAuthenticationProvider"
 class="org.springframework.security.authentication.AnonymousAuthenticationProvider">
<property name="key" value="foobar"/>
</bean>
```

在筛选器和身份验证提供程序之间共享“key”,使前者所创建的符号被后者的脚注所接受[key 属性的使用不应被视为在这里提供任何真正的安全性.它仅仅是一个簿记锻炼, 如果你是共享一个` providermanager 包含 anonymousauthenticationprovider , 在这样的情形下, 它是可能的认证客户端构建`Authentication 对象(如RMI调用),然后，一个恶意的客户端可以提交一个` anonymousauthenticationtoken ， 它创造了自己 (选择的用户名和权限列表 ).如果`key 是人或能被发现，然后代表将被匿名者接受，这不是正常使用的问题，但如果你使用RMI你最好使用一个定制的没有匿名的` providermanager 提供者而不是分享你使用HTTP认证机制. ]. ` userattribute `表达的形式 usernameintheauthenticationtoken，授予权威 ,这是相同的语法使

用等号后的`InMemoryDaoImpl`的`userMap”属性.

如前所述,匿名身份验证的好处是,所有URI模式可以安全应用。例如

```
  <bean id="filterSecurityInterceptor"
 class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
<property name="securityMetadata">
 <security:filter-security-metadata-source>
 <security:intercept-url pattern='/index.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
 <security:intercept-url pattern='/hello.htm' access='ROLE_ANONYMOUS,ROLE_USER'/>
 <security:intercept-url pattern='/logoff.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
 <security:intercept-url pattern='/login.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
 <security:intercept-url pattern='/**' access='ROLE_USER'/>
 </security:filter-security-metadata-source>" +
</property>
</bean>
```

验证信任解析器

排在匿名身份验证讨论是`AuthenticationTrustResolver`界面,与相应的`AuthenticationTrustResolverImpl`实现.这个接口提供了一个`isAnonymous(Authentication)`的方法,感兴趣的类可以考虑这种特殊类型的身份验证状态. `ExceptionTranslationFilter`使用这个接口在处理`AccessDeniedException`,如果抛出`AccessDeniedException`,和一个匿名类型的身份验证,而不是扔403(禁止)的反应,过滤器将开始` authenticationentrypoint 所以可以验证正确.这是一个必要的区别,否则，负责将永远被视为"authenticated"，并没有得到一个机会通过表格登录,基本、摘要或其他一些正常的身份验证机制。你会经常看到 role_anonymous 属性在上述拦截器配置更换 is_authenticated_anonymously ，这实际上是一样的定义访问控制,这是一个例子，使用的 authenticatedvoter 我们将看到在authorization chapter.它使用一个 authenticationtrustresolver 处理这个特定的配置属性和访问权限授予匿名用户.它使用一个 authenticationtrustresolver 处理这个特定的配置属性和访问权限授予匿名用户. authenticatedvoter 方法更强大，因为它允许你区分匿名,请记住我和完全身份验证的用户.如果你不需要这个功能,然后你可以坚持`ROLE_ANONYMOUS`,这将是由Spring Security的标准` rolevoter `.

WebSocket Security

Spring Security 4 added support for securing Spring’s WebSocket support. This section describes how to use Spring Security’s WebSocket support.

> You can find a complete working sample of WebSocket security in samples/javaconfig/chat.

Direct JSR-356 Support

Spring Security does not provide direct JSR-356 support because doing so would provide little value. This is because the format is unknown, so there is little Spring can do to secure an unknown format. Additionally, JSR-356 does not provide a way to intercept messages, so security would be rather invasive.

WebSocket Configuration

Spring Security 4.0 has introduced authorization support for WebSockets through the Spring Messaging abstraction. To configure authorization using Java Configuration, simply extend the `AbstractSecurityWebSocketMessageBrokerConfigurer` and configure the `MessageSecurityMetadataSourceRegistry`. For example:

```
  @Configuration
public class WebSocketSecurityConfig
      extends AbstractSecurityWebSocketMessageBrokerConfigurer { ❶ ❷

    protected void configureInbound(MessageSecurityMetadataSourceRegistry messages) {
        messages
                .simpDestMatchers("/user/*").authenticated() ❸
    }
}
```

This will ensure that:

❶ Any inbound CONNECT message requires a valid CSRF token to enforce Same Origin Policy

❷ The SecurityContextHolder is populated with the user within the simpUser header attribute for any inbound request.

3. Our messages require the proper authorization. Specifically, any inbound message that starts with "/user/" will require ROLE_USER. Additional details on authorization can be found in WebSocket Authorization

Spring Security also provides XML Namespace support for securing WebSockets. A comparable XML based configuration looks like the following:

```
<websocket-message-broker> (1) (2)
  (3)
  <intercept-message pattern="/user/**" access="hasRole('USER')" />
</websocket-message-broker>
```

This will ensure that:

1. Any inbound CONNECT message requires a valid CSRF token to enforce Same Origin Policy
2. The SecurityContextHolder is populated with the user within the simpUser header attribute for any inbound request.
3. Our messages require the proper authorization. Specifically, any inbound message that starts with "/user/" will require ROLE_USER. Additional details on authorization can be found in WebSocket Authorization

WebSocket Authentication

WebSockets reuse the same authentication information that is found in the HTTP request when the WebSocket connection was made. This means that the `Principal` on the `HttpServletRequest` will be handed off to WebSockets. If you are using Spring Security, the `Principal` on the `HttpServletRequest` is overridden automatically.

More concretely, to ensure a user has authenticated to your WebSocket application, all that is necessary is to ensure that you setup Spring Security to authenticate your HTTP based web application.

WebSocket Authorization

Spring Security 4.0 has introduced authorization support for WebSockets through the Spring Messaging abstraction. To configure authorization using Java Configuration, simply extend the `AbstractSecurityWebSocketMessageBrokerConfigurer` and configure the `MessageSecurityMetadataSourceRegistry`. For example:

```
@Configuration
public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {

    @Override
    protected void configureInbound(MessageSecurityMetadataSourceRegistry messages) {
        messages
                .nullDestMatcher().authenticated() (1)
                .simpSubscribeDestMatchers("/user/queue/errors").permitAll() (2)
                .simpDestMatchers("/app/**").hasRole("USER") (3)
                .simpSubscribeDestMatchers("/user/**", "/topic/friends/*").hasRole("USER") (4)
                .simpTypeMatchers(MESSAGE, SUBSCRIBE).denyAll() (5)
                .anyMessage().denyAll(); (6)

    }
}
```

This will ensure that:

1. Any message without a destination (i.e. anything other that Message type of MESSAGE or SUBSCRIBE) will require the user to be authenticated
2. Anyone can subscribe to /user/queue/errors
3. Any message that has a destination starting with "/app/" will be require the user to have the role ROLE_USER
4. Any message that starts with "/user/" or "/topic/friends/" that is of type SUBSCRIBE will require ROLE_USER
5. Any other message of type MESSAGE or SUBSCRIBE is rejected. Due to 6 we do not need this step, but it illustrates how one can match on specific message types.
6. Any other Message is rejected. This is a good idea to ensure that you do not miss any messages.

Spring Security also provides XML Namespace support for securing WebSockets. A comparable XML based configuration looks like the following:

```
<websocket-message-broker>
  (1)
  <intercept-message type="CONNECT" access="permitAll" />
  <intercept-message type="UNSUBSCRIBE" access="permitAll" />
  <intercept-message type="DISCONNECT" access="permitAll" />
```

```
    <intercept-message pattern="/user/queue/errors" type="SUBSCRIBE" access="permitAll" /> (2)
    <intercept-message pattern="/app/**" access="hasRole('USER')" />      (3)

    (4)
    <intercept-message pattern="/user/**" access="hasRole('USER')" />
    <intercept-message pattern="/topic/friends/*" access="hasRole('USER')" />

    (5)
    <intercept-message type="MESSAGE" access="denyAll" />
    <intercept-message type="SUBSCRIBE" access="denyAll" />

    <intercept-message pattern="/**" access="denyAll" /> (6)
</websocket-message-broker>
```

This will ensure that:

1. Any message of type CONNECT, UNSUBSCRIBE, or DISCONNECT will require the user to be authenticated
2. Anyone can subscribe to /user/queue/errors
3. Any message that has a destination starting with "/app/" will be require the user to have the role ROLE_USER
4. Any message that starts with "/user/" or "/topic/friends/" that is of type SUBSCRIBE will require ROLE_USER
5. Any other message of type MESSAGE or SUBSCRIBE is rejected. Due to 6 we do not need this step, but it illustrates how one can match on specific message types.
6. Any other message with a destination is rejected. This is a good idea to ensure that you do not miss any messages.

WebSocket Authorization Notes

In order to properly secure your application it is important to understand Spring's WebSocket support.

WebSocket Authorization on Message Types

It is important to understand the distinction between SUBSCRIBE and MESSAGE types of messages and how it works within Spring.

Consider a chat application.

- The system can send notifications MESSAGE to all users through a destination of "/topic/system/notifications"
- Clients can receive notifications by SUBSCRIBE to the "/topic/system/notifications".

While we want clients to be able to SUBSCRIBE to "/topic/system/notifications", we do not want to enable them to send a MESSAGE to that destination. If we allowed sending a MESSAGE to "/topic/system/notifications", then clients could send a message directly to that endpoint and impersonate the system.

In general, it is common for applications to deny any MESSAGE sent to a message that starts with the[broker prefix](#) (i.e. "/topic/" or "/queue/").

WebSocket Authorization on Destinations

It is also is important to understand how destinations are transformed.

Consider a chat application.

- User's can send messages to a specific user by sending a message to the destination of "/app/chat".
- The application sees the message, ensures that the "from" attribute is specified as the current user (we cannot trust the client).
- The application then sends the message to the recipient using `SimpMessageSendingOperations.convertAndSendToUser("toUser", "/queue/messages", message)`.
- The message gets turned into the destination of "/queue/user/messages-<sessionid>"

With the application above, we want to allow our client to listen to "/user/queue" which is transformed into "/queue/user/messages-<sessionid>". However, we do not want the client to be able to listen to "/queue/*" because that would allow the client to see messages for every user.

In general, it is common for applications to deny any SUBSCRIBE sent to a message that starts with the[broker prefix](#) (i.e. "/topic/" or "/queue/"). Of course we may provide exceptions to account for things like

Outbound Messages

Spring contains a section titled Flow of Messages that describes how messages flow through the system. It is important to note that Spring Security only secures the `clientInboundChannel`. Spring Security does not attempt to secure the `clientOutboundChannel`.

The most important reason for this is performance. For every message that goes in, there are typically many more that go out. Instead of securing the outbound messages, we encourage securing the subscription to the endpoints.

Enforcing Same Origin Policy

It is important to emphasize that the browser does not enforce the Same Origin Policy for WebSocket connections. This is an extremely important consideration.

Why Same Origin?

Consider the following scenario. A user visits bank.com and authenticates to their account. The same user opens another tab in their browser and visits evil.com. The Same Origin Policy ensures that evil.com cannot read or write data to bank.com.

With WebSockets the Same Origin Policy does not apply. In fact, unless bank.com explicitly forbids it, evil.com can read and write data on behalf of the user. This means that anything the user can do over the websocket (i.e. transfer money), evil.com can do on that users behalf.

Since SockJS tries to emulate WebSockets it also bypasses the Same Origin Policy. This means developers need to explicitly protect their applications from external domains when using SockJS.

Spring WebSocket Allowed Origin

Fortunately, since Spring 4.1.5 Spring's WebSocket and SockJS support restricts access to the current domain. Spring Security adds an additional layer of protection to provide defence in depth.

Adding CSRF to Stomp Headers

By default Spring Security requires the CSRF token in any CONNECT message type. This ensures that only a site that has access to the CSRF token can connect. Since only the **Same Origin** can access the CSRF token, external domains are not allowed to make a connection.

Typically we need to include the CSRF token in an HTTP header or an HTTP parameter. However, SockJS does not allow for these options. Instead, we must include the token in the Stomp headers

Applications can obtain a CSRF token by accessing the request attribute named _csrf. For example, the following will allow accessing the `CsrfToken` in a JSP:

```
  var headerName = "${_csrf.headerName}";
var token = "${_csrf.token}";
```

If you are using static HTML, you can expose the `CsrfToken` on a REST endpoint. For example, the following would expose the `CsrfToken` on the URL /csrf

```
  @RestController
public class CsrfController {

    @RequestMapping("/csrf")
    public CsrfToken csrf(CsrfToken token) {
        return token;
    }
}
```

The javascript can make a REST call to the endpoint and use the response to populate the headerName and the token.

We can now include the token in our Stomp client. For example:

```
  ...
var headers = {};
headers[headerName] = token;
stompClient.connect(headers, function(frame) {
  ...

}
```

Disable CSRF within WebSockets

If you want to allow other domains to access your site, you can disable Spring Security's protection. For example, in Java Configuration you can use the following:

```
  @Configuration
public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {

    ...

    @Override
    protected boolean sameOriginDisabled() {
        return true;
    }
}
```

Working with SockJS

SockJS provides fallback transports to support older browsers. When using the fallback options we need to relax a few security constraints to allow SockJS to work with Spring Security.

SockJS & frame-options

SockJS may use an transport that leverages an iframe. By default Spring Security will deny the site from being framed to prevent Clickjacking attacks. To allow SockJS frame based transports to work, we need to configure Spring Security to allow the same origin to frame the content.

You can customize X-Frame-Options with the frame-options element. For example, the following will instruct Spring Security to use "X-Frame-Options: SAMEORIGIN" which allows iframes within the same domain:

```
  <http>
    <!-- ... -->

    <headers>
        <frame-options
          policy="SAMEORIGIN" />
    </headers>
</http>
```

Similarly, you can customize frame options to use the same origin within Java Configuration using the following:

```
  @EnableWebSecurity
public class WebSecurityConfig extends
   WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      // ...
      .headers()
        .frameOptions()
            .sameOrigin();
  }
}
```

SockJS & Relaxing CSRF

SockJS uses a POST on the CONNECT messages for any HTTP based transport. Typically we need to include the CSRF token in an HTTP header or an HTTP parameter. However, SockJS does not allow for these options. Instead, we must include the token in the Stomp headers as described in Adding CSRF to Stomp Headers.

It also means we need to relax our CSRF protection with the web layer. Specifically, we want to disable CSRF protection for our connect URLs. We do NOT want to disable CSRF protection for every URL. Otherwise our site will be vulnerable to CSRF attacks.

We can easily achieve this by providing a CSRF RequestMatcher. Our Java Configuration makes this extremely easy. For example, if our stomp endpoint is "/chat" we can disable CSRF protection for only URLs that start with "/chat/" using the following configuration:

```
  @Configuration
@EnableWebSecurity
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

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

        http
            .csrf()
                // ignore our stomp endpoints since they are protected using Stomp headers
```

```
                .ignoringAntMatchers("/chat/**")
                .and()
            .headers()
                // allow same origin to frame our site to support iframe SockJS
                .frameOptions().sameOrigin()
                .and()
            .authorizeRequests()

            ...
```

If we are using XML based configuration, we can use the csrf@request-matcher-ref. For example:

```
 <http ...>
   <csrf request-matcher-ref="csrfMatcher"/>

   <headers>
       <frame-options policy="SAMEORIGIN"/>
   </headers>

   ...
</http>

<b:bean id="csrfMatcher"
    class="AndRequestMatcher">
    <b:constructor-arg value="#{T(org.springframework.security.web.csrf.CsrfFilter).DEFAULT_CSRF_MATCHER}"/>
    <b:constructor-arg>
        <b:bean class="org.springframework.security.web.util.matcher.NegatedRequestMatcher">
          <b:bean class="org.springframework.security.web.util.matcher.AntPathRequestMatcher">
            <b:constructor-arg value="/chat/**"/>
          </b:bean>
        </b:bean>
    </b:constructor-arg>
</b:bean>
```

授权

在Spring Security的高级授权功能是其受欢迎的最有说服力的原因之一，无论您如何选择如何进行身份验证-是否使用提供的机制和提供程序的Spring Security，或整合与一个容器或其他non-Spring Security 认证机构-你会发现授权服务可以在您的应用程序中使用一个一致的和简单的方式.

在这一部分我们将探讨不同的` abstractsecurityinterceptor `实现,在第一部分介绍.然后，我们将继续探索如何通过使用域访问控制列表微调授权.

授权体系结构

授权

正如我们在<<技术授予权，技术综述 >>所看到，所有的`Authentication`实现存储的列表`GrantedAuthority`对象.这些代表已被授予主要的的当局. `GrantedAuthority`对象是由` authenticationManager 插入到`Authentication` 对象，然后读取`AccessDecisionManager`做出判断.

`GrantedAuthority`是一个只有一个方法的接口

```
String getAuthority();
```

这个方法允许`AccessDecisionManager` 来判断得到一个精确的`String`表示的` GrantedAuthority `.通过返回一个表示作为一个`String`,一个` GrantedAuthority `可以很容易的通过`AccessDecisionManager` 来`read`,如果一个` GrantedAuthority `不能精确地表示为一个`String`,` GrantedAuthority `将会被认为是"complex"和`getAuthority()` 必须返回为`null`.

"complex" `GrantedAuthority`的一个将一个应用于不同客户帐户号码的操作和权限阈值的列表的实现例子.代表这个复杂的` GrantedAuthority `作为`String`将是相当困难的，作为一个结果，` getauthority() `方法应该返回`null`.这将对任何` accessDecisionManager `表明它需要明确的支持` GrantedAuthority `实施以了解其内容.

Spring Security包括一个具体的` GrantedAuthority `实施，` grantedauthorityimpl `.这允许用户指定的任何`String`转换成一种` GrantedAuthority `.所有的` AuthenticationProvider `的包括与安全架构使用` grantedauthorityimpl`填充`Authentication`对象.

Pre-Invocation处理

正如我们在<<安全对象、技术综述>>章节中也看到过的,Spring Security,提供拦截控制访问安全对象如方法调用或Web请求。是否允许进行调用前调用的决定是由`AccessDecisionManager `作出判断.

访问决策管理器

` accessDecisionManager 被 abstractsecurityinterceptor 和负责制定最终的访问控制决策.`AccessDecisionManager 接口包含三种方法:

```
 void decide(Authentication authentication, Object secureObject,
Collection<ConfigAttribute> attrs) throws AccessDeniedException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

`AccessDecisionManager 的 decide 方法传递了它所需要的所有相关信息，以作出授权决策.尤其,通过安全的“对象”，使这些参数包含在实际的安全对象调用中进行检查.例如,让我们假设安全对象是一个 MethodInvocation `资料，这将是很容易实现 `MethodInvocation 对于任何 Customer 论点.然后执行某种安全逻辑判断、来确保 AccessDecisionManager `主允许对客户操作.如果访问被拒绝并抛出`AccessDeniedException 我们的预期就实现了.

如果` accessDecisionManager 可以处理通过 configattribute ,`supports(ConfigAttribute) 方法由 AbstractSecurityInterceptor 在决定启动时候命名. supports(Class) 方法被安全拦截器实现，确保配置` accessDecisionManager `支持类型的安全对象的被拦截.

Voting-Based访问决策管理器实现

同时，用户可以实现自己的` AccessDecisionManager`判断、控制授权的所有方面,Spring Security包括几个基于投票的` accessDecisionManager `实现. Voting Decision Manager说明相关类.

Figure 2. Voting Decision Manager

使用这种方法，一系列的` accessdecisionvoter 实现调查授权决策. accessDecisionManager 然后决定是否抛出 accessdeniedexception `基于其选票的评估。

`AccessDecisionVoter 接口包含三种方法:

```
 int vote(Authentication authentication, Object object, Collection<ConfigAttribute> attrs);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

具体返回一个 int ，可能反映在` accessdecisionvoter 的静态字段 access_abstain , access_denied 和 access_granted .如果对授权决策没有意见一个投票的实施将返回 access_abstain . 如果真的有一个观点，它必须返回 access_denied 或 access_granted `.

这里三个具体的 AccessDecisionManager 提供了符合投票的 Spring Security,` consensusbased 实施将授予或拒绝基于非共识的弃权票的访问.提供的属性来控制在一个平等的投票活动的行为，或者如果所有的选票都弃权. affirmativebased 实施将会授权访问如果你个或者更多的`ACCESS_GRANTED 投票被收到（即使投票被忽略，也至少有一个投票）.像“ConsensusBased”实现,有一个参数,控制行为如果所有选民弃权. UnanimousBased 的提供者预计一致 ACCESS_GRANTED 的选票以授予访问权限,忽略了自制。它会拒绝访问如果有任何“ACCESS_DENIED”投票.像其他实现,有一个参数,控制行为如果所有选民弃权.

可以实现自定义` accessDecisionManager 计算选票不同。例如，从一个特定的 accessdecisionvoter `可能会得到额外的加权投票，而否认从一个特定的选民投票可能有否决权的影响.

角色选民

最常用的由Spring Security提供的 AccessDecisionVoter 是简单的 RoleVoter ,如果用户已被分配该角色,将配置属性视为简单的角色名称和投票授予访问权限.

如果 ConfigAttribute 带着前缀 ROLE_ 开始将会进行投票，它将投票授予访问权限，如果有` GrantedAuthority 它返回一个 `String 表示(通过` getauthority() 方法)恰好等于一个或多个从前缀 role_ 的 configattributes .如果没有从 role_ 精确匹配任何 configattribute , rolevoter 会投票拒绝访问。如果 role_ 没有开始 configattribute `，选民将投弃权票。

经过身份验证的选民

另一个选民,我们从 AuthenticatedVoter 看到，可以用来区分匿名，fully-authenticated,记得我通过身份验证的用户，许多网站允许某些有限的访问在记得我认证,但是需要用户确认他们的身份登录的完全访问.当我们使用属性 IS_AUTHENTICATED_ANONYMOUSLY授予匿名访问,这个属性是由“AuthenticatedVoter”进行处理。有关更多信息,请参见这个类的Javadoc.

定制的选民

显然,您还可以实现自定义“AccessDecisionVoter”,你可以把任何你想要访问控制逻辑.这可能是特定于应用程序(业务逻辑相关)

或可能实现一些安全管理逻辑，例如你会发现 blog article 在Spring web网站描述如何使用实时选民拒绝访问用户的账户被暂停.

调用处理

而“AccessDecisionManager”是由“AbstractSecurityInterceptor”在继续之前调用安全对象调用.某些应用程序需要一种方法修改对象的实际安全返回的对象调用.同时您可以很容易地实现自己的AOP实现这一担忧, Spring Security提供了一个方便的钩,几个集成ACL的功能的具体实现.

After Invocation Implementation说明了Spring Security的`AfterInvocationManager`以及这个具体实现.

Figure 3. After Invocation Implementation

像Spring Security的其他部分，`AfterInvocationManager`有一个具体的实现，`AfterInvocationProviderManager`调查`AfterInvocationProvider`的表，每一个`AfterInvocationProvider`允许修改返回对象或抛出`AccessDeniedException`.的确多个提供者可以修改对象,如之前的供应商的结果传递给下一个列表中.

请注意,如果您正在使用`AfterInvocationManager`,你仍然需要配置属性,使`MethodSecurityInterceptor`的`AccessDecisionManager`允许一个操作.如果您使用的是典型的Spring Security包括“AccessDecisionManager”实现,没有配置属性定义为一个特定的安全方法调用将导致每个“AccessDecisionVoter”投弃权票,相反,如果`AccessDecisionManager`性能 “allow IfAllAbstainDecisions”是`false`就会抛出一个`AccessDeniedException`，你可以通过设置“allowIfAllAbstainDecisions”避免这种潜在的问题改变为`true`（虽然这是一般不推荐），或者（ii）只是确保至少有一个配置属性,一个“AccessDecisionVoter”将投票授权访问，后者(推荐)的方法通常是通过“ROLE_USER”或“ROLE_AUTHENTICATED”配置属性.

层次的角色

它是一个共同的要求，一个特定的应用程序中的角色应该自动"include"其他角色,例如，在一个应用程序中有一个"admin"和"user"的角色的概念,你可能希望一个管理员能够尽一切正常的用户可以。要做到这一点，你可以确保所有的管理用户也被分配到"user"的角色.另外，您可以修改每一个访问约束，这需要"user"的角色，还包括"admin"的作用.这可能会变得相当复杂，如果你有很多不同的角色在你的应用程序.

使用角色层次结构允许您配置哪些角色（或主管）应包括其他角色（或主管部门）.一个额外的of Spring Security’s RoleVoter的版本，`RoleHierarchyVoter`配置了一个` rolehierarchy `，从它获得所有的"reachable authorities"，用户被分配。一个典型的配置可能看起来像这样：

```
  <bean id="roleVoter" class="org.springframework.security.access.vote.RoleHierarchyVoter">
 <constructor-arg ref="roleHierarchy" />
</bean>
<bean id="roleHierarchy"
  class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">
 <property name="hierarchy">
  <value>
   ROLE_ADMIN > ROLE_STAFF
   ROLE_STAFF > ROLE_USER
   ROLE_USER > ROLE_GUEST
  </value>
 </property>
</bean>
```

在这里，我们有四个角色在一个层次结构`ROLE_ADMIN ⇒ ROLE_STAFF ⇒ ROLE_USER ⇒ ROLE_GUEST`.通过身份验证的用户`ROLE_ADMIN`，要表现得好像他们所有的四种角色在安全约束的评价与判断,要表现得好像他们所有的四种角色在安全约束的评价与判断,`AccessDecisionManager`配置上述`RoleHierarchyVoter`.> 符号可以被认为是意义的"includes".

角色层次结构提供了一个方便的方法简化了您的应用程序的访问控制配置数据和/或减少您需要分配给用户的权限数.对于更复杂的要求，您可能希望定义您的应用程序所需的特定访问权限和被分配给用户的角色之间的逻辑映射，在加载用户信息时将两者翻译成两者之间的关系.

安全对象的实现

AOP联盟(方法调用)安全拦截器

`Spring Security 2.0之前,确保`MethodInvocation`资料需要相当多的配置.现在方法安全的推荐方法是使用< <ns-method-security,namespace configuration>>.这种方法的安全基础设施beans是为您自动配置的，所以您不需要知道实现类的情况。我们将提供一个在这里涉及的类的快速概述。`

方法在执行安全使用` methodsecurityinterceptor `,这是一个固定的`MethodInvocation`.根据配置的方法，一个拦截可能是特定的一个单一的bean或多个beans之间的共享.拦截器使用` methodsecuritymetadatasource `实例获取配置属性，适用于一

个特定的方法调用.`MapBasedMethodSecurityMetadataSource`用于存储配置属性的键控的方法名称（可以使用通配符），将在内部使用时，这些属性定义在应用程序的上下文中使用的`<intercept-methods>`拦截或`<protect-point>`元素。

当然你可以使用一个Spring AOP的代理机制配置一个` methodsecurityiterceptor `直接应用程序上下文中:

```
  <bean id="bankManagerSecurity" class=
 "org.springframework.security.access.intercept.aopalliance.MethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
 <sec:method-security-metadata-source>
 <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
 <sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE_TELLER,ROLE_SUPERVISOR"/>
 </sec:method-security-metadata-source>
</property>
</bean>
```

AspectJ(连接点)安全拦截器

AspectJ的安全拦截器是AOP联盟安全拦截器在上一节讨论非常相似。事实上，我们将只讨论在这一部分的差异.

AspectJ拦截器被命名为`AspectJSecurityInterceptor`.不像AOP联盟安全拦截器，它依赖于Spring应用程序上下文编织的安全拦截器通过代理,`AspectJSecurityInterceptor`是基于AspectJ编译器.不会罕见的在同一个程序中使用的两种安全拦截器,与`AspectJSecurityInterceptor`用于域对象实例安全,AOP联盟` methodsecurityinterceptor `用于服务层安全.

让我们首先考虑的是如何`AspectJSecurityInterceptor`配置在Spring应用程序上下文：

```
  <bean id="bankManagerSecurity" class=
 "org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
 <sec:method-security-metadata-source>
 <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
 <sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE_TELLER,ROLE_SUPERVISOR"/>
 </sec:method-security-metadata-source>
</property>
</bean>
```

正如你所看到的，除了类名称，`AspectJSecurityInterceptor`是完全一样的AOP联盟安全拦截器.事实上，两个拦截器可以共享相同的` securitymetadatasource `,作为` securitymetadatasource `作品 `java.lang.reflect.Method`而不是AOP库类。当然,你访问的决定获得有关特定AOP库调用（即` MethodInvocation `或`JoinPoint`），这样可以使访问的决定时，考虑的范围之外的标准（如方法的参数）.

下次你需要定义一个AspectJ `aspect`.例如:

```
  package org.springframework.security.samples.aspectj;

import org.springframework.security.access.intercept.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.access.intercept.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

 private AspectJSecurityInterceptor securityInterceptor;

 pointcut domainObjectInstanceExecution(): target(PersistableEntity)
  && execution(public * *(..)) && !within(DomainObjectInstanceSecurityAspect);

 Object around(): domainObjectInstanceExecution() {
  if (this.securityInterceptor == null) {
   return proceed();
  }

  AspectJCallback callback = new AspectJCallback() {
   public Object proceedWithObject() {
    return proceed();
   }
  };

  return this.securityInterceptor.invoke(thisJoinPoint, callback);
 }

 public AspectJSecurityInterceptor getSecurityInterceptor() {
  return securityInterceptor;
 }

 public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {
  this.securityInterceptor = securityInterceptor;
 }
```

```
 public void afterPropertiesSet() throws Exception {
  if (this.securityInterceptor == null)
   throw new IllegalArgumentException("securityInterceptor required");
  }
 }
}
```

在上面的例子中，安全拦截器将被应用到每一个实例` persistableentity ，这是一个抽象类没有显示（你可以使用任何其他类或`pointcut表达你喜欢）.对于那些好奇的人，` aspectjcallback 是因为 proceed()；声明具有特殊的意义只有在 around() 体。当它想要的目标继续,`AspectJSecurityInterceptor 调用这个匿名` aspectjcallback `类.

你需要配置Spring 负载方面和连接`AspectJSecurityInterceptor`。一个实现这一的bean声明如下所示：

```
  <bean id="domainObjectInstanceSecurityAspect"
 class="security.samples.aspectj.DomainObjectInstanceSecurityAspect"
 factory-method="aspectOf">
<property name="securityInterceptor" ref="bankManagerSecurity"/>
</bean>
```

是这么回事！现在你可以从你的应用程序中的任何地方创建你的豆子，使用任何你认为合适的方式(eg `new Person();`)他们会拥有安全拦截器的应用

表达式的访问控制

Spring Security 3.0介绍了使用的能力，作为一个授权机制，除了简单的使用配置属性和访问决策的选民，以前见过的使用弹簧的表达。基于表达式的访问控制是建立在相同的架构，但允许复杂的布尔逻辑被封装在一个单一的表达.

概述

Spring Security 使用Spring EL来支持，你应该看看如何，如果你有兴趣在更深入了解主题。表达式是用"root object" 评估的，作为评价上下文的一部分。Spring Security使用特定的类用于Web和方法安全作为根对象，以提供内置表达式和访问当前主体的值等.

常见的内置的表达式

表达根对象的基类是` securityexpressionroot `.这提供了一些常见的表达式，可应用在网络和方法安全性.

Table 3. Common built-in expressions

表达	描述
`hasRole([role])`	如果当前主体具有指定的角色，则返回`true`.默认情况下，如果提供的角色不从'ROLE_'这里提供,这将增加。这可以通过修改` defaultroleprefix 在 defaultwebsecurityexpressionhandler `配置.
`hasAnyRole([role1,role2])`	如果当前的主体有任何提供的角色（给定的作为一个逗号分隔的字符串列表）的话，返回`true`,默认情况下，如果提供的角色不从'ROLE_',这将增加。这可以通过修改` defaultroleprefix 在 defaultwebsecurityexpressionhandler `定制.
`hasAuthority([authority])`	如果当前的主体具有指定的权限，则返回`true`.
`hasAnyAuthority([authority1,authority2])`	如果当前的主体有任何提供的角色（给定的作为一个逗号分隔的字符串列表）的话，返回`true`.
`principal`	允许直接访问表示当前用户的主对象
`authentication`	允许直接访问从`SecurityContext`得出当前的`Authentication`对象
`permitAll`	总是评估为`true`
`denyAll`	总是评估为`false`
`isAnonymous()`	如果当前的主体是一个匿名用户，则返回`true`.
`isRememberMe()`	如果当前的主体是一个匿名用户，则返回`true`
`isAuthenticated()`	如果用户不是匿名的，则返回`true`

表达	描述
isFullyAuthenticated()	如果用户不是一个匿名的或是一个记住我的用户返回true
hasPermission(Object target, Object permission)	如果用户已访问给定权限的提供的目标，则返回 `true` ，例如 `hasPermission(domainObject, 'read')`
hasPermission(Object targetId, String targetType, Object permission)	如果用户已访问给定权限的提供的目标，则返回 `true` ,例如 `hasPermission(1, 'com.example.domain.Message', 'read')`

Web Security Expressions

使用表达式来保护个人网址,首先需要设置“use-expressions”属性的< http >为`true`.Spring Security预期的“访问”属性的< intercept-url >元素包含Spring EL表达式。一个布尔表达式应该评估,定义是否应该允许访问. 例如:

```
 <http>
<intercept-url pattern="/admin*"
 access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>
...
</http>
```

这里我们定义了应用程序的“admin”区域(定义的URL模式)只能提供给用户授予机关“admin”,其IP地址匹配本地子网.在前一节中我们已经看到内置hasRole表达式。表达“hasIpAddress”是另一个内置的表达式是特定于网络安全.这由`WebSecurityExpressionRoot`下定义，一个实例时用作表达式根对象评估web访问表达式。这个对象也直接暴露的HttpServletRequest对象的名字“请求”,这样你就可以直接调用请求在一个表达式。如果正在使用表情,“WebExpressionVoter”将被添加到“AccessDecisionManager”所使用的名称空间. 如果你不使用名称空间和想使用表情,你必须添加一个配置.

在网络安全bean表达式

如果你想扩展表达式可用,您可以很容易地操作任何你暴露的Spring Bean。例如,assumming你有一个Bean的名称“webSecurity”包含以下方法

```
public class WebSecurity {
 public boolean check(Authentication authentication, HttpServletRequest request) {
   ...
 }
}
```

你可以参考使用方法:

```
 <http>
<intercept-url pattern="/user/**"
 access="@webSecurity.check(authentication,request)"/>
...
</http>
```

或在Java配置

```
http
 .authorizeRequests()
   .antMatchers("/user/**").access("@webSecurity.check(authentication,request)")
   ...
```

Path Variables in Web Security Expressions

有时在一个URL它很好能够参考路径变量. 例如,考虑一个RESTful应用程序从URL路径的格式查找用户id`/user/{userId}`.

你可以很容易地将参考路径变量的模式.例如,如果你有一个Bean的名称“webSecurity”包含以下:

```
public class WebSecurity {
 public boolean checkUserId(Authentication authentication, int id) {
   ...
 }
}
```

你可以参考使用方法:

```
 <http>
<intercept-url pattern="/user/{userId}/**"
 access="@webSecurity.checkUserId(authentication,#userId)"/>
...
</http>
```

或在Java配置

```
http
 .authorizeRequests()
```

```
    .antMatchers("/user/{userId}/**").access("@webSecurity.checkUserId(authentication,#userId)")
    ...
```

在这两个配置相匹配的url将通过path变量(和)转换成checkUserId方法.

例如,如果这个URLs是 `/user/123/resource`,id是`123`.

Method Security Expressions

方法安全性是一个更复杂的比一个简单的规则允许或拒绝，Spring Security 3.0介绍了一些新的注释,以便全面支持表达式的使用.

@Pre and @Post Annotations

有四个属性注释支持表达式允许pre和post-invocation授权检查并提交支持过滤收集参数或返回值.他们是`@PreAuthorize`, `@PreFilter`, `@PostAuthorize` and `@PostFilter`. 它们的使用是通过“global-method-security”名称空间的元素:

```
 <global-method-security pre-post-annotations="enabled"/>
```

使用@PreAuthorize和@PostAuthorize访问控制,最明显的是有用的注释是“@PreAuthorize”决定是否可以被调用方法。例如(从"Contacts"示例应用程序)

```
 @PreAuthorize("hasRole('USER')")
public void create(Contact contact);
```

这意味着用户与角色"ROLE_USER"才会允许访问.显然同样的事情可以很容易地通过使用传统的配置和一个简单的配置属性所需的角色:

```
 @PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

这里我们使用一个方法参数的表达式来决定当前用户是否有"admin"允许给定的接触。内置的`hasPermission()`表达式是通过应用程序上下文链接到Spring Security ACL模块,see below,你可以访问任何变量名称作为方法参数的表达式.

有很多方式Spring Security可以解决方法参数。Spring Security使用DefaultSecurityParameterNameDiscoverer发现参数名称。默认情况下,下列选项尝试方法作为一个整体.

- 如果Spring Security的@P注释存在一个参数的方法,将使用价值。这是使用JDK JDK 8之前有用的接口,编译不包含任何有关参数名称的信息。例如:

  ```
  import org.springframework.security.access.method.P;

  ...

  @PreAuthorize("#c.name == authentication.name")
  public void doSomething(@P("c") Contact contact);
  ```

  在幕后使用使用“AnnotationParameterNameDiscoverer”可实现自定义支持value属性指定的任何注释.

- 如果Spring Data'的@Param注释存在至少一个参数的方法,将使用价值。这是使用JDK JDK 8之前有用的接口,编译不包含任何有关参数名称的信息。例如:

  ```
  import org.springframework.data.repository.query.Param;

  ...

  @PreAuthorize("#n == authentication.name")
  Contact findContactByName(@Param("n") String name);
  ```

在幕后使用使用“AnnotationParameterNameDiscoverer”可实现自定义支持value属性指定的任何注释.

- 如果JDK 8是用来编译源参数的参数和使用Spring 4+,那么标准JDK反射API用于发现参数名称。这包含两类和接口工作.
- 最后,如果代码编译与调试符号,参数名称将被发现使用调试符号。这不会为接口工作,因为他们没有调试信息参数名称。为接口,必须使用注释或JDK 8的方法.

任何Spring-EL功能可在表达,所以你也可以访问属性参数。举个例子,如果你想要一个特定方法只允许一个用户访问的用户名匹配的接触,你可以写

```
 @PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

我们访问另一个内置的表情, `authentication`,也就是 `Authentication` 存储在安全上下文。您也可以直接访问它的 `principal` 属性,使用表达式 `principal`。值往往会是一个“UserDetails”实例,所以你可能会使用一个表达式 `principal.username` 或“principal.enabled”.

一般,您可能希望执行访问控制检查方法调用之后。这可以通过使用@PostAuthorize注释.访问一个方法的返回值,使用内置的名字“returnObject”的表示

过滤用@PreFilter and @PostFilter

正如你可能已经知道,Spring Security支持集合和数组的过滤,这可以通过使用表达式。这是最常见的一个方法的返回值上执行。例如

```
  @PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
public List<Contact> getAll();
```

当用`@PostFilter`注释,Spring Security遍历返回的集合并删除任何元素提供的表达式是错误的。“filterObject”指的是当前的对象的集合。你还可以过滤方法调用之前,使用“@PreFilter”,尽管这是一个不太常见的需求。语法是相同的,但是如果有一个以上的论证它是一个集合类型然后你必须选择一个的名字使用“filterTarget”属性的注释.

注意,过滤显然是不能代替调优数据检索查询。如果你是过滤大收藏和删除的条目,那么这可能是低效的.

内置的表达式

有一些内置的表达式具体方法安全,我们已经看到在上面使用。`filterTarget`和`returnValue`值是很简单,但使用`hasPermission()`的表达式权证更仔细的观察.

许可评估者接口

`hasPermission()`的表达式是委托给一个实例的`PermissionEvaluator`.它旨在表达系统和Spring Security的ACL系统之间的桥梁,允许您指定授权约束域对象,基于抽象的权限.没有显式的依赖ACL模块,所以你可以互换,如果需要另一个实现。接口有两个方法:

```
  boolean hasPermission(Authentication authentication, Object targetDomainObject,
      Object permission);

boolean hasPermission(Authentication authentication, Serializable targetId,
      String targetType, Object permission);
```

这直接映射到可用的版本的表情,除了那第一个参数(`Authentication`的对象)是不提供的。首先是在域对象的情况下,使用的访问控制,已经加载。然后表达式将返回true,如果当前用户拥有该对象的批准。第二个版本是用于装载情况下,对象不是,但是它的标识符。域对象的抽象的"type"说明符也是必需的,允许加载正确的ACL权限。这历来是对象的Java类,但是这不是必须的，只要符合权限如何加载就可以.

使用“hasPermission()的表情,必须在您的应用程序上下文配置一个PermissionEvaluator”.这看起来像这样:

```
  <security:global-method-security pre-post-annotations="enabled">
<security:expression-handler ref="expressionHandler"/>
</security:global-method-security>

<bean id="expressionHandler" class=
"org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
 <property name="permissionEvaluator" ref="myPermissionEvaluator"/>
</bean>
```

在“myPermissionEvaluator”是实现“PermissionEvaluator”bean。通常这将从ACL实现模块叫做“AclPermissionEvaluator”。见"Contacts"示例应用程序配置更多的细节.

方法安全性元注释

你可以使用元数据注释方法安全性提高代码的可读性。

如果你发现你是在代码库重复相同的复杂表达式。尤其方便

例如,考虑以下几点

```
  @PreAuthorize("#contact.name == authentication.name")
```

Instead of repeating this everywhere, we can create a meta annotation that can be used instead.

```
  @Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
public @interface ContactPermission {}
```

元注释可用于任何Spring安全方法的安全注释.

为了保持符合jsr - 250规范的注释不支持元注释.

额外的话题

在本部分中,我们将介绍功能,这需要一个前一章节的知识以及一些更高级的和不常使用的功能框架。

域对象的安全(acl)

概述

复杂的应用需要的往往不是简单的得到在web请求或方法调用级别来定义访问权限。而是，安全决议需要包括谁（认证），其中（的MethodInvocation）和什么（一些领域对象）。换句话说，授权决策还需要考虑一个方法调用的实际域对象实例的主题。

想象你正在设计申请宠物诊所。将会有两个主要基于spring的应用程序的用户组:宠物诊所的工作人员,以及宠物诊所的客户。员工将获得的所有数据,而你的客户只能看到自己的客户记录。使它更有趣,你的客户可以让其他用户看到他们的客户记录,例如"puppy preschool"的导师或当地的"Pony Club"的总统。使用Spring安全为基础,有几种方法可以使用:

- 写下你的业务方法执行的安全。你可以咨询一个集合内的 `Customer` 域对象实例来确定哪些用户可以访问。通过使用 `SecurityContextHolder.getContext().getAuthentication()` ,你将能够访问 `Authentication` 的对象.
- 写一个 `AccessDecisionVoter` 执行的安全 `GrantedAuthority[]` 存储在 `Authentication` 对象。这将意味着你的 `AuthenticationManager` 需要填充的 `Authentication` 自定义 `GrantedAuthority[]` 代表每一个 `Customer` 访问域对象实例.
- 写一个 `AccessDecisionVoter` 执行安全和直接打开目标客户的域对象。这将意味着你的选民需要访问一个DAO,允许它来检索 `Customer` 的对象。它将访问的 `Customer` 对象的集合的批准用户和做出适当的决定.

每一个这些方法是完全合法的。然而,第一对授权检查你的业务代码。主要问题包括增强的困难的单元测试和重用会更困难的 `Customer` 授权逻辑。获得“GrantedAuthority[]的从 `Authentication` 对象也很好,但不会扩展到大量的 `Customer` 。如果用户可以访问5000`Customer`(不可能在这种情况下,但试想一下,如果它是一个受欢迎的兽医大小马俱乐部!)所需的内存消耗和时间来构造对象将是不受欢迎的 `Authentication` 。最后的方法,直接从外部代码打开 `Customer` ,可能是最好的三个。实现关注点分离,不滥用内存或CPU周期,但它仍然是低效的,“AccessDecisionVoter”和最终的商业方法本身将执行调用DAO负责检索 `AccessDecisionVoter` 对象。两个访问每个方法调用显然是不可取的。此外,每个方法列出你需要编写自己的访问控制列表(ACL)从头持久性和业务逻辑.

幸运的是,还有另一个选择,我们下面会讲到.

关键概念

Spring Security的ACL服务运送 `spring-security-acl-xxx.jar` .您需要将这个JAR添加到类路径中使用Spring安全域对象实例的安全功能.

Spring Security的域对象实例的安全能力中心的概念一个访问控制列表(ACL)。每个域对象实例系统中有自己的ACL,和ACL记录的细节,谁能和不能使用域对象。有鉴于此,Spring Security提供三个主要ACL-related功能应用程序:

- 一种有效地检索ACL条目你所有的域对象(和修改ACL).
- 确保给定的方式主要是允许的工作与你的对象,之前被称为方法.
- 确保给定的方式主要是允许使用对象(或者他们返回),后被称为方法.

第一个要点,Spring Security ACL的的一个主要功能模块提供了一个高性能的方式检索ACL。这个ACL库能力是极其重要的,因为每一个域对象实例系统中可能有多个访问控制条目,并且每个ACL可能继承其他树形结构中的ACL(这是开箱即用的支持由Spring Security,非常常用)。Spring Security的ACL能力都是被仔细设计以提供高性能检索ACL,加上可插入的缓存,deadlock-minimizing数据库更新、独立于ORM框架(我们直接使用JDBC),适当的封装,和透明的数据库更新.

给定数据库ACL的操作模块的核心,让我们探索使用四个主要表的默认实现。下面表的大小在一个典型的Spring Security ACL部署,最后列出的表行:

- ACL_SID允许我们唯一地标识系统中的任何本金或权威("SID" 代表"security identity")。唯一列ID、文本表示的SID,国旗表明是否文本表示是指主体名称或 `GrantedAuthority` .因此,有一个为每个独特的主体或 `GrantedAuthority` 行。的上下文中使用时获得许可,SID通常称为"recipient".
- ACL_CLASS允许我们唯一地标识系统中任何域对象类。只列ID和Java类名。因此,有一行我们希望每一个独特的类存储ACL权限.
- ACL_OBJECT_IDENTITY门店信息系统中每个独特的域对象实例。列包含ID,ACL_CLASS表的外键,所以我们知道唯一标识符ACL_CLASS实例我们提供信息,父,ACL_SID表的外键表示域对象实例的所有者,以及我们是否允许ACL条目继承任何父ACL。我们已经为每个域对象实例一行我们存储ACL权限.

*最后,ACL_ENTRY存储个人权限分配给每个收件人。ACL_OBJECT_IDENTITY列包括一个外键,收件人ACL_SID(外键),是否我们将审计,整数位屏蔽代表实际的权限被授予或拒绝。我们为每个收件人接收一行允许使用一个域对象.

如最后一段中所述,ACL系统使用整数位屏蔽。别担心,你不需要知道的细微之处,转向使用ACL系统,但是我想说的是,我们有32位我们可以打开或关闭。每一个位代表一个许可,并默认权限阅读(0),写(1),创建(2)、删除(第3位)和管理(4)。很容易实现自己的“许可”实例如果你希望使用其他权限,和其余的ACL框架将没有知识的扩展.

重要的是要理解,域对象的数量在系统完全没有影响我们选择使用整数位屏蔽。虽然你有32位用于权限,你可以有数十亿的域对象实例(这将意味着数十亿行ACL_OBJECT_IDENTITY而且很可能ACL_ENTRY)。我们这一点,因为我们发现有时人们错误地认为他们需要一点对于每一个可能的域对象,这并非如此.

现在我们已经提供了一个基本的概述ACL系统做什么,看起来在一个表结构,让我们探索的关键接口。关键接口:

- `Acl`:每一个域对象都有且只有一个Acl的对象,内部持有AccessControlEntry的年代以及知道的Acl的所有者。Acl不直接引用到域对象,而是一个`ObjectIdentity`.Acl的存储在ACL_OBJECT_IDENTITY表.
- `AccessControlEntry`:一个Acl拥有多个“AccessControlEntry”年代,通常缩写为ace框架。每个ACE是指一个特定的元组的“许可”,“Sid”和“Acl”。ACE还可以授予或non-granting和包含审计设置。ACE ACL_ENTRY表中存储。
- `Permission`:权限代表一个特定不变的位元遮罩,为钻头提供了便利的函数屏蔽和输出信息。上面给出的基本权限(字节0到4)中包含“BasePermission”类。
- `Sid`:ACL模块需要指校长和“GrantedAuthority[]的年代。提供了一个间接层的Sid的界面,这是一种“安全标识”的缩写。常见的类包括“PrincipalSid”(代表校长在一个“身份验证”对象)和“GrantedAuthoritySid”。安全身份信息存储在ACL_SID表。
- `ObjectIdentity`:每个域对象内部ACL表示模块由一个“ObjectIdentity”。默认实现叫做“ObjectIdentityImpl”。
- `AclService`:检索Acl的适用于一个给定的“ObjectIdentity”。包括实现(JdbcAclService),检索操作委托给一个“LookupStrategy”。“LookupStrategy”为检索ACL信息提供了一个高度优化的策略,使用“(BasicLookupStrategy”批处理检索)和支持自定义实现利用物化视图,分级查询和performance-centric相似,non-ANSI SQL功能。
- `MutableAclService`:允许提出了修改Acl的持久性。这并不是最重要的如果你不希望使用这个接口。

请注意我们的开箱即用的AclService和相关数据库类都使用ANSI SQL.这是主要的数据库.在写这篇文章的时候,系统已经成功测试了使用超音速SQL,PostgreSQL,Microsoft SQL Server和Oracle.

两个样本船与演示Spring Security ACL模块。第一个是联系人样本,另一个是文档管理系统(DMS)样本。我们建议采取一看这些例子.

开始

要开始使用Spring Security ACL的功能,你需要你的ACL信息存储在某个地方。这需要实例化的`DataSource`使用Spring。`DataSource`然后注入`JdbcMutableAclService`和`BasicLookupStrategy`实例。后者提供高性能的ACL检索功能,和前提供增变基因功能。指的一个样本船与Spring Security配置的一个示例。您还需要用四个ACL-specific填充数据库表中列出的最后一部分(参见ACL样本的适当的SQL语句).

一旦您创建了所需的模式和实例化`JdbcMutableAclService`,接下来将需要确保您的域模型支持互操作性的Spring Security ACL包。希望`ObjectIdentityImpl`将是足够的,因为它提供了大量的方法可以使用它。大部分人都有包含`public Serializable getId()`的方法。如果返回类型是长,或兼容长(例如int),你会发现你不需要提供进一步的考虑`ObjectIdentity`问题。许多地方的ACL模块依赖长标识符。如果你不使用长(或int,字节等),有一个非常好的机会你需要重新实现的类。我们不打算支持非long标识符在Spring Security的ACL模块,多头已经兼容所有数据库序列,最常见的标识符的数据类型和长度足够容纳所有常见的使用场景。

以下代码片段显示了如何创建一个Acl,或修改现有`Acl`:

```
  // Prepare the information we'd like in our access control entry (ACE)
ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new Long(44));
Sid sid = new PrincipalSid("Samantha");
Permission p = BasePermission.ADMINISTRATION;

// Create or update the relevant ACL
MutableAcl acl = null;
try {
acl = (MutableAcl) aclService.readAclById(oi);
} catch (NotFoundException nfe) {
acl = aclService.createAcl(oi);
}

// Now grant some permissions via an access control entry (ACE)
acl.insertAce(acl.getEntries().length, p, sid, true);
aclService.updateAcl(acl);
```

在上面的示例中,我们检索ACL的44号"Foo" 域对象标识符。我们添加一个ACE,然后名叫"Samantha"能"administer"的对象。代码片段相对明显,除了insertAce方法。insertAce方法的第一个参数是确定在什么位置Acl新条目将被插入。在上面的示例中,我们只是把新的现有ACE。最后一个参数是一个布尔值指示是否允许或拒绝。大部分时间它将授予(真实的),但如果是否认(假),实际上是被屏蔽的权限.

Spring Security并不提供任何特殊的集成自动创建、更新或删除acl DAO或存储库操作的一部分。相反,您需要编写代码如上图所示为你单独的域对象.值得考虑使用AOP在服务层与服务层自动把ACL信息操作.我们发现这在过去的一个相当有效的方法.

一旦你使用上述技术将一些ACL信息存储在数据库中,下一步是实际使用ACL信息作为授权决策逻辑的一部分。这里有许多选择。您可以编写自己的 `AccessDecisionVoter` 或 `AfterInvocationProvider` 分别触发一个方法调用之前或之后。这些课程将使用 `AclService` 来检索相关的ACL,然后调用的ACL。i`Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode) 决定是否授予或拒绝许可。或者,您可以使用我们的`AclEntryVoter`, `AclEntryAfterInvocationProvider` 或 `AclEntryAfterInvocationCollectionFilteringProvider` 类。所有这些类提供一个declarative-based方法评估ACL信息在运行时,释放你需要编写任何代码。请参阅示例应用程序来学习如何使用这些类。

Pre-Authentication场景

在有些情况下,您希望使用Spring安全授权,但是用户已经被一些外部系统可靠地验证之前访问应用程序。我们称这种情况为 "pre-authenticated"场景。例子包括 X.509,Siteminder和身份验证的Java EE容器的应用程序正在运行。当使用pre-authentication,Spring Security

*识别用户的请求

*为用户获得当局

细节将取决于外部身份验证机制。用户可能会被他们的证书信息的X.509,或通过一个HTTP请求头Siteminder的情况。如果依靠容器身份验证,用户将被调用 `getUserPrincipal()` 的方法传入的HTTP请求。在某些情况下,外部机制可能为用户提供角色/权威信息但在其他当局必须获得一个单独的源,如 `UserDetailsService`.

Pre-Authentication框架类

因为大多数pre-authentication机制遵循相同的模式,Spring Security一组类,提供一个内部框架实现pre-authenticated身份验证提供者。这个删除复制和允许添加新的实现结构化的方式,无需写一切从头开始。你不需要知道这些类,如果希望使用类似于X.509 authentication,因为它已经有了一个名称空间配置选项,简单的使用和开始使用。如果你需要使用显式的bean配置或计划编写自己的实现提供的实现如何工作的理解将是有用的。你会发现类“org.springframework.security.web.authentication.preauth”。我们在适当的地方提供一个大纲你应该咨询Javadoc和源.

抽象的预认证处理过滤器

这个类将检查的当前内容安全上下文,如果空,它将尝试从HTTP请求中提取用户信息并提交 `AuthenticationManager`,子类覆盖以下方法来获得这些信息:

```
  protected abstract Object getPreAuthenticatedPrincipal(HttpServletRequest request);

protected abstract Object getPreAuthenticatedCredentials(HttpServletRequest request);
```

把这些后,过滤器将包含返回的数据创建一个 `PreAuthenticatedAuthenticationToken` 并提交身份验证。
由"authentication",我们或许真的只是意味着进一步的处理负荷用户的部门,而不是按照标准的Spring Security验证架构。

像其他Spring安全身份验证过滤器,pre-authentication过滤器有一个 `authenticationDetailsSource` 属性,默认情况下将创建一个 `WebAuthenticationDetails` 对象来存储更多的信息,比如会话标识符和原始IP地址在 `Authentication` 对象的属性 `details`。在这种情况下,用户角色信息可从pre-authentication获取机制,数据也存储在这个属性,实现 `GrantedAuthoritiesContainer` 接口的细节。这使的身份验证提供者阅读部门外部分配给用户。接下来我们将看一个具体的例子.

基于J2ee的前验证Web身份验证源细节

如果过滤器配置了一个 `authenticationDetailsSource` 这类的一个实例,权威的信息是通过调用 `isUserInRole(String role)` 的一组预先确定的方法为每一个"mappable roles"配置的类这些来自一个 `MappableAttributesRetriever`.可能的实现包括硬编码应用程序上下文中的一个列表和阅读的角色信息 `<security-role>` 在 `web.xml` 文件.pre-authentication示例应用程序使用了后一种方法.

有一个额外的阶段(或属性)的角色被映射到Spring Security `GrantedAuthority` 对象使用一个配置 `Attributes2GrantedAuthoritiesMapper`。默认只会添加通常具备ROLE_前缀的名字,但它让你完全控制行为

前验证身份验证提供者

re-authenticated提供者有更多比为用户负载的`UserDetails`对象。它通过委托给一个`AuthenticationUserDetailsService`.后者是类似于标准`UserDetailsService`但以一个`Authentication`对象而不是用户名:

```
  public interface AuthenticationUserDetailsService {
UserDetails loadUserDetails(Authentication token) throws UsernameNotFoundException;
}
```

这个接口可能也其他用途,但pre-authentication它允许访问官方包装在“身份验证”对象,正如我们在前一节中看到的。`PreAuthenticatedGrantedAuthoritiesUserDetailsService`这类,或者,它可能代表一个标准的`UserDetailsService`通过`UserDetailsByNameServiceWrapper`实现.

Http403禁止入口点

`AuthenticationEntryPoint`是讨论的technical overview 一章。通常它负责启动未经过身份验证的用户的身份验证过程(当他们试图访问受保护的资源),但是在pre-authenticated情况下不适用。你只会配置`ExceptionTranslationFilter`与这个类的一个实例,如果你不使用pre-authentication结合其他身份验证机制。它将被称为`AbstractPreAuthenticatedProcessingFilter`如果用户被拒绝的结果在一个空的身份验证。它总是返回一个“403”错误.

具体实现

X.509认证被 own chapter覆盖.下面我们来看看一些类,它们提供支持其他pre-authenticated场景.

请求头身份验证(Siteminder)

设置特定的HTTP请求。一个众所周知的例子是Siteminder,通过用户名在一个标题叫`SM_USER`。这种机制是类`RequestHeaderAuthenticationFilter`支持,只是从标题中提取用户名。它默认使用的名称`SM_USER`作为标题名称。看到更多的细节的Javadoc.

>
> 注意,当使用这样的一个系统,框架执行任何身份验证检查和*extremely*重要外部系统的正确配置和保护所有访问应用程序。如果攻击者能够伪造原始请求的头文件没有被发现之后,他们可以选择任何他们希望的用户名.

Siteminder示例配置

一个典型的配置使用这个过滤器看起来像这样:

```
  <security:http>
<!-- Additional http configuration omitted -->
<security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />
</security:http>

<bean id="siteminderFilter"
class="org.springframework.security.web.authentication.preauth.RequestHeaderAuthenticationFilter">
<property name="principalRequestHeader" value="SM_USER"/>
<property name="authenticationManager" ref="authenticationManager" />
</bean>

<bean id="preauthAuthProvider"
class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthenticationProvider">
<property name="preAuthenticatedUserDetailsService">
 <bean id="userDetailsServiceWrapper"
  class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
 <property name="userDetailsService" ref="userDetailsService"/>
 </bean>
</property>
</bean>

<security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="preauthAuthProvider" />
</security:authentication-manager>
```

我们认为这里security namespace是用于配置。还假定您已经添加了一个`UserDetailsService`(称为"userDetailsService")到您的配置加载用户的角色.

Java EE容器认证

`J2eePreAuthenticatedProcessingFilter`将从`userPrincipal`属性中提取`HttpServletRequest`.使用这个过滤器通常会结合使用Java EE角色如上所述在< < j2ee-preauth-details > >表示.

有一个示例应用程序代码中使用这种方法,从github得到的代码从github,如果你对这些文件感兴趣你可以看下，代码是在`samples/xml/preauth`目录中.

LDAP 身份验证

概述

LDAP作为一个中央存储库对用户信息和身份验证服务是常用的组织.它也可以用于存储应用程序用户的角色信息.这里有一些不同的场景对于如何配置LDAP服务器，因此Spring Security LDAP提供者是完全可配置的，它使用单独的策略为身份验证和角色接口检索,并提供默认的实现,可以配置为处理各种情况.在使用之前你应该熟悉Spring Security LDAP.以下链接提供了一个很好的介绍涉及的概念和使用OpenLDAP免费的LDAP服务器建立一个目录指南 http://www.zytrax.com/books/ldap/[http://www.zytrax.com/books/ldap/]. 一些熟悉的JNDI api用于访问LDAP从Java也可能是有用的。我们在LDAP不使用任何第三方LDAP库(Mozilla,JLDAP等等),但Spring 的广泛使用是由LDAP,如果你打算添加您自己的定制,对这方面有所了解对你的项目可能是有用的.

使用LDAP身份验证时,重要的是要确保你正确配置LDAP连接池。如果你不熟悉如何做到这一点,你可以参考 Java LDAP documentation.

使用LDAP Spring Security

在Spring的LDAP身份验证安全大致可以分为以下几个阶段

- 获得独特的LDAP"Distinguished Name",或DN,登录名。这通常意味着执行搜索的目录,除非用户名的具体映射DNs是提前知道。所以用户可能输入名称登录"joe",但实际LDAP名称用于验证将完整的DN,如 `uid=joe,ou=users,dc=spring,dc=io`.
- 验证用户,通过"binding",用户操作的用户的密码与密码属性执行远程"compare" 目录条目的DN.
- 加载当局为用户的列表.

唯一的例外是当LDAP目录只是被用于检索用户信息并在本地对其进行身份验证.这个不可能设置有限的读访问属性目录,如用户密码.

下面,我们将看看一些配置场景。完整的可用配置选项的信息,请查阅安全模式名称空间(信息应该在XML编辑器中可用).

配置LDAP服务器

你需要做的第一件事是配置的服务器身份验证应该发生。这是通过使用 `<ldap-server>` 的元素从安全名称空间。这可以配置为指向外部LDAP服务器,使用的 `url` 属性:

```
<ldap-server url="ldap://springframework.org:389/dc=springframework,dc=org" />
```

Using an Embedded Test Server

`<ldap-server>` 元素也可以用来创建一个嵌入式服务器,它可以是非常有用的进行测试和演示。在这种情况下,你没有使用它的url属性:

```
<ldap-server root="dc=springframework,dc=org"/>
```

这里我们指定目录是"dc=springframework,dc=org",这是默认的.这种方式,使用名称空间解析器将创建一个嵌入式Apache目录服务器的类路径和扫描任何LDIF文件,它将尝试加载到服务器。你可以定制这种行为使用ldif的属性,它定义了一个ldif资源加载:

```
<ldap-server ldif="classpath:users.ldif" />
```

这使它更容易与LDAP同步,因为它可以方便工作与外部服务器。它还将用户从复杂bean配置需要隔离一个Apache连接目录服务器。使用普通的Spring bean配置将会更加混乱。你必须要有必要的Apache Directory依赖性jar用于您的应用程序使用。如LDAP示例应用程序。

使用绑定验证

这是最常见的LDAP身份验证场景.

```
<ldap-authentication-provider user-dn-pattern="uid={0},ou=people"/>
```

这个简单的例子将获得用户的DN替代模式和提供的用户登录名试图绑定,用户的登录密码。如果你所有的用户都是存储在单个节点的目录下这很好,如果您希望配置LDAP搜索筛选器来定位用户,你可以使用以下

```
<ldap-authentication-provider user-search-filter="(uid={0})"
user-search-base="ou=people"/>
```

如果使用上面的服务器定义,这将执行搜索在DN `ou=people,dc=springframework,dc=org` 使用 `user-search-filter` 属性的值作为一个过滤器。用户登录名是代替过滤器的参数名称,所以它将搜索条目 `uid` 属性等于用户名。如果 `user-search-base` 并不提供,从根搜索.

加载机构

当局是如何从组加载在LDAP目录中由以下属性控制

- `group-search-base` .定义了目录树下的一部分应该执行哪一组搜索。

- `group-role-attribute`.属性包含的名称定义的权限组条目。默认为`cn`
- 组搜索过滤器。过滤器用于搜索组成员。默认是`uniqueMember={0}`,对应于`groupOfUniqueNames`的LDAP类脚注:(注意,这是不同于缺省配置底层`DefaultLdapAuthoritiesPopulator`使用`member={0}`。]。在这种情况下,替换参数是用户的专有名称。可以使用参数`{1}`如果你想过滤的登录名.

因此,如果我们使用以下配置

```
<ldap-authentication-provider user-dn-pattern="uid={0},ou=people"
group-search-base="ou=groups" />
```

和经过验证的成功作为用户"ben",随后的加载下当局将执行搜索的目录条目的`ou=groups,dc=springframework,dc=org`,寻找条目包含的属性`uniqueMember`价值`uid=ben,ou=people,dc=springframework,dc=org`。默认的权限名称前缀`ROLE_`前缀。你可以改变这个使用`role-prefix`属性。如果你不想要任何前缀,使用`role-prefix="none"`.加载机构的更多信息,请参阅`DefaultLdapAuthoritiesPopulator`类的Javadoc.

实现类

上面的名称空间配置选项我们使用易于使用,更比使用Spring bean简洁明确。有些情况下,您可能需要了解如何配置Spring Security LDAP直接在您的应用程序上下文。您可能希望定制的一些类的行为,例如。如果你使用名称空间配置,那么你可以跳过这一节和下一个.

LDAP provider,`LdapAuthenticationProvider`,实际上并不做太多工作本身,而是代表其他两个bean,一个`LdapAuthenticator`和一个`LdapAuthoritiesPopulator`分别负责验证用户和检索用户的组`GrantedAuthority`.

Ldap身份验证实现

authenticator还负责检索所需的用户属性。这是因为权限的属性可能取决于正在使用的身份验证类型。例如,如果绑定用户,与用户可能需要阅读的权限有关.

目前有两种身份验证策略提供Spring安全:

*直接向LDAP服务器的身份验证("bind" 身份验证)。

*密码比较,用户提供的密码是与一个存储在存储库中。这可以通过检索密码属性的值和检查本地或通过执行LDAP"compare"操作,提供的密码在哪里传递到服务器进行比较和真正的密码值是没有检索到.

常用功能

之前可以验证一个用户(通过策略),专有名称(DN)必须从登录名获得提供给应用程序.这可以通过简单的模式匹配(通过设置`setUserDnPatterns`数组属性)或通过设置`userSearch`属性.对于DN模式匹配方法,格式是使用一个标准的Java模式,将登录名代替参数`{0}`.他应该相对于DN模式,配置`SpringSecurityContextSource`将绑定到部分(参见connecting to the LDAP server 更多这方面的信息).`ldap://monkeymachine.co.uk/dc=springframework,dc=org`和有一个模式`uid={0},ou=greatapes`,"gorilla" 的登录名将会映射到一个DN`uid=gorilla,ou=greatapes,dc=springframework,dc=org`.每个配置的DN模式将尝试直到找到一个匹配，有关使用搜索的信息,看到部分下面的search objects，结合这两种方法也可以使用——模式首先会检查,如果没有找到匹配DN,将使用搜索.

绑定认证者

org.springframework.security.ldap BindAuthenticator实现身份验证绑定验证策略。它只是试图将用户绑定.

身份验证密码比较

`PasswordComparisonAuthenticator`实现了密码比较验证策略.

连接到LDAP服务器

上面讨论的bean必须能够连接到服务器。他们都必须提供一个`SpringSecurityContextSource`这是SpringLDAP的`ContextSource`的延伸。除非你有特殊要求,您通常会配置一个DefaultSpringSecurityContextSource bean,可以配置LDAP服务器的URL和可选的"manager"的用户的用户名和密码,使用时将默认绑定到服务器(而不是匿名绑定)。更多信息,读取这个类的Javadoc和SpringLDAP的`AbstractContextSource`”.

LDAP搜索对象

通常需要一个比DN-matching定位目录中的用户条目更复杂的策略，这可以封装在一个`LdapUserSearch`实例,可以提供身份验证实现.例如,让他们来定位用户.提供的实现是`FilterBasedLdapUserSearch`.

滤波器基于Ldap用户搜索

这个bean使用LDAP目录中的过滤器匹配用户对象。Javadoc的过程解释相应的搜索方法 JDK DirContext class.作为解释,搜索筛选器可以提供参数。这个类,唯一有效的参数是`{0}`将取代用户的登录名.

Ldap当局填充器

验证用户成功后,`LdapAuthenticationProvider` 将试图通过调用配置LdapAuthoritiesPopulator bean加载一组当局用户.`DefaultLdapAuthoritiesPopulator` 是一个实现加载当局通过搜索目录组的用户成员(通常这些将 `groupOfNames` ”或 `groupOfUniqueNames` 加入目录中的条目)详细内容,请参阅这个类的Javadoc中它是如何工作的.

如果你只想使用LDAP身份验证,但加载当局从不同来源(比如数据库),那么您可以提供自己的实现这个接口和注入.

Spring Bean 配置

典型的配置中,我们这里讨论使用一些bean,看起来像这样:

```
  <bean id="contextSource"
  class="org.springframework.security.ldap.DefaultSpringSecurityContextSource">
<constructor-arg value="ldap://monkeymachine:389/dc=springframework,dc=org"/>
<property name="userDn" value="cn=manager,dc=springframework,dc=org"/>
<property name="password" value="password"/>
</bean>

<bean id="ldapAuthProvider"
 class="org.springframework.security.ldap.authentication.LdapAuthenticationProvider">
<constructor-arg>
<bean class="org.springframework.security.ldap.authentication.BindAuthenticator">
 <constructor-arg ref="contextSource"/>
 <property name="userDnPatterns">
 <list><value>uid={0},ou=people</value></list>
 </property>
</bean>
</constructor-arg>
<constructor-arg>
<bean
 class="org.springframework.security.ldap.userdetails.DefaultLdapAuthoritiesPopulator">
 <constructor-arg ref="contextSource"/>
 <constructor-arg value="ou=groups"/>
 <property name="groupRoleAttribute" value="ou"/>
</bean>
</constructor-arg>
</bean>
```

这将设置提供程序访问LDAP服务器URL`ldap://monkeymachine:389/dc=springframework,dc=org`.身份验证将由试图结合DN`uid=<user-login-name>,ou=people,dc=springframework,dc=org`.成功的身份验证之后,角色分配给用户通过搜索下的DN`ou=groups,dc=springframework,dc=org`用默认的过滤器 `(member=<user's-DN>)` .角色名称将从每一个"ou"属性开始匹配.

配置一个用户搜索对象,使用过滤器 `(uid=<user-login-name>)` 的使用而不是DN-pattern(或补充),您将配置以下bean

```
  <bean id="userSearch"
 class="org.springframework.security.ldap.search.FilterBasedLdapUserSearch">
<constructor-arg index="0" value=""/>
<constructor-arg index="1" value="(uid={0})"/>
<constructor-arg index="2" ref="contextSource" />
</bean>
```

并使用它通过设置 `BindAuthenticator` bean的 `userSearch` 属性.authenticator将调用搜索对象来获得正确的用户作为该用户的DN之前绑定.

LDAP Attributes and Customized UserDetails

身份验证使用 `LdapAuthenticationProvider` 的最终结果是一样的一个正常的Spring安全身份验证使用标准的 `UserDetailsService` 界面。创建一个 `UserDetails` ”对象并返回存储在 `Authentication` ”对象。作为 `UserDetailsService` 使用,一个常见需求是能够定制这个实现和添加额外的属性。当使用LDAP,这些通常会从用户条目属性。 `UserDetails` 对象的创建是由提供者的 `UserDetailsContextMapper` 策略,负责从LDAP上下文映射用户对象和数据:

```
  public interface UserDetailsContextMapper {

UserDetails mapUserFromContext(DirContextOperations ctx, String username,
  Collection<GrantedAuthority> authorities);

void mapUserToContext(UserDetails user, DirContextAdapter ctx);
}
```

唯一重要的是第一个方法进行身份验证。如果您提供该接口的一个实现,它注入 `LdapAuthenticationProvider` ,你能够控制如何创建UserDetails对象。Spring 的第一个参数是一个实例LDAP的 `DirContextOperations` 可以让你接触的LDAP属性加载在身份验证。 `username` 参数是用于验证和最后一个参数是集当局为用户加载的配置 `LdapAuthoritiesPopulator` .

上下文数据加载略有不同的方式取决于您正在使用的身份验证类型。 `BindAuthenticator` ,返回的上下文绑定操作将被用于读取属性,否则数据将从配置读取使用标准的背景下获得 `ContextSource` (当配置搜索来定位用户,这将是搜索返回的数据对象).

活动目录的认证

活动目录支持自己的标准身份验证选项,和正常的使用模式不适合太明显与标准 `LdapAuthenticationProvider` .通常执行身份验

证使用域用户名(`user@domain`),而不是使用LDAP专有名称.为了更简单,Spring Security 3.1有一个身份验证提供者是一个典型的定制活动目录设置.

Active Directory Ldap身份验证提供者

配置ActiveDirectoryLdapAuthenticationProvider非常简单。你只需要提供域名和LDAP服务器的URL提供地址脚注:[还可以获得使用DNS查找服务器的IP地址。当前不支持,但是希望在以后的版本可以实现)。一个例子配置会看起来像这样:

```
  <bean id="adAuthenticationProvider"
class="org.springframework.security.ldap.authentication.ad.ActiveDirectoryLdapAuthenticationProvider">
 <constructor-arg value="mydomain.com" />
 <constructor-arg value="ldap://adserver.mydomain.com/" />
</bean>
}
```

注意,不需要指定一个单独的`ContextSource`来定义服务器位置- bean是完全自包含的。用户名为`sharon`,例如,将能够验证通过输入用户名`sharon`或完整的Active Directory`userPrincipalName`,即`sharon@mydomain.com`.用户的目录条目将被定位,并可能返回的属性中使用自定义创建的UserDetails对象(`UserDetailsContextMapper`可以被注入为此,如上所述)。所有与目录发生交互用户的身份。没有一个"manager"用户的概念.

默认情况下,用户当局正在从`memberOf` 获得用户输入的属性值。政府再分配给用户可以使用被定制`UserDetailsContextMapper`。你也可以注入一个`GrantedAuthoritiesMapper`提供者实例来控制政府最终在`Authentication`对象.

活动目录错误代码

默认情况下,一个失败的结果将导致一个标准的Spring Security`BadCredentialsException`,如果你设置的属性`convertSubErrorCodesToExceptions` 是`true`,异常消息将解析试图提取活性Directory-specific错误代码,提高一个更具体的异常。检查类Javadoc的更多信息。

JSP 标签库

Spring Security有自己的标签库提供基本支持访问安全信息并在jsp应用安全约束.

宣布Taglib

要使用的任何标签,必须有安全JSP 标签库:

```
  <%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

授权标签

这个标签是用来确定是否应评估其内容。在Spring Security 3.0中,可以以两种方式使用脚注:[遗留的选项从Spring Security 2.0也支持,但不推荐.].第一种方法使用一个web-security expression,`access` 属性中指定的标签,表达式求值将委托给`SecurityExpressionHandler<FilterInvocation>` 中定义的应用程序(你应该启用web表达式`<http>`的名称空间配置,以确保这个服务是可用的).

```
  <sec:authorize access="hasRole('supervisor')">

此内容将只对用户可见的"supervisor"权威的< tt > GrantedAuthority < / tt >。

</sec:authorize>
```

在与Spring Security PermissionEvaluator共同使用时,标签也可以用来检查权限.例如:

```
  <sec:authorize access="hasPermission(#domain,'read') or hasPermission(#domain,'write')">

作为一个请求属性命名的"domain",这个内容只会看到读写权限对象用户.

</sec:authorize>
```

一个常见的需求是只显示一个特定的链接,如果用户实际上是允许点击它。我们如何能提前确定事情是否会被允许吗?这个标签也可以在另一种操作模式,允许您定义一个特定的URL属性。如果允许用户调用的URL,然后标记的身体将被评估,否则它将被忽略。所以你可能类似

```
  <sec:authorize url="/admin">

This content will only be visible to users who are authorized to send requests to the "/admin" URL.

</sec:authorize>
```

在你的程序内使用这个标签也必须的一个实例`WebInvocationPrivilegeEvaluator`,如果您正在使用名称空间,将自动注册。这是一 个`DefaultWebInvocationPrivilegeEvaluator`的实例，它创建了一个虚拟网络提供的URL请求,并调用安全拦截器请求

是否会成功或失败，这允许您代表您定义的访问控制设置中使用 `intercept-url` 声明 `<http>` 的名称空间配置省去了重复的信息(如所需的角色)在您的jsp.这种方法还可以加上一个 `method` 属性,提供HTTP方法,更具体的匹配.

布尔结果评估标签(是否允许或拒绝访问)通过设置 `var` 属性变量名称可以存储在一个页面上下文范围变量,避免复制和重新评估在页面中的其他点.

禁用标记授权进行测试

隐藏链接在页面不防止未经授权的用户访问URL。他们可以直接在浏览器中类型,例如，作为您的测试过程的一部分,您可能想要显示隐藏领域为了检查环节是安全的后端。如果你设置系统属性 `spring.security.disableUISecurity` 为 `true` , `authorize` 的标签仍然会跑但不会隐藏其内容,默认情况下它也会包含 `<span class="securityHiddenUI">…</span>` 标签。这允许你显示"hidden"与一个特定的CSS样式等内容不同的背景颜色。尝试运行此属性启用的"tutorial"示例应用程序，例如.

你也可以设置 `spring.security.securedUIPrefix` 和 `spring.security.securedUISuffix` 属性,如果你想改变周围的文字从默认的 `跨度` 标签（或使用空字符串完全删除）.如果你想改变周围文本从默认“跨度”标签(或使用空字符串完全删除它).

身份验证标记

这个标签允许访问当前存储在安全内 `Authentication` 对象,它直接在JSP中 呈现一个属性对象.所以,例如,如果principal`属性的 `Authentication` 是Spring Security的 `UserDetails` 的对象的一个实例,之后用 `<sec:authentication property="principal.username" />` 将会显示当前用户的名称.

当然,这种事情没有必要使用JSP标记,有些人喜欢保持尽可能少的逻辑视图.您可以在你的MVC控制器访问 `Authentication` 对象(通过调用 `SecurityContextHolder.getContext().getAuthentication()`)并将数据直接添加到您的模型渲染的视图.

The accesscontrollist Tag

这个标记只在使用Spring Security的ACL模块,它检查一个以逗号分隔的所需权限指定的域对象.如果当前用户所有的权限,然后标记的内容将被评估。如果他们不这样做,它将被忽略.例如

```
  <sec:accesscontrollist hasPermission="1,2" domainObject="${someObject}">

这将显示用户在给定对象上的所有权限所代表的值“1”或“2”.

</sec:accesscontrollist>
```

在程序内容中,权限被传递到“PermissionFactory”中定义中定义,将它们转换为ACL`Permission`的实例,所以他们支持任何格式工厂,他们不一定是整数,字符串可以像 `READ` 或 `WRITE` .如果没有找到 `PermissionFactory` ,将使用 `DefaultPermissionFactory` 的一个实例.程序中的“AclService”将被用于加载Acl的实例提供对象.Acl的将调用所需的权限,以检查是否都是合法的.

这个标签还支持 `var` 属性,以同样的方式 `authorize` 的标签.

csrfInput标签

如果启用了CSRF保护,这个标签插入一个隐藏表单字段的正确名称和值CSRF保护令牌。如果没有启用CSRF保护,这个标签输出.对于任何的 `<form:form>` 标签使用,通常Spring Security自动插入一个CSRF表单字段.但如果由于某种原因你不能使用 `<form:form>` ,你可以使用 `<form:form>` 更换.

你应该把这个标签在一个HTML `<form></form>` 形式的块,通常地方其他输入字段.不要将这个标签放置在一个Spring `<form:form></form:form>` ,block—Spring Security会自动处理 Spring.

```
   <form method="post" action="/do/something">
  <sec:csrfInput />
  Name:<br />
  <input type="text" name="name" />
  ...
 </form>
```

csrfMetaTags标签

如果启用了CSRF保护,这个标签插入元素标记包含CSRF保护令牌表单字段和标题名称和CSRF保护令牌的值。这些元素标记可用于使用CSRF保护在JavaScript应用程序.

你应该把 `csrfMetaTags` 在一个 HTML `<head></head>` ,通常地方其他元标记。一旦你使用这个标签,你可以使用JavaScript访问表单字段的名称,标题名称.JQuery中使用这个例子使这项任务变得更加简单.

```
  <!DOCTYPE html>
<html>
 <head>
  <title>CSRF Protected JavaScript Page</title>
```

```
  <meta name="description" content="This is the description for this page" />
  <sec:csrfMetaTags />
  <script type="text/javascript" language="javascript">

   var csrfParameter = $("meta[name='_csrf_parameter']").attr("content");
   var csrfHeader = $("meta[name='_csrf_header']").attr("content");
   var csrfToken = $("meta[name='_csrf']").attr("content");

   // using XMLHttpRequest directly to send an x-www-form-urlencoded request
   var ajax = new XMLHttpRequest();
   ajax.open("POST", "http://www.example.org/do/something", true);
   ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded data");
   ajax.send(csrfParameter + "=" + csrfToken + "&name=John&...");

   // using XMLHttpRequest directly to send a non-x-www-form-urlencoded request
   var ajax = new XMLHttpRequest();
   ajax.open("POST", "http://www.example.org/do/something", true);
   ajax.setRequestHeader(csrfHeader, csrfToken);
   ajax.send("...");

   // using JQuery to send an x-www-form-urlencoded request
   var data = {};
   data[csrfParameter] = csrfToken;
   data["name"] = "John";
   ...
   $.ajax({
    url: "http://www.example.org/do/something",
    type: "POST",
    data: data,
    ...
   });

   // using JQuery to send a non-x-www-form-urlencoded request
   var headers = {};
   headers[csrfHeader] = csrfToken;
   $.ajax({
    url: "http://www.example.org/do/something",
    type: "POST",
    headers: headers,
    ...
   });

  <script>
 </head>
 <body>
  ...
 </body>
</html>
```

如果不启用CSRF保护,`csrfMetaTags`输出.

Java Authentication and Authorization Service (JAAS) Provider

概述

Spring Security提供一个包可以将身份验证请求委托给Java身份验证和授权服务(JAAS).这个包是在下面详细讨论.

Jaas远程准入Provider摘要

`AbstractJaasAuthenticationProvider`提供JAAS `AuthenticationProvider`实现的基础.子类必须实现方法,创建了`LoginContext`. `AbstractJaasAuthenticationProvider`有许多依赖关系,下面讨论它

JAAS回调处理程序

大多数JAAS `LoginModule`的年代需要一个回调。这些回调通常用于获得用户的用户名和密码.

在Spring Security部署中,Spring Security负责这个用户交互(通过身份验证机制),因此,当委托到JAAS身份验证请求,Spring安全的身份验证机制已经完全填充一个身份验证的对象包含所有所需的JAAS LoginModule的信息.

因此,为Spring Security JAAS包提供了两个默认回调处理程序,`JaasNameCallbackHandler`和`JaasPasswordCallbackHandler`.每一个回调处理程序实现`JaasAuthenticationCallbackHandler`.在大多数情况下,这些回调处理程序可以简单地使用不了解内部力学.

对于那些需要完全控制回调行为,内部`AbstractJaasAuthenticationProvider`用`InternalCallbackHandler`包装这些`JaasAuthenticationCallbackHandler`. `InternalCallbackHandler`是类实现JAAS正常的CallbackHandler接口。任何时候使用JAAS LoginModule的,它是通过一个应用程序上下文列表配置InternalCallbackHandler年代。如果LoginModule的请求一个回调兑`InternalCallbackHandler`,回调是循序传递到`JaasAuthenticationCallbackHandler`被包装.

JAAS授权授与者

JAAS和主程序一起工作。甚至"roles"在JAAS表示为主体，Spring Security, 另一方面, 与`Authentication`对象,每个`Authentication`对象包含一个校长,和多个`GrantedAuthority` .促进这些不同概念之间的映射,Spring Security的JAAS包包括一个`AuthorityGranter`接口.

`AuthorityGranter` 负责检查JAAS并返回一组字符串的,代表权利分配给主程序.对于每一个权威返回字符串.为每个字符串,返回权威`AbstractJaasAuthenticationProvider`创建了一个“JaasGrantedAuthority”(实现Spring Security的`GrantedAuthority` 接口)包含字符串和JAAS程序`AuthorityGranter`通过.`AbstractJaasAuthenticationProvider`获得JAAS,首先成功地验证用户的使用JAAS LoginModule的凭证,然后访问LoginContext将它返回.调用`LoginContext.getSubject().getPrincipals()`,与每个生成的`AuthorityGranter`主要传递给每个定义为与`AbstractJaasAuthenticationProvider.setAuthorityGranters(List)`的内容.

Spring Security不包括任何生产`AuthorityGranter`,每一个JAAS都有一个特定实现的意义。然而,有一个`TestAuthorityGranter`的单元测试演示了一个简单的`AuthorityGranter`实现.

默认Jaas身份验证提供

`DefaultJaasAuthenticationProvider`允许将一个JAAS配置的对象注入依赖项。然后使用JAAS配置的注入创建一个`LoginContext` .这意味着`DefaultJaasAuthenticationProvider`不绑定任何特定实现的`Configuration`因为`JaasAuthenticationProvider` .

在内存配置

为了使`Configuration`容易注入一个`DefaultJaasAuthenticationProvider` , 默认在内存中实现名为“InMemoryConfiguration”.实现构造函数接受一个`Map` ,每个键代表登录配置名称和值代表一个数组`AppConfigurationEntry` 。如果没有找到`Map`映射提供了`InMemoryConfiguration`还支持一个默认的`AppConfigurationEntry`对象`Array` ,使用详情,请参阅“InMemoryConfiguration”的类级别的javadoc.

默认的Jaas身份验证提供者配置示例

而 `InMemoryConfiguration`的Spring可以更详细配置standarad JAAS配置文件.它在文中`DefaultJaasAuthenticationProvider`比`JaasAuthenticationProvider` 更灵活,因为它不依赖默认配置的实现.

使用`InMemoryConfiguration`配置一个例子`DefaultJaasAuthenticationProvider` .注意,自定义的“配置”可以很容易地实现注入`DefaultJaasAuthenticationProvider` .

```
  <bean id="jaasAuthProvider"
class="org.springframework.security.authentication.jaas.DefaultJaasAuthenticationProvider">
<property name="configuration">
<bean class="org.springframework.security.authentication.jaas.memory.InMemoryConfiguration">
<constructor-arg>
 <map>
 <!--
 SPRINGSECURITY is the default loginContextName
 for AbstractJaasAuthenticationProvider
 -->
 <entry key="SPRINGSECURITY">
 <array>
 <bean class="javax.security.auth.login.AppConfigurationEntry">
  <constructor-arg value="sample.SampleLoginModule" />
  <constructor-arg>
  <util:constant static-field=
   "javax.security.auth.login.AppConfigurationEntry$LoginModuleControlFlag.REQUIRED"/>
  </constructor-arg>
  <constructor-arg>
  <map></map>
  </constructor-arg>
  </bean>
 </array>
 </entry>
 </map>
 </constructor-arg>
</bean>
</property>
<property name="authorityGranters">
<list>
 <!-- You will need to write your own implementation of AuthorityGranter -->
 <bean class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
</list>
</property>
</bean>
```

Jaas Provider准入

`JaasAuthenticationProvider` 假定默认的`Configuration`的一个实例 http://download.oracle.com/javase/1.4.2/docs/guide/security/jaas/spec/com/sun/security/auth/login/ConfigFile.html[ConfigFile].这种假设是为了尝试更新`Configuration` .`JaasAuthenticationProvider`使用默认配置的创建`LoginContext` .

假设我们有一个JAAS登录配置文件,`/WEB-INF/login.conf`,用下面的内容:

```
 JAASTest {
sample.SampleLoginModule required;
};
```

像所有Spring Security beans,`JaasAuthenticationProvider`通过应用程序上下文配置.下面的定义将对应于上面的JAAS登录配置文件

```
 <bean id="jaasAuthenticationProvider"
class="org.springframework.security.authentication.jaas.JaasAuthenticationProvider">
<property name="loginConfig" value="/WEB-INF/login.conf"/>
<property name="loginContextName" value="JAASTest"/>
<property name="callbackHandlers">
<list>
<bean
 class="org.springframework.security.authentication.jaas.JaasNameCallbackHandler"/>
<bean
 class="org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler"/>
</list>
</property>
<property name="authorityGranters">
 <list>
 <bean class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
 </list>
</property>
</bean>
```

项目运行

如果配置,“JaasApiIntegrationFilter”将试图运行“JaasAuthenticationToken”上的`Subject`。这意味着可以使用访问`Subject`:

```
 Subject subject = Subject.getSubject(AccessController.getContext());
```

这种集成可以很容易地使用[jaas-api-provision]配置属性。当集成遗留或外部依赖JAAS Subject API被填充,这个特性很有用

CAS 身份验证

Overview

JA-SIG产生一个企业范围的单点登录系统称为CAS。与其他计划,JA-SIG中央身份验证服务是开源的,广泛使用,容易理解,平台独立,支持代理功能。Spring Security完全支持ca,并提供了一个简单的迁移路径从Spring Security的单个应用程序部署到多个应用程序部署的企业级CAS服务器.

你可以从at http://www.ja-sig.org/cas了解更多CAS.您还需要访问这个网站下载CAS服务器文件.

CAS如何工作

而中科院网站包含文档的细节CAS的体系结构,提出总体概述了Spring Security.Spring Security 3.x支持中科院。在撰写本文时,还在使用CAS服务器3.4版.

在您的企业,您需要设置一个CAS服务器。CAS服务器只是一个标准的WAR文件,所以设置您的服务器没有什么困难。在WAR文件您将定制登录和其他单点登录页面显示给用户.

当部署一个CAS 3.4服务器,您还需要指定一个`AuthenticationHandler`在`deployerConfigContext.xml`包含ca。`AuthenticationHandler`有一个简单的方法,该方法返回一个布尔判断是否一个给定的证书是有效的.
你`AuthenticationHandler`的实现需要链接到某种类型的后端身份验证存储库,如LDAP服务器或数据库,CAS本身包含许多`AuthenticationHandler`的协助.当你下载服务器和部署war文件时,它被设置为成功进行身份验证的用户输入一个密码匹配他们的用户名,用于测试.

Spring Security和CAS交互序列

基本的web浏览器之间的交互,CAS服务器和春天Security-secured服务如下:

- web用户浏览服务的公共页面.CAS或Spring Security不参与。
- 最终用户请求一个页面或者安全bean的使用是安全的。Spring Security的ExceptionTranslationFilter将检测到`AccessDeniedException`或`AuthenticationException`.
- 因为用户的`Authentication`的对象(或缺乏)引起了`AuthenticationException`,`ExceptionTranslationFilter`将调用配置的`AuthenticationEntryPoint`。如果使用CAS,这将是`CasAuthenticationEntryPoint`类.
- `CasAuthenticationEntryPoint`会将用户的浏览器重定向到CAS服务器。它也将显示一个`service`参数,这是春天的回调

URL安全服务(应用程序)。例如,浏览器的URL重定向可能是https://my.company.com/cas/login?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas。

- 用户的浏览器重定向到ca后,他们将提示输入用户名和密码。如果用户提供了一个会话cookie这表明他们之前登录的,他们不会被提示重新登录(这个过程有一个例外,我们稍后将讨论)。中科院将使用`PasswordHandler`(或果使用CAS 3.0`AuthenticationHandler`)以上讨论决定是否用户名和密码是有效的。
- 成功登录之后,中科院重定向用户的浏览器将回到原来的服务。它还将包括一个`ticket`参数,这是一个不透明的字符串代表"service ticket".继续我们前面的例子中,浏览器重定向到URL https://server3.company.com/webapp/login/cas?ticket=ST-0-ER94xMJmn6pha35CQRoZ。
- 服务的web应用程序中,`CasAuthenticationFilter`总是侦听请求`/login/cas`(这是可配置的,但是我们将使用默认本介绍)。处理过滤器将构造一个`UsernamePasswordAuthenticationToken`代表服务票证。校长就等于`CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`,同时将服务票证凭据不透明值。这种身份验证请求将被交给`AuthenticationManager`配置。
- `AuthenticationManager`实现`ProviderManager`,这是依次配置了`CasAuthenticationProvider`.`CasAuthenticationProvider`只响应包含`UsernamePasswordAuthenticationToken`(如`CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`)和`CasAuthenticationToken`(稍后讨论).
- `CasAuthenticationProvider`将验证服务票据使用`TicketValidator`实现。这通常是一个`Cas20ServiceTicketValidator`这是一个类包含在CAS客户端库。如果应用程序需要验证代理机票,使用`Cas20ProxyTicketValidator`.`TicketValidator`发出一个HTTPS请求CAS服务器以验证服务票证。它可能还包括一个代理回调URL,包括在这个例子:https://my.company.com/cas/proxyValidate?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas&ticket=ST-0-ER94xMJmn6pha35CQRoZ&pgtUrl=https / /server3.company.com/webapp/login/cas/proxyreceptor。
- CAS服务器,验证请求将被接收。如果提供的服务票证匹配服务URL票发行,中科院将提供积极响应XML显示用户名。如果任何代理参与身份验证(下面讨论),代理列表中也包含在XML响应。
- [OPTIONAL] 如果请求CAS验证服务包括代理回调URL(在`pgtUrl`参数),中科院将包括一个`pgtIou`字符串在XML响应。这种`pgtIou`代表一个借据proxy-granting机票。CAS服务器会创建自己的HTTPS连接回`pgtUrl`.这是相互CAS服务器进行身份验证,声称服务URL。HTTPS连接将被用来发送一个代理发放门票最初的web应用程序.例如,https://server3.company.com/webapp/login/cas/proxyreceptor?pgtIou=PGTIOU-0-R0zlgrl4pdAQwBvJWO3vnNpevwqStbSGcq3vKB2SqSFFRnjPHt&pgtId=PGT-1-si9YkkHLrtACBo64rmsi3v2nf7cpCResXg5MpESZFArbaZiOKH。
- `Cas20TicketValidator`将收到CAS服务器解析XML。它将返回到`CasAuthenticationProvider`的`TicketResponse`,其中包括用户名(强制),代理列表(如果有涉及),和proxy-granting票借据(如果请求代理回调)。
- 接下来的`CasAuthenticationProvider`将称之为`CasProxyDecider`配置.`CasProxyDecider`表明代理列表中`TicketResponse`是否接受服务。几个实现提供SpringSecurity:`RejectProxyTickets`,`AcceptAnyCasProxy`和`NamedCasProxyDecider`。这些名字在很大程度上是自解释的,除了`NamedCasProxyDecider`允许`List`提供可信的代理。
- `CasAuthenticationProvider`将下一个请求的`AuthenticationUserDetailsService`加载`GrantedAuthority`对象,适用于用户包含在`Assertion`.
- 如果没有问题,`CasAuthenticationProvider`构造CasAuthenticationToken包括细节包含在`TicketResponse`和`GrantedAuthority`.
- 控制然后返回`CasAuthenticationFilter`,把`CasAuthenticationToken`创建安全上下文.
- `AuthenticationException`导致用户的浏览器被重定向到原始页面 (or a custom destination根据配置).

很好,你还在这里!现在让我们看看这是如何配置的

配置客户案件

由于Spring Security中科院的web应用程序是很容易.这是假设你已经知道使用Spring安全的基本知识,下面这些是不会再覆盖。我们假设基于命名空间的配置使用,根据需要添加在CAS bean。每个部分建立在前一节。一个完整的CAS sample application可以在Spring Security样本找到.

门票远程准入服务

本节描述如何设置Spring Security验证Service Tickets。很多时候这都是一个web应用程序需要。您需要添加一个“ServiceProperties”bean到您的应用程序上下文。这代表你的CAS服务:

```
  <bean id="serviceProperties"
 class="org.springframework.security.cas.ServiceProperties">
<property name="service"
 value="https://localhost:8443/cas-sample/login/cas"/>
```

```
<property name="sendRenew" value="false"/>
</bean>
```

`service` 必须等于一个URL,将由 `CasAuthenticationFilter` 监控. `sendRenew` 的默认值为false,但如果应用程序尤其敏感应该设置为true.这个参数的作用是告诉CAS登录服务,一个单点登录登录是不可接受的。相反,用户将需要重新输入自己的用户名和密码来访问服务.

下面的bean应该配置开始CAS认证过程(假设您正在使用一个名称空间配置):

```
  <security:http entry-point-ref="casEntryPoint">
...
<security:custom-filter position="CAS_FILTER" ref="casFilter" />
</security:http>

<bean id="casFilter"
 class="org.springframework.security.cas.web.CasAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>

<bean id="casEntryPoint"
 class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
<property name="loginUrl" value="https://localhost:9443/cas/login"/>
<property name="serviceProperties" ref="serviceProperties"/>
</bean>
```

CAS操作, `ExceptionTranslationFilter` 必须有它 `authenticationEntryPoint` 属性设置为 `CasAuthenticationEntryPoint` "bean.这可以很容易地通过使用 entry-point-ref,是在上面的示例中完成的。 `CasAuthenticationEntryPoint` 必须参考的ServiceProperties bean(如上所述),它提供了企业的CAS登录服务器的URL.就是用户的浏览器重定向.

`CasAuthenticationFilter` 已经和属性 `UsernamePasswordAuthenticationFilter` 非常相似(用于基于表单的登录)。您可以使用这些属性来定制诸如认证成功和失败的行为.

接下来,您需要添加一个"CasAuthenticationProvider"及其合作者:

```
  <security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="casAuthenticationProvider" />
</security:authentication-manager>

<bean id="casAuthenticationProvider"
 class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
<property name="authenticationUserDetailsService">
 <bean class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
 <constructor-arg ref="userService" />
 </bean>
</property>
<property name="serviceProperties" ref="serviceProperties" />
<property name="ticketValidator">
 <bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
 <constructor-arg index="0" value="https://localhost:9443/cas" />
 </bean>
</property>
<property name="key" value="an_id_for_this_auth_provider_only"/>
</bean>

<security:user-service id="userService">
<security:user name="joe" password="joe" authorities="ROLE_USER" />
...
</security:user-service>
```

`CasAuthenticationProvider` 使用 `UserDetailsService` 实例加载权限用户,一旦被CAS认证。我们已经演示了一个简单的内存设置。注意, `CasAuthenticationProvider` 并不实际使用密码进行身份验证,但它使用当局.

如果你参考How CAS Works 部分beans 是合理的.

这对CAS完成最基本的配置。如果你没有做任何错误,您的web应用程序框架内应该记录地工作CAS的单点登录。没Spring Security 的其他部分的安全需要关心事实CAS处理身份验证。在下面几节中我们将讨论一些更高级的配置(可选).

Single Logout

CAS协议支持单注销和Spring Security 很容易地添加到您的安全配置。下面是更新Spring Security配置处理单注销

```
  <security:http entry-point-ref="casEntryPoint">
...
<security:logout logout-success-url="/cas-logout.jsp"/>
<security:custom-filter ref="requestSingleLogoutFilter" before="LOGOUT_FILTER"/>
<security:custom-filter ref="singleLogoutFilter" before="CAS_FILTER"/>
</security:http>

<!-- This filter handles a Single Logout Request from the CAS Server -->
```

```
<bean id="singleLogoutFilter" class="org.jasig.cas.client.session.SingleSignOutFilter"/>

<!-- This filter redirects to the CAS Server to signal Single Logout should be performed -->
<bean id="requestSingleLogoutFilter"
 class="org.springframework.security.web.authentication.logout.LogoutFilter">
<constructor-arg value="https://localhost:9443/cas/logout"/>
<constructor-arg>
 <bean class=
  "org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/>
</constructor-arg>
<property name="filterProcessesUrl" value="/logout/cas"/>
</bean>
```

`logout` 元素记录用户的本地应用程序,但不与CAS服务器终止会话或任何其他已经登录的应用程序。 `requestSingleLogoutFilter` 的过滤器将允许 `/spring_security_cas_logout` 请求的url重定向应用程序配置的CAS服务器注销url。然后CAS服务器将发送一个注销请求签署的所有服务. `singleLogoutFilter` 处理单注销请求通过在静态 `Map` 查找的HttpSession然后无效.

也许会困惑,为什么 `logout` 元素和 `singleLogoutFilter` 是必要的。最佳实践是在当地注销以来首次 `SingleSignOutFilter` 只是将HttpSession的存储在一个静态的 `Map` ,以调用无效。与上面的配置中,注销的流程是

- `/logout` 的用户请求将记录用户的本地应用程序并发送用户注销成功页面。
- 注销成功页面 `/cas-logout.jsp` ',为了注销的所有应用程序应该指导用户点击一个链接指向的 `/logout/cas` 。
- 当用户单击链接时,用户被重定向到中科院单注销URL(https://localhost:9443/cas/logout).
- 在CAS服务器端,CAS单注销URL然后提交单注销所有中科院服务的请求。在中科院服务方面,JASIG invaliditing 的 `SingleSignOutFilter` 处理注销请求的原始会话。

下一步是添加到你的web.xml中

```
  <filter>
<filter-name>characterEncodingFilter</filter-name>
<filter-class>
 org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
<init-param>
 <param-name>encoding</param-name>
 <param-value>UTF-8</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>characterEncodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
<listener-class>
 org.jasig.cas.client.session.SingleSignOutHttpSessionListener
</listener-class>
</listener>
```

验证与CAS无状态的服务

本节描述如何使用CAS认证服务。换句话说,本节将讨论如何建立一个客户使用服务,与 CAS进行身份验证。下一节描述了如何设置使用CAS无状态的服务进行身份验证.

配置CAS获得代理发放门票

为了验证一个无状态服务,应用程序需要获得一个代理发放门票(页面表)。本节描述如何配置Spring Security获得页面表构建在thencas-st[Service Ticket Authentication] 配置.

第一步是包括 `ProxyGrantingTicketStorage` 在你的Spring Security配置。这是用于存储页面表所获得 `CasAuthenticationFilter` ,这样他们可以用来获得代理票。一个示例配置如下所示

```
  <!--
NOTE: In a real application you should not use an in
  memory implementation. You will also want to ensure
  to clean up expired tickets by calling ProxyGrantingTicketStorage.cleanup()
-->
<bean id="pgtStorage" class="org.jasig.cas.client.proxy.ProxyGrantingTicketStorageImpl"/>
```

下一步是更新的 `CasAuthenticationProvider` 能够获得代理票。将 `Cas20ServiceTicketValidator` 替换为一个 `Cas20ProxyTicketValidator` . `proxyCallbackUrl` 应该设置为一个应用程序将接收页面表的URL.最后,配置也应该参考 `ProxyGrantingTicketStorage` 所以它可以使用页面表获取代理机票。你可以找到一个例子,

```
  <bean id="casAuthenticationProvider"
 class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
```

```
<property name="ticketValidator">
 <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
 <constructor-arg value="https://localhost:9443/cas"/>
  <property name="proxyCallbackUrl"
  value="https://localhost:8443/cas-sample/login/cas/proxyreceptor"/>
 <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
 </bean>
</property>
</bean>
```

最后一步是在 `ProxyGrantingTicketStorage` 更新 `CasAuthenticationFilter` 接受页面表和存储。重要的是 `proxyReceptorUrl` 与 `proxyCallbackUrl` 匹配的 `Cas20ProxyTicketValidator` .一个示例配置如下所示

```
  <bean id="casFilter"
  class="org.springframework.security.cas.web.CasAuthenticationFilter">
 ...
 <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
 <property name="proxyReceptorUrl" value="/login/cas/proxyreceptor"/>
</bean>
```

使用代理调用无状态服务票

现在Spring Security获得页面表,您可以使用它们来创建代理门票可以用来验证无状态的服务,CAS sample application在 `ProxyTicketSampleServlet` 包含一个工作示例。可以找到示例代码如下:

```
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
// NOTE: The CasAuthenticationToken can also be obtained using
// SecurityContextHolder.getContext().getAuthentication()
final CasAuthenticationToken token = (CasAuthenticationToken) request.getUserPrincipal();
// proxyTicket could be reused to make calls to the CAS service even if the
// target url differs
final String proxyTicket = token.getAssertion().getPrincipal().getProxyTicketFor(targetUrl);

// Make a remote call using the proxy ticket
final String serviceUrl = targetUrl+"?ticket="+URLEncoder.encode(proxyTicket, "UTF-8");
String proxyResponse = CommonUtils.getResponseFromServer(serviceUrl, "UTF-8");
...
}
```

代理身份验证票

`CasAuthenticationProvider` 的区分有状态的和无状态的客户。一个有状态的客户被认为是提交 `CasAuthenticationFilter` 的 `filterProcessUrl` .无状态的客户的任何一个身份验证请求 `CasAuthenticationFilter` 比 `filterProcessUrl` 另一个URL更好.

因为远程协议没有办法展示自己的HttpSession,它不可能依赖于默认的做法将安全上下文存储在会话请求之间。此外,由于CAS服务器无效罚单后,在后续请求中验证了 `TicketValidator` 呈现相同的代理机票不会工作.

一个显而易见的选择是为远程协议客户不使用CAS。然而,这将消除许多CAS。作为一个中间立场, `CasAuthenticationProvider` 使用 `StatelessTicketCache` .这是仅用于无状态的客户主要使用等于 `CasAuthenticationFilter.CAS_STATELESS_IDENTIFIER` .发生了什么是 `CasAuthenticationProvider` 将存储产生的 `CasAuthenticationToken` 放在 `StatelessTicketCache` ,键控代理机票。因此,远程协议客户可以呈现相同的代理机票和 `CasAuthenticationProvider` 不需要接触CAS服务器进行验证(除了第一个请求)。一旦验证,除了最初的目标服务代理机票可以用于url.

本节建立在前面几节容纳代理机票验证.第一步是指定验证所有工件如下所示

```
  <bean id="serviceProperties"
 class="org.springframework.security.cas.ServiceProperties">
...
<property name="authenticateAllArtifacts" value="true"/>
</bean>
```

下一步是指定 `serviceProperties` 和 `authenticationDetailsSource` 和 `CasAuthenticationFilter` 。“serviceProperties”属性指示 `CasAuthenticationFilter` 尝试所有的工件进行身份验证,而不是只有出现在 `filterProcessUrl` . `ServiceAuthenticationDetailsSource` 创建了一个 `ServiceAuthenticationDetails` 确保当前URL,基于 `HttpServletRequest` ,用作服务URL时验证票。方法用于生成服务URL可以被注入一个自定义定制的 `AuthenticationDetailsSource` ,返回一个自定义 `ServiceAuthenticationDetails` .

```
  <bean id="casFilter"
 class="org.springframework.security.cas.web.CasAuthenticationFilter">
...
<property name="serviceProperties" ref="serviceProperties"/>
<property name="authenticationDetailsSource">
 <bean class=
 "org.springframework.security.cas.web.authentication.ServiceAuthenticationDetailsSource">
 <constructor-arg ref="serviceProperties"/>
```

```
 </bean>
</property>
</bean>
```

您还需要更新`CasAuthenticationProvider`处理代理票。将`Cas20ServiceTicketValidator`替换为一个`Cas20ProxyTicketValidator`.您需要配置代理的`statelessTicketCache`,你想接受。你可以找到一个例子,下面的更新需要接受所有代理.

```
  <bean id="casAuthenticationProvider"
 class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
 <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
 <constructor-arg value="https://localhost:9443/cas"/>
 <property name="acceptAnyProxy" value="true"/>
 </bean>
</property>
<property name="statelessTicketCache">
 <bean class="org.springframework.security.cas.authentication.EhCacheBasedTicketCache">
 <property name="cache">
  <bean class="net.sf.ehcache.Cache"
   init-method="initialise" destroy-method="dispose">
  <constructor-arg value="casTickets"/>
  <constructor-arg value="50"/>
  <constructor-arg value="true"/>
  <constructor-arg value="false"/>
  <constructor-arg value="3600"/>
  <constructor-arg value="900"/>
  </bean>
 </property>
 </bean>
</property>
</bean>
```

X.509 Authentication

概述

最常用的X.509 证书身份验证是验证服务器在使用SSL的身份,从浏览器通常在使用HTTPS。浏览器会自动检查服务器证书的已发布(即数字签名)的一个受信任的证书颁发机构的列表维护。

您还可以使用SSL与"mutual authentication";服务器将请求从客户端作为一个有效的证书的SSL握手。服务器将验证客户端通过检查其签署的证书是一个可接受的权威。如果提供了一个有效的证书,它可以通过servlet API的应用程序。Spring Security X.509模块提取证书使用一个过滤器。它将证书映射到一个应用程序用户和加载用户的组授予机关使用标准的Spring安全基础设施。

增加X.509认证您的Web应用程序

X.509客户端身份验证非常简单。只是`<x509/>`元素添加到您的http安全性名称空间配置.

```
  <http>
...
 <x509 subject-principal-regex="CN=(.*?)," user-service-ref="userService"/>;
</http>
```

元素有两个可选属性:

- `subject-principal-regex`正则表达式用来提取用户名从证书的主题名称。上面所示的默认值。这是用户名,将传递给“UserDetailsService”为用户负载当局。
- `user-service-ref`.这是的bean Id`UserDetailsService`用于x.如果只有一个定义在应用程序上下文它不需要.

`subject-principal-regex`应该包含一个组。例如默认表达式"CN=(.*?)," 与常见的名称字段。如果证书的主题名称是"CN=Jimi Hendrix, OU=…",这将给一个用户名"Jimi Hendrix",不分大小写。所以"emailAddress=(.?),"匹配"EMAILADDRESS=jimi@hendrix.org,CN=…"给一个用户名"jimi@hendrix.org",如果客户端提供一个证书,成功提取有效的用户名,然后应该有一个有效的安全上下文中的`Authentication`对象.如果没有找到证书,或没有相应的用户可能会发现然后安全上下文仍将是空的。这意味着您可以轻松地使用 X.509年与其他选项,如基于表单的登录身份验证.

在Tomcat中设置SSL

有一些证书的`samples/certificate`的目录在春季安全项目.您可以使用这些启用SSL进行测试如果你不想生成自己的.文件的服务器.jks包含服务器证书、私钥和发行证书的证书颁发机构.也有一些客户端证书文件从示例应用程序用户.你可以在你的浏览器安装这些启用SSL客户机身份验证.

在SSL支持下tomcat运行,下降的`server.jks`文件到tomcat的配置的目录并添加以下连接器的`server.xml`l的文件

```
  <Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true"
    clientAuth="true" sslProtocol="TLS"
    keystoreFile="${catalina.home}/conf/server.jks"
    keystoreType="JKS" keystorePass="password"
    truststoreFile="${catalina.home}/conf/server.jks"
    truststoreType="JKS" truststorePass="password"
/>
```

如果你仍然希望SSL连接成功`clientAuth`也可以被设置为`want`,即使客户没有提供一个证书。客户不提供证书将无法获得的任何对象的访问Spring Security,除非你使用一个non-X.509认证机制,如表单身份验证.

---

run - as验证替换

概述

`AbstractSecurityInterceptor`能够暂时取代`Authentication`对象在`SecurityContext`和`SecurityContextHolder`安全对象回调阶段。这只发生如果最初的`Authentication`对象是成功处理的`AuthenticationManager`和`AccessDecisionManager`.`RunAsManager`将指示更换`Authentication`对象,如果有的话,应该使用在`SecurityInterceptorCallback`.

Configuration

Spring Security提供`RunAsManager`接口:

```
  Authentication buildRunAs(Authentication authentication, Object object,
 List<ConfigAttribute> config);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

第一个方法返回`Authentication`对象应该取代现有的身份验证的对象方法调用的持续时间。如果方法返回`null`,它表明你没有更换。第二种方法是使用`AbstractSecurityInterceptor`启动验证配置属性的一部分。`supports(Class)`的方法被调用以确保安全拦截器将安全对象实现配置的`RunAsManager支持的类型安全拦截器.

一个具体实现`RunAsManager`提供Spring Security,如果`ConfigAttribute`从`RUN_AS_`开始，`RunAsManagerImpl`的类返回一个替代`RunAsUserToken`.如果找到任何此类`ConfigAttribute`,替换“RunAsUserToken”将包含相同的主要,凭证,当局为最初的`Authentication`对象.伴随着一个新的`GrantedAuthorityImpl`为每个`RUN_AS_`ConfigAttribute`.每一个新的`GrantedAuthorityImpl`将前缀`ROLE_`,其次是`RUN_AS`ConfigAttribute`.例如,`RUN_AS_SERVER`将导致更换`RunAsUserToken`包含`ROLE_RUN_AS_SERVER`授予权力.

替代`RunAsUserToken`就像任何其他身份验证的对象。需要验证的`Authentication`,可能通过一个合适的`AuthenticationManager`代表团。`RunAsImplAuthenticationProvider`执行身份验证。它只是接受任何有效的`RunAsUserToken`.

为了确保恶意代码不创建一个`RunAsUserToken`和现在它保证接受‘`RunAsImplAuthenticationProvider`,散列键存储在所有生成的令牌.`RunAsManagerImpl`和`RunAsImplAuthenticationProvider`中创建bean相同的:

```
  <bean id="runAsManager"
 class="org.springframework.security.access.intercept.RunAsManagerImpl">
<property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"
 class="org.springframework.security.access.intercept.RunAsImplAuthenticationProvider">
<property name="key" value="my_run_as_password"/>
</bean>
```

通过使用相同的密钥,每个`RunAsUserToken`可以验证它是由一个`RunAsManagerImpl`批准。出于安全原因`RunAsManagerImpl`创造后是不可变的

Spring Security Crypto模块

引言

Spring Security Crypto模块提供了对称加密，支持密钥的生成、编码和密码。该代码是分布式的核心模块的一部分 但没有任何其他Spring Security (或Spring) 代码的依赖关系。

加密器

加密类提供了构造对称加密工厂方法,使用这个类 您可以创建字节加密器加密数据在byte[]形式， 你也可以构建textencryptors加密文本字符串，且加密是线程安全的。

BytesKeyGenerator

使用encryptors.standard工厂方法构建 "standard" 字节加密机:

```
Encryptors.standard("password", "salt");
```

"standard" 采用的加密方法是：256-bit AES using PKCS #5’s PBKDF2 (Password-Based Key Derivation Function #2). 这种方法要求 Java 6. 该密码用于生成密钥应存放在安全的地方，不能共享。salt是用来防止在事件中的密钥对密钥的字典攻击，你的加密的数据被泄露 一个16字节的随机初始化向量也适用，所以每个加密的消息是唯一的。

所提供的salt应在十六进制编码的字符串形式，是随机的，并至少有8个字节的长度。这种salt可以用keygenerator生成。

```
String salt = KeyGenerators.string().generateKey(); // generates a random 8-byte salt that is then hex-encoded
```

TextEncryptor

使用encryptors.text工厂方法构建一个标准textencryptor:

```
Encryptors.text("password", "salt");
```

TextEncryptor使用一个标准的bytesencryptor加密文本数据。加密结果返回为十六进制编码的字符串，便于存储在文件系统或数据库中。

使用encryptors.queryabletext工厂方法构建一个“可查询”textencryptor:

```
Encryptors.queryableText("password", "salt");
```

一个可查询的textencryptor和标准textencryptor之间的差异做初始化向量 vector(iv) 处理.iv用于查询的textencryptor#加密操作是共享的，或不变的，而不是随机生成的。这意味着同一个文本加密的多次将始终产生相同的加密结果。这是不太安全的，但需要对加密的数据，需要进行查询。可查询加密文本的一个例子是一个OAuth的apikey。

keygenerators

keygenerators类构造密钥生成器不同类型提供了许多便利的工厂方法。 使用这个类，你可以创建一个byteskeygenerator生成byte[]秘钥。你也可以建立一个stringkeygenerator生成字符串键。keygenerators线程是安全的。

BytesKeyGenerator

使用keygenerators.securerandom工厂方法生成的实例byteskeygenerator提供支持:

```
KeyGenerator generator = KeyGenerators.secureRandom();
byte[] key = generator.generateKey();
```

默认密钥长度为8字节.还有一个keygenerators.securerandom变异提供密钥长度控制:

```
KeyGenerators.secureRandom(16);
```

使用keygenerators.shared工厂方法来构建一个byteskeygenerator每次调用，总是返回相同的关键:

```
KeyGenerators.shared(16);
```

StringKeyGenerator

使用keygenerators.string工厂方法构建一个8字节，提供keygenerator进制编码，每个键为字符串:

```
KeyGenerators.string();
```

passwordencoders

Spring Security Crypto模块的密码包提供了编码密码支持. `PasswordEncoder` 是中心的服务接口，并具有以下签名:

```
public interface PasswordEncoder {

String encode(String rawPassword);

boolean matches(String rawPassword, String encodedPassword);
}
```

如果rawpassword编码，等于encodedpassword，方法返回true，此方法的目的是支持基于密码的身份验证方案。

`BCryptPasswordEncoder` 实现使用广泛支持的“BCrypt”算法哈希密码。BCrypt的使用16字节的随机salt值是故意减慢的算法，以阻碍密码破解。它可以使用“强度”参数，从4到31的“强度”参数来调整它的数量。值越高，就必须做更多的工作来计算哈希

值。默认值为10。您可以在部署的系统中更改此值，而不影响现有的密码，因为该值也存储在编码的散列中

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

`Pbkdf2PasswordEncoder`实现使用PBKDF2算法哈希密码。为了打败密码破解PBKDF2是故意慢的算法，调整需要。用5秒来验证你的系统上的密码。

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

并发支持

在大多数环境中，安全存储在每一个 `Thread` 基础. 这意味着当工作是在一个新的 `Thread`，`SecurityContext` 失去了 Spring Security provides提供了一些基础设施，以帮助使这更容 易为用户，Spring Security提供了低级别的抽象，用于在多线程环境中使用 Spring Security，事实上，这是Spring Security建立与整合 AsyncContext.start(Runnable) 和 Spring MVC 异步集成.

DelegatingSecurityContextRunnable

最基本的建筑块内Spring Security’s并发支持 `DelegatingSecurityContextRunnable`. 不包委托 `Runnable` 为了初始化的 `SecurityContextHolder` 用指定的 `SecurityContext` 为代表。然后调用委托运行保障明确 `SecurityContextHolder` 以后 `DelegatingSecurityContextRunnable` 有些东西看起来就像这样:

```
  public void run() {
try {
 SecurityContextHolder.setContext(securityContext);
 delegate.run();
} finally {
 SecurityContextHolder.clearContext();
}
}
```

虽然很简单，但这使得它无缝的securitycontext从一个线程转移到另一个。这很重要，因为，在大多数情况下，每个线程的基础上的securitycontextholder行为都可能使用了Spring Security的 <global-method-security> 来支持你的服务，你现在可以很容易地转移 `SecurityContext` 当前 `Thread` 到 `Thread` 调用安全服务。下面是你如何做这件事的一个例子：

```
  Runnable originalRunnable = new Runnable() {
public void run() {
 // invoke secured service
}
};

SecurityContext context = SecurityContextHolder.getContext();
DelegatingSecurityContextRunnable wrappedRunnable =
 new DelegatingSecurityContextRunnable(originalRunnable, context);

new Thread(wrappedRunnable).start();
```

上面的代码执行以下步骤:

- 创建一个 `Runnable` 这是我们的invoking安全服务，但请注意这是不相关的Spring Security。
- 获得 `SecurityContext` 我们希望从 `SecurityContextHolder` 中初始化 `DelegatingSecurityContextRunnable`。
- 使用 `DelegatingSecurityContextRunnable` 创建一个线程。
- 自从创建了 `DelegatingSecurityContextRunnable` 和 `SecurityContext` 从 `SecurityContextHolder` 这里有一个快捷方式的构造函数，下面的代码与前面的代码相同：

```
  Runnable originalRunnable = new Runnable() {
public void run() {
 // invoke secured service
}
};

DelegatingSecurityContextRunnable wrappedRunnable =
 new DelegatingSecurityContextRunnable(originalRunnable);

new Thread(wrappedRunnable).start();
```

我们有的代码是简单使用的,但它仍然需要使用我们的 Spring Security知识。 在下一节中，我们将看看我们如何利用 `DelegatingSecurityContextExecutor` 隐藏的因素来使用Spring Security。

DelegatingSecurityContextExecutor

在前面的章节中，我们发现，它是很容易使用的`DelegatingSecurityContextRunnable`,但它是不理想，不方便的，因为我们必须意识到Spring Security 是为了使用它.让我们看看`DelegatingSecurityContextExecutor`如何能屏蔽我们的任何代码知识，如果我们使用Spring Security.

设计`DelegatingSecurityContextExecutor`是非常相似于`DelegatingSecurityContextRunnable`除非它接受委托`Executor`而不是一个代表`Runnable`。你可以看到一个例子，它可能会被用在下面：

```
 SecurityContext context = SecurityContextHolder.createEmptyContext();
Authentication authentication =
 new UsernamePasswordAuthenticationToken("user","doesnotmatter", AuthorityUtils.createAuthorityList("ROLE_USER"));
context.setAuthentication(authentication);

SimpleAsyncTaskExecutor delegateExecutor =
 new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
 new DelegatingSecurityContextExecutor(delegateExecutor, context);

Runnable originalRunnable = new Runnable() {
public void run() {
 // invoke secured service
}
};

executor.execute(originalRunnable);
```

该代码执行以下步骤:

- 创建`SecurityContext`用于我们的`DelegatingSecurityContextExecutor`。请注意，在这个例子中，我们简单地手工创建`SecurityContext`。然而在哪里或者怎么得到并不重要`SecurityContext` (i.e. 我们可以从这里获得`SecurityContextHolder` 如果我们想要)。
- 创建一个delegateexecutor是在执行提交`Runnable``s
- 最后我们创建一个 `DelegatingSecurityContextExecutor`这是在包装任何运行，传递到执行的方法`DelegatingSecurityContextRunnable`. 在这个实例中，将包里的delegateexec运行， `SecurityContext` 将每一个Runnable提交给我们 `DelegatingSecurityContextExecutor`. 如果我们运行后台任务，需要由一个具有提升权限的用户运行，这是非常好的，
- 在这一点上，你可能会问自己 "我该如何保护我所学过的 Spring Security代码知识?"来代替创建`SecurityContext`andthe`DelegatingSecurityContextExecutor` 在我们的代码中, 我们可以注入一个已经初始化的实例`DelegatingSecurityContextExecutor`.

```
 @Autowired
private Executor executor; // becomes an instance of our DelegatingSecurityContextExecutor

public void submitRunnable() {
Runnable originalRunnable = new Runnable() {
 public void run() {
 // invoke secured service
 }
};
executor.execute(originalRunnable);
}
```

现在我们的代码是不可知的 `SecurityContext` 正在传播到`Thread`,然后`originalRunnable` 被执行，然后`SecurityContextHolder` 被清除。在这个示例中，同一个用户正在被用于执行每个线程.如果我们想使用户从`SecurityContextHolder` at the time we invoked `executor.execute(Runnable)` (i.e. the currently logged in user) 处理 `originalRunnable`？是可以做到的 removing the `SecurityContext`我们的争论是`DelegatingSecurityContextExecutor`构造函数，比如:

```
 SimpleAsyncTaskExecutor delegateExecutor = new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
 new DelegatingSecurityContextExecutor(delegateExecutor);
```

现在任何时候 `executor.execute(Runnable)` 执行 `SecurityContext` 是首先获得的 `SecurityContextHolder` 然后`SecurityContext` 是用来创建 `DelegatingSecurityContextRunnable`的. 这意味着我们正在执行的 `Runnable` 使用相同的用户来调用 `executor.execute(Runnable)` 代码.

Spring Security Concurrency 并发类

指javadoc额外的集成与java并发API和Spring的抽象任务，一旦你理解了以前的代码你会发现，他们的解释是很单一的。

- DelegatingSecurityContextCallable
- DelegatingSecurityContextExecutor

- DelegatingSecurityContextExecutorService
- DelegatingSecurityContextRunnable
- DelegatingSecurityContextScheduledExecutorService
- DelegatingSecurityContextSchedulingTaskExecutor
- DelegatingSecurityContextAsyncTaskExecutor
- DelegatingSecurityContextTaskExecutor

Spring MVC 整合

Spring Security 本节涵盖了进一步的细节的集成， 提供了一些可选的集成与Spring MVC

@EnableWebMvcSecurity

> Spring Security 4.0, `@EnableWebMvcSecurity` 是 不好的. 更换 `@EnableWebSecurity` w这决定将基于加入Spring MVC的特点。

使MVC和Spring Security更好的整合与集成`@EnableWebSecurity` 对你的配置的注释。

> Spring Security 提供配置使用 Spring MVC's WebMvcConfigurerAdapter. 这意味着，如果你使用的是更高级的选项，如 `WebMvcConfigurationSupport` ，那么你将需要手动提供 Spring Security 配置.

MvcRequestMatcher

Spring Security 提供如何深度整合Spring MVC 匹配的网址和 `MvcRequestMatcher`. 这是有帮助的，以确保您的Security规则匹配用于处理您的请求的逻辑.

> 它总是建议提供授权规则匹配的`HttpServletRequest`和方法的安全性.
>
> 通过匹配提供授权规则`HttpServletRequest`是很好的，因为它很早就发生在代码路径，https://en.wikipedia.org/wiki/Attack_surface[attack surface].方法安全性确保如果有人绕过了Web权限规则，您的应用程序仍然是安全的. 这是你该知道的 Defence in Depth

考虑一个控制器映射如下:

```
 @RequestMapping("/admin")
public String admin() {
```

如果我们想通过限制访问该控制器的方法来管理用户, 一个开发人员可以通过匹配的 `HttpServletRequest` 得到以下的:

```
 protected configure(HttpSecurity http) throws Exception {
 http
  .authorizeRequests()
   .antMatchers("/admin").hasRole("ADMIN");
}
```

或者在xml中

```
 <http>
 <intercept-url pattern="/admin" access="hasRole('ADMIN')"/>
</http>
```

任何配置URL `/admin` 将经过身份验证的用户作为管理用户. 然而, 这取决于我们 Spring MVC配置, URL `/admin.html` 也会告诉我们 `admin()` 方法.

问题是，我们的安全规则只是保护 `/admin`. 我们可以为所有的排列添加额外的规则 Spring MVC,但这将是相当冗长而乏味的.

相反，我们可以利用Spring Security's `MvcRequestMatcher`. 下面的配置会保护，Spring MVC将匹配利用Spring MVC匹配URL的URL。

```
 protected configure(HttpSecurity http) throws Exception {
 http
  .authorizeRequests()
   .mvcMatchers("/admin").hasRole("ADMIN");
}
```

在XML中：

```
  <http request-matcher="mvc">
 <intercept-url pattern="/admin" access="hasRole('ADMIN')"/>
</http>
```

@AuthenticationPrincipal

Spring Security provides `AuthenticationPrincipalArgumentResolver` 能解决问题 `Authentication.getPrincipal()` 对于 Spring MVC 争论. 通过使用 `@EnableWebSecurity` 您将自动将此添加到您的 Spring MVC 配置. 如果你使用 XML 基于你的配置, 你必须自己添加这个。例如:

```
  <mvc:annotation-driven>
   <mvc:argument-resolvers>
     <bean class="org.springframework.security.web.method.annotation.AuthenticationPrincipalArgumentResolver" />
   </mvc:argument-resolvers>
</mvc:annotation-driven>
```

一旦 `AuthenticationPrincipalArgumentResolver` 是正确配置的，您可以完全完成 Spring Security 在Spring MVC层。

考虑一个自定义的情况 `UserDetailsService` 返回一个 `Object` 实现 `UserDetails` 你自己的 `CustomUser` `Object`. 和 `CustomUser` 当前已验证的用户可以使用以下代码访问:

```
  @RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser() {
 Authentication authentication =
 SecurityContextHolder.getContext().getAuthentication();
 CustomUser custom = (CustomUser) authentication == null ? null : authentication.getPrincipal();

 // .. find messags for this user and return them ...
}
```

至于 Spring Security 3.2 我们可以更直接地通过添加注释来解决这个问题。例如:

```
  import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser) {

 // .. find messags for this user and return them ...
}
```

有时，它可能是必要的，以某种方式来改造. 比如, 如果 `CustomUser` 需要是最终它不能被扩展。 在这种情况下 `UserDetailsService` 可能会返回一个 `Object` 实现 `UserDetails` 并提供了一个命名的方法 `getCustomUser` 访问 `CustomUser`. 比如，他看起来是这个样子的:

```
  public class CustomUserUserDetails extends User {
  // ...
  public CustomUser getCustomUser() {
    return customUser;
  }
}
```

然后，我们可以访问 `CustomUser` 使用 SpEL expression 使用 `Authentication.getPrincipal()` 作为根对象:

```
  import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal(expression = "customUser") CustomUser customUser) {

 // .. find messags for this user and return them ...
}
```

我们可以进一步消除我们的依赖 Spring Security通过标记 `@AuthenticationPrincipal` 我们有我们自己的元注释注释. 下面，我们展示了如何我们可以这样做的注释命名 `@CurrentUser`.

NOTE:重要的是要认识到消除依赖 Spring Security,它是将创建的消耗应用程序 `@CurrentUser`. 这一步不是严格要求，但有助于隔离你的依赖 Spring Security 到一个更重要的位置.

```
  @Target({ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@AuthenticationPrincipal
public @interface CurrentUser {}
```

现在 `@CurrentUser` 已指定, 我们可以用它来信号来解决我们的 `CustomUser` 当前已验证的用户孤立了我们依赖 Spring

Security 到一个单一的文件.

```
 @RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@CurrentUser CustomUser customUser) {

 // .. find messags for this user and return them ...
}
```

Spring MVC 异步集成

Spring Web MVC 3.2+ 有极好的支持 Asynchronous Request Processing. 没有额外的配置, Spring Security 将自动设置 `SecurityContext` 到 `Thread` 执行 `Callable` 由你的控制器返回. 例如，下面的方法将自动有它的 `Callable` 执行的 `SecurityContext` 这是可用的，当 `Callable` 被创建:

```
 @RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

return new Callable<String>() {
 public Object call() throws Exception {
 // ...
 return "someView";
 }
};
}
```

Associating SecurityContext to Callable's

从技术上讲, Spring Security 结合 `WebAsyncManager`. `SecurityContext` 这是用来处理 `Callable` 是 `SecurityContext` 存在于 `SecurityContextHolder` 此时 `startCallableProcessing` 被调用.

没有一个自动集成与 `DeferredResult` 由控制器返回. 这是因为 `DeferredResult` 是由用户处理的，因此没有自动整合的方式。 当然, 你依然可以用 Concurrency Support提供透明的集成与 Spring Security.

Spring MVC 和 CSRF 整合

自动令牌包

Spring Security 将自动 include the CSRF Token 使用形式为 Spring MVC form tag. 比如说, 下面的 JSP:

```
 <jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
 xmlns:c="http://java.sun.com/jsp/jstl/core"
 xmlns:form="http://www.springframework.org/tags/form" version="2.0">
 <jsp:directive.page language="java" contentType="text/html" />
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
 <!-- ... -->

 <c:url var="logoutUrl" value="/logout"/>
 <form:form action="${logoutUrl}"
  method="post">
 <input type="submit"
  value="Log out" />
 <input type="hidden"
  name="${_csrf.parameterName}"
  value="${_csrf.token}"/>
 </form:form>

 <!-- ... -->
</html>
</jsp:root>
```

将输出HTML，类似于下面的代码:

```
 <!-- ... -->

<form action="/context/logout" method="post">
<input type="submit" value="Log out"/>
<input type="hidden" name="_csrf" value="f81d4fae-7dec-11d0-a765-00a0c91e6bf6"/>
</form>

<!-- ... -->
```

解决 CsrfToken

Spring Security 提供 `CsrfTokenArgumentResolver` 就能自动解决 `CsrfToken` Spring MVC 争论. 通过使用 @EnableWebSecurity 您将自动将此添加到您的 Spring MVC 配置. 如果你使用基于XML的配置，您必须自己手动添加.

一旦 `CsrfTokenArgumentResolver` 是正确配置的，你可以暴露 `CsrfToken` 你的静态HTML为基础的应用.

```
 @RestController
public class CsrfController {

 @RequestMapping("/csrf")
```

```
 public CsrfToken csrf(CsrfToken token) {
  return token;
 }
}
```

重要的是要保持 `CsrfToken` 从其他域的一个秘密. 这意味着如果你正在使https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS[Cross Origin Sharing (CORS)], 你应该 **NOT** 暴露 `CsrfToken` 到任何外部域.

Spring Data 整合

Spring Security 提供 Spring Data整合允许在您的查询中引用当前的用户. 它很有用，但要包括在查询用户支持分页结果由于过滤后，结果规模不会减少。

Spring 数据 & Spring 安全配置

要使用这种支持，提供一个类型的 `SecurityEvaluationContextExtension`. 在Java配置中, 他看起来会是这个样子:

```
  @Bean
public SecurityEvaluationContextExtension securityEvaluationContextExtension() {
 return new SecurityEvaluationContextExtension();
}
```

在XMl配置中, 他看起来会是这个样子:

```
  <bean class="org.springframework.security.data.repository.query.SecurityEvaluationContextExtension"/>
```

安全的表达 @Query

如今 Spring Security 可以在您的查询中使用，例如：

```
  @Repository
public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {
 @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
 Page<Message> findInbox(Pageable pageable);
}
```

检查 `Authentication.getPrincipal().getId()` 看看是否 平等于容器 `Message`. 请注意，这个示例假定您已定制了一个对象的，该对象是具有一个身份属性. 通过暴露 `SecurityEvaluationContextExtension` bean, 所有的Common Security Expressions 可用在 Query.

Appendix

安全数据库模式

有使用的框架和本附录提供了一个单一的参考点和给 他们所有的不同的数据库模式. 你只需要为你需要的部分提供functonality表.

DDL声明给出了 HSQLDB 数据库.您可以使用这些作为定义您正在使用的数据库的架构的指导方针.

用户模式

JDBC 的标准实现了 `UserDetailsService` （`JdbcDaoImpl`）需要表加载密码, 帐户状态(可用或者不可用)和权力名单 (角色) 给用户. 您将需要调整此架构以与您正在使用的数据库语言相匹配.

```
  create table users(
 username varchar_ignorecase(50) not null primary key,
 password varchar_ignorecase(50) not null,
 enabled boolean not null
);

create table authorities (
 username varchar_ignorecase(50) not null,
 authority varchar_ignorecase(50) not null,
 constraint fk_authorities_users foreign key(username) references users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

集团机关

Spring Security 2.0 介绍了当前组支持 `JdbcDaoImpl`.如果组启用表，结构如下. 您将需要调整此架构以与您正在使用的数据库语言相匹配.

```
 create table groups (
 id bigint generated by default as identity(start with 0) primary key,
 group_name varchar_ignorecase(50) not null
);

create table group_authorities (
 group_id bigint not null,
 authority varchar(50) not null,
 constraint fk_group_authorities_group foreign key(group_id) references groups(id)
);

create table group_members (
 id bigint generated by default as identity(start with 0) primary key,
 username varchar(50) not null,
 group_id bigint not null,
 constraint fk_group_members_group foreign key(group_id) references groups(id)
);
```

请记住，这些表只需要由您使用所提供的JDBC `UserDetailsService` 实施. 如果你自己写或选择实施 `AuthenticationProvider` 没有 `UserDetailsService`,然后只要接口的合同是满意的，你有完整的自由来存储的数据。

持续登录 (记住我) Schema

此表用于存储更安全的使用的数据 persistent token remember-me 实施. 如果你正在使用 `JdbcTokenRepositoryImpl` 无论是直接或通过命名空间，您都将需要此表. 请记住调整此架构以与您正在使用的数据库语言相匹配.

```
 create table persistent_logins (
 username varchar(64) not null,
 series varchar(64) primary key,
 token varchar(64) not null,
 last_used timestamp not null
);
```

ACL Schema

有四个表所使用 Spring Security ACL 实施.

1. `acl_sid` Store身份确认和安全通过ALC系统来保障， 这些可以适用于唯一的或者多个负责人。
2. `acl_class` 定义域的对象类型的ACL应用。 `class` 列储存java类对象名称。
3. `acl_object_identity` 储存的具体领域对象的标识的定义.
4. `acl_entry` 储存的ACL权限适用于一个特定的对象标识和安全标识。

它假定数据库将自动生成每个身份的主键。 `JdbcMutableAclService` 必须能够检索这些时，它已创建了一个新的行 `acl_sid` 或者 `acl_class` 表。它有两个属性定义需要检索这些值的SQL `classIdentityQuery` 和 `sidIdentityQuery`. 这两个默认 `call identity()`

ACL加工创建JRC包含在hypersql创建ACL模式的文件 (HSQLDB), PostgreSQL, MySQL/MariaDB, Microsoft SQL Server, and Oracle 数据库。 这些架构也被证明在以下几个部分.

HyperSQL

默认架构的工作，采用的是单元测试的框架内嵌入的HSQLDB数据库.

```
 create table acl_sid(
 id bigint generated by default as identity(start with 100) not null primary key,
 principal boolean not null,
 sid varchar_ignorecase(100) not null,
 constraint unique_uk_1 unique(sid,principal)
);

create table acl_class(
 id bigint generated by default as identity(start with 100) not null primary key,
 class varchar_ignorecase(100) not null,
 constraint unique_uk_2 unique(class)
);

create table acl_object_identity(
 id bigint generated by default as identity(start with 100) not null primary key,
 object_id_class bigint not null,
 object_id_identity bigint not null,
 parent_object bigint,
 owner_sid bigint,
 entries_inheriting boolean not null,
 constraint unique_uk_3 unique(object_id_class,object_id_identity),
```

```
 constraint foreign_fk_1 foreign key(parent_object)references acl_object_identity(id),
 constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
 constraint foreign_fk_3 foreign key(owner_sid)references acl_sid(id)
);

create table acl_entry(
 id bigint generated by default as identity(start with 100) not null primary key,
 acl_object_identity bigint not null,
 ace_order int not null,
 sid bigint not null,
 mask integer not null,
 granting boolean not null,
 audit_success boolean not null,
 audit_failure boolean not null,
 constraint unique_uk_4 unique(acl_object_identity,ace_order),
 constraint foreign_fk_4 foreign key(acl_object_identity) references acl_object_identity(id),
 constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);
```

PostgreSQL

```
 create table acl_sid(
 id bigserial not null primary key,
 principal boolean not null,
 sid varchar(100) not null,
 constraint unique_uk_1 unique(sid,principal)
);

create table acl_class(
 id bigserial not null primary key,
 class varchar(100) not null,
 constraint unique_uk_2 unique(class)
);

create table acl_object_identity(
 id bigserial primary key,
 object_id_class bigint not null,
 object_id_identity bigint not null,
 parent_object bigint,
 owner_sid bigint,
 entries_inheriting boolean not null,
 constraint unique_uk_3 unique(object_id_class,object_id_identity),
 constraint foreign_fk_1 foreign key(parent_object)references acl_object_identity(id),
 constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
 constraint foreign_fk_3 foreign key(owner_sid)references acl_sid(id)
);

create table acl_entry(
 id bigserial primary key,
 acl_object_identity bigint not null,
 ace_order int not null,
 sid bigint not null,
 mask integer not null,
 granting boolean not null,
 audit_success boolean not null,
 audit_failure boolean not null,
 constraint unique_uk_4 unique(acl_object_identity,ace_order),
 constraint foreign_fk_4 foreign key(acl_object_identity) references acl_object_identity(id),
 constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);
```

你将不得不分别地设置 `classIdentityQuery` 和 `sidIdentityQuery` 性能 `JdbcMutableAclService` 以下的值，：

- `select currval(pg_get_serial_sequence('acl_class', 'id'))`
- `select currval(pg_get_serial_sequence('acl_sid', 'id'))`

MySQL and MariaDB

```
 CREATE TABLE acl_sid (
 id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 principal BOOLEAN NOT NULL,
 sid VARCHAR(100) NOT NULL,
 UNIQUE KEY unique_acl_sid (sid, principal)
) ENGINE=InnoDB;

CREATE TABLE acl_class (
 id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 class VARCHAR(100) NOT NULL,
 UNIQUE KEY uk_acl_class (class)
) ENGINE=InnoDB;

CREATE TABLE acl_object_identity (
 id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 object_id_class BIGINT UNSIGNED NOT NULL,
 object_id_identity BIGINT NOT NULL,
 parent_object BIGINT UNSIGNED,
 owner_sid BIGINT UNSIGNED,
```

```
 entries_inheriting BOOLEAN NOT NULL,
 UNIQUE KEY uk_acl_object_identity (object_id_class, object_id_identity),
 CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
 CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;

CREATE TABLE acl_entry (
 id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 acl_object_identity BIGINT UNSIGNED NOT NULL,
 ace_order INTEGER NOT NULL,
 sid BIGINT UNSIGNED NOT NULL,
 mask INTEGER UNSIGNED NOT NULL,
 granting BOOLEAN NOT NULL,
 audit_success BOOLEAN NOT NULL,
 audit_failure BOOLEAN NOT NULL,
 UNIQUE KEY unique_acl_entry (acl_object_identity, ace_order),
 CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;
```

Microsoft SQL Server

```
 CREATE TABLE acl_sid (
 id BIGINT NOT NULL IDENTITY PRIMARY KEY,
 principal BIT NOT NULL,
 sid VARCHAR(100) NOT NULL,
 CONSTRAINT unique_acl_sid UNIQUE (sid, principal)
);

CREATE TABLE acl_class (
 id BIGINT NOT NULL IDENTITY PRIMARY KEY,
 class VARCHAR(100) NOT NULL,
 CONSTRAINT uk_acl_class UNIQUE (class)
);

CREATE TABLE acl_object_identity (
 id BIGINT NOT NULL IDENTITY PRIMARY KEY,
 object_id_class BIGINT NOT NULL,
 object_id_identity BIGINT NOT NULL,
 parent_object BIGINT,
 owner_sid BIGINT,
 entries_inheriting BIT NOT NULL,
 CONSTRAINT uk_acl_object_identity UNIQUE (object_id_class, object_id_identity),
 CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
 CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
);

CREATE TABLE acl_entry (
 id BIGINT NOT NULL IDENTITY PRIMARY KEY,
 acl_object_identity BIGINT NOT NULL,
 ace_order INTEGER NOT NULL,
 sid BIGINT NOT NULL,
 mask INTEGER NOT NULL,
 granting BIT NOT NULL,
 audit_success BIT NOT NULL,
 audit_failure BIT NOT NULL,
 CONSTRAINT unique_acl_entry UNIQUE (acl_object_identity, ace_order),
 CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
);
```

Oracle Database

```
 CREATE TABLE acl_sid (
 id NUMBER(38) NOT NULL PRIMARY KEY,
 principal NUMBER(1) NOT NULL CHECK (principal in (0, 1)),
 sid NVARCHAR2(100) NOT NULL,
 CONSTRAINT unique_acl_sid UNIQUE (sid, principal)
);
CREATE SEQUENCE acl_sid_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_sid_id_trigger
 BEFORE INSERT ON acl_sid
 FOR EACH ROW
BEGIN
 SELECT acl_sid_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_class (
 id NUMBER(38) NOT NULL PRIMARY KEY,
 class NVARCHAR2(100) NOT NULL,
 CONSTRAINT uk_acl_class UNIQUE (class)
);
CREATE SEQUENCE acl_class_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_class_id_trigger
 BEFORE INSERT ON acl_class
 FOR EACH ROW
BEGIN
```

```
 SELECT acl_class_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_object_identity (
 id NUMBER(38) NOT NULL PRIMARY KEY,
 object_id_class NUMBER(38) NOT NULL,
 object_id_identity NUMBER(38) NOT NULL,
 parent_object NUMBER(38),
 owner_sid NUMBER(38),
 entries_inheriting NUMBER(1) NOT NULL CHECK (entries_inheriting in (0, 1)),
 CONSTRAINT uk_acl_object_identity UNIQUE (object_id_class, object_id_identity),
 CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
 CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
);
CREATE SEQUENCE acl_object_identity_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_object_identity_id_trigger
 BEFORE INSERT ON acl_object_identity
 FOR EACH ROW
BEGIN
 SELECT acl_object_identity_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_entry (
 id NUMBER(38) NOT NULL PRIMARY KEY,
 acl_object_identity NUMBER(38) NOT NULL,
 ace_order INTEGER NOT NULL,
 sid NUMBER(38) NOT NULL,
 mask INTEGER NOT NULL,
 granting NUMBER(1) NOT NULL CHECK (granting in (0, 1)),
 audit_success NUMBER(1) NOT NULL CHECK (audit_success in (0, 1)),
 audit_failure NUMBER(1) NOT NULL CHECK (audit_failure in (0, 1)),
 CONSTRAINT unique_acl_entry UNIQUE (acl_object_identity, ace_order),
 CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
 CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
);
CREATE SEQUENCE acl_entry_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_entry_id_trigger
 BEFORE INSERT ON acl_entry
 FOR EACH ROW
BEGIN
 SELECT acl_entry_sequence.nextval INTO :new.id FROM dual;
END;
```

安全空间

本附录提供了在安全命名空间中可用的元素以及它们创建的基础上的信息的元素的引用 (个别类的知识，和他们是如何一起工作的，假设你可以找到项目中的javadoc更多信息和本文档中的其他部分)。 如果您以前没有使用过命名空间，请阅读这部分。 introductory chapter 对命名空间的配置, 这是作为一个补充的信息。 使用一个很好的质量的XML编辑器，同时编辑一个基于架构的配置建议，这将提供有关的元素和属性的上下文信息，以及解释他们的目的。 命名空间是写在 RELAX NG 的紧凑的格式，后来转为XSD架构. 如果您对这种格式很熟悉，您可能希望检查它。 schema file 直接。

Web应用安全

<debug>

启用Spring Security调试基础设施。这将给人类提供可读的 (多线) 调试信息以监视安全筛选器的请求. 这可能包括敏感信息，如请求参数或头文件，并且只在开发环境中使用。

<http>

如果你能使用 `<http>` 在您的应用程序中的元素 `FilterChainProxy` bean 叫做 "springSecurityFilterChain"创建和元素内的配置是用来在内部建立一个过滤器链 `FilterChainProxy`. 作为Spring Security 3.1, 额外的 `http` 元素可以用来添加额外的过滤链脚注:[See the introductory chapter 为你建立映射 `web.xml`]. 一些核心过滤器总是在一个过滤器链中创建，其他一些核心过滤器将被添加到堆栈中，这取决于当前的属性和子元素. 标准过滤器的位置是固定的 (看 the filter order table 在命名空间介绍中，删除一个与以前版本的框架的常见错误源，当用户在该框架中明确地配置了过滤器链时 `FilterChainProxy` bean.如果你需要完全控制配置的话，你仍然可以这样做。

所需要参考所有过滤器 `AuthenticationManager` 将自动注入由命名空间配置创建的内部实例(看到introductory chapter 为更多的 `AuthenticationManager`).每个 `<http>` 命名空间块总是创建一个 `SecurityContextPersistenceFilter`,一个 `ExceptionTranslationFilter` 和一个 `FilterSecurityInterceptor`. 这些都是固定的，不能被替代品替代

<http> 属性

`<http>` 的属性元素控制核心过滤器上的一些属性。

- **access-decision-manager-ref** 可选属性指定的标识 `AccessDecisionManager` 实现应该用于授权的HTTP请求. 通过一个默认的 `AffirmativeBased` 实现用于与一个 `RoleVoter` 和 `AuthenticatedVoter`.

- **authentication-manager-ref** 参考的 `AuthenticationManager` 用来 `FilterChain` 通过这个HTTP元素创建。
- **auto-config** 自动注册登录表单，基本认证，注销服务。如果设置为“真”，所有这些功能都被添加 (虽然您仍然可以通过提供相应的元素来自定义每个元素的配置)。如果未指定，默认为“假”。不建议使用此属性。使用明确的配置元素，而不是避免混乱。
- **create-session** 控制一个Spring Security类的创建，包括以下:
  - `always` - Spring Security 如果不存在，将主动创建一个会话。
  - `ifRequired` -Spring Security只会创建一个会话，如果是必需的 (默认值)。
  - `never` - Spring Security 永远不会创建一个会话, 如果是应用程序它将会使用一个会话。
  - `stateless` - Spring Security 不会创建一个会话，忽略获得一个会话的会话Spring。 `Authentication`.
- **disable-url-rewriting** 防止会话标识被添加到应用程序中的网址。 客户端必须使用该属性设置为 `true`. 默认值是 `true`.
- **entry-point-ref** 通常情况下 `AuthenticationEntryPoint` 使用的将取决于已配置的身份验证机制。 此属性允许这种行为是通过定义一个定制的重写 `AuthenticationEntryPoint` bean 这将启动认证过程。
- **jaas-api-provision** 如果可用，运行请求作为 `Subject` 获得的 `JaasAuthenticationToken` 这是通过添加一个 `JaasApiIntegrationFilter` bean 到堆栈。默认值为 `false`.
- **name** 一个bean标识符，用于指在上下文中的其他地方的bean.
- **once-per-request** 对应于 `observeOncePerRequest` 财产 `FilterSecurityInterceptor`. 默认值为 `true`.
- **pattern** 定义一个模式 http 元素控制将被过滤的请求通过它定义的过滤器列表。解释是依赖于配置的 request-matcher. 如果没有定义模式，所有的请求都将被匹配，所以最具体的模式应该首先声明。
- **realm** 设置用于基本身份验证的域名称 (如果启用). 对应于 `realmName` 财产 `BasicAuthenticationEntryPoint`.
- **request-matcher** 定义 `RequestMatcher` 战略中使用的 `FilterChainProxy` 由 `intercept-url` 创建的bean匹配传入的请求。 选择目前 `mvc`, `ant`, `regex` 和 `ciRegex`, Spring MVC, ant, 正则表达式和不区分大小写的正则表达式。为每个创建一个单独的实例intercept-url 元素的使用 pattern 和 method 属性，Ant路径匹配使用 `AntPathRequestMatcher` 和正则表达式匹配使用 `RegexRequestMatcher`. 看到这些类Javadoc详情究竟如何进行匹配。Ant路径是默认策略。
- **request-matcher-ref** 一个参考到bean的实现 `RequestMatcher` 这将决定是否 `FilterChain` 应使用， 这是一个更强大的替代。 pattern.
- **security** 一个请求模式可以被映射到一个空的过滤链，通过设置这个属性 `none`. 没有安全将被应用，没有Spring Security的功能将是可用的。
- **security-context-repository-ref** 允许注入一个自定义 `SecurityContextRepository` 进入 `SecurityContextPersistenceFilter`.
- **servlet-api-provision** 提供的版本 `HttpServletRequest` 安全方法，如 `isUserInRole()` 和 `getPrincipal()` 这是通过添加一个 `SecurityContextHolderAwareRequestFilter` bean 到堆栈。默认值为 `true`.
- **use-expressions** 使能在 `access` 属性，如章节所述 expression-based access-control 默认值为真。

子元素 <http>

- access-denied-handler
- anonymous
- cors
- csrf
- custom-filter
- expression-handler
- form-login
- headers
- http-basic
- intercept-url

- jee
- logout
- openid-login
- port-mappings
- remember-me
- request-cache
- session-management
- x509

<access-denied-handler>

此元素允许您设置 `errorPage` 默认属性 `AccessDeniedHandler` 通过使用 `ExceptionTranslationFilter`,使用 error-page 属性，或使用该属性提供您自己的实现。ref属性。这会在更详细的章节中讨论。 ExceptionTranslationFilter.

父元素 <access-denied-handler>

- http

<access-denied-handler> Attributes

- **error-page** 访问被拒绝页，一个经过验证的用户将被重定向到他们请求的一个没有权限访问的页面。
- **ref** 定义一个引用 Spring bean 类型 `AccessDeniedHandler`.

<cors>

此元素允许配置 `CorsFilter`. 如果不 `CorsFilter` 或者 `CorsConfigurationSource` 是指定在Spring MVC类路径中, a `HandlerMappingIntrospector` 被用作 `CorsConfigurationSource`.

<cors> 属性

`<cors>` 的属性元素控制头元素。

- **ref** 指定一个指定的名称的属性的可选属性 `CorsFilter`.
- **ref** 指定一个指定的名称的属性的可选属性 `CorsConfigurationSource` 被注入到一个 `CorsFilter` 由XML命名空间创建。

父元素 <cors>

- http

<headers>

此元素允许配置额外的 (security) 将发送的标题与响应。 它可以方便地配置几个头文件，也允许通过设置自定义头 header，可以发现在 Security Headers 参考截面。

- `Cache-Control`, `Pragma`,和 `Expires` - 可以设置使用 cache-control元。这将确保浏览器不缓存您的安全页。
- `Strict-Transport-Security` - 可以设置使用 hsts元。这保证了浏览器自动请求未来要求HTTPS。
- `X-Frame-Options` - 可以设置使用 frame-options 元 X-Frame-Options 可以用来阻止clickjacking攻击。
- `X-XSS-Protection` -可以用来设置 xss-protection元 X-XSS-Protection 可以通过浏览器来做基本的控制。
- `X-Content-Type-Options` -可以用来设置 content-type-options元。 X-Content-Type-Options 标头防止Internet Explorer MIME嗅探响应从宣布的内容类型。当下载扩展时这也适用于谷歌浏览器。
- `Public-Key-Pinning` or `Public-Key-Pinning-Report-Only` -可以用来设置 hpkp 元。 这允许攻击者使用HTTPS的网站通过发布虚假证书的MIS或抵抗冒充。
- `Content-Security-Policy` 或 `Content-Security-Policy-Report-Only` -可以用来设置 content-security-policy element. Content Security Policy (CSP) 是一种机制，Web应用程序可以利用减轻内容注入漏洞，如跨站脚本（XSS）。

<headers> 属性

`<headers>` 的属性元控制头元素。

- **defaults-disabled** 可选属性，指定要禁用默认的Spring Security的HTTP响应头。默认是错误的 （包含默认标题）。
- **disabled** 可选属性，指定要禁用Spring Security的HTTP响应头。默认是错误的（标题是启用的）。

父元素 <headers>

- http

子元素 <headers>

- cache-control
- content-security-policy
- content-type-options
- frame-options
- header
- hpkp
- hsts
- xss-protection

<cache-control>

添加 `Cache-Control`，`Pragma`，和 `Expires` 标头，以确保浏览器不缓存您的安全页。

<cache-control> 属性

- **disabled** 指定是否禁用缓存控件。默认的错误。

父元素 <cache-control>

- headers

<hsts>

当可以添加时 Strict-Transport-Security 对任何安全请求的响应的标头。这允许服务器指示浏览器自动使用HTTPS为未来的要求。

<hsts> 属性

- **disabled** 指定是否禁用严格的传输安全性。默认的错误。
- **include-sub-domains** 指定区域应包括。默认为true。
- **max-age-seconds** 指定时间的主人应该是已知的最大量HSTS主机。默认一年。
- **request-matcher-ref** requestmatcher实例如果被用来确定标题应设置，HttpServletRequest.isSecure()的默认值为真。

父元素 <hsts>

- headers

<hpkp>

当能够添加 Public Key Pinning Extension for HTTP 对任何安全请求的响应的标头。这允许攻击者使用HTTPS的网站通过发布虚假证书的MIS或抵抗冒充。

<hpkp> 属性

- **disabled** 指定HTTP（HPKP）公有键应禁用。默认为true。
- **include-sub-domains** 指定区域应包括。默认的错误。
- **max-age-seconds** 设置公有键引脚头的最大年龄指令的值。默认60天。
- **report-only** 指定如果浏览器只应报告引脚验证失败。默认为true。
- **report-uri** 指定的URI，浏览器应该报告PIN验证失败。

父元素 <hpkp>

- headers

<pins>

引脚列表。

子元素 <pins>

- pin

<pin>

指定使用base64编码SPKI指纹值和加密哈希算法的属性。

<pin> 属性

- **algorithm** 密码散列算法。默认是SHA256。

父元素 <pin>

- pins

<content-security-policy>

当可以添加 Content Security Policy (CSP) 响应标头 CSP 是一种机制，Web应用程序可以利用减轻内容注入漏洞，如跨站脚本（XSS）。

<content-security-policy> 属性

- **policy-directives** 对于内容安全策略头文件的安全策略指令（S）或如果报表只设置为真，则只使用头文件的内容安全策略报告。
- **report-only** 设置为真，使内容安全策略报告只报告策略违规行为的标题。默认为假。

父元素 <content-security-policy>

- headers

<frame-options>

当可以添加 X-Frame-Options header的响应，这会使得浏览器做一些安全检查和预防 clickjacking 攻击。

<frame-options> 属性

- **disabled** 如果禁用, 且不包括X帧选项报头，将默认为错误。
- **policy**
  - `DENY` 页面不能被显示在一个框架中，无论该网站试图怎样做。当指定了帧选项策略时，这还是默认值。
  - `SAMEORIGIN` 该页面只能在同一个页面上的同一个页面的框架中显示。
  - `ALLOW-FROM origin` 该页只能显示在指定的原点的框中。

  换言之，如果你指定了拒绝，不仅将试图加载在一个框架中的页面失败时从其他网站加载，试图这样做会失败，从同一个网站加载。另一方面，如果你指定的sameorigin，你仍然可以使用在一个框架网页只要站点包括在一个框架是作为一个相同的服务页面。
- **strategy** 选择 `AllowFromStrategy` 来使用，当使用allow-from策略时。
  - `static` 使用一个单一的静态allow-from价值。该值可以通过 value属性.
  - `regexp` 如果他们被允许使用regelur表达验证传入的请求和。正则表达式可以通过 value 属性。 用于检索用于验证的值的请求参数可以被使用。 from-parameter.
  - `whitelist` 一个逗号分离含有允许域列表。逗号分隔的可以通过设置 value属性。 请求参数用于检索的值来验证可以指定使用的 from-parameter.
- **ref** 不是使用一个预定义的策略也可以使用一个自定义的` allowfromstrategy `。这个bean的引用可以通过ref属性指定。
- **value** 要使用的值是用allow-from的。 strategy.
- **from-parameter** 指定的请求参数的名称使用时，使用正则表达式或白名单的allow-from策略。

父元素 <frame-options>

- headers

<xss-protection>

添加 X-XSS-Protection header 协助防止响应 reflected / Type-1 Cross-Site Scripting (XSS) 攻击。 这不是一个对XSS攻击的充分保护！

<xss-protection> 属性

- **xss-protection-disabled** 不包括标题 reflected / Type-1 Cross-Site Scripting (XSS)的保护。

- **xss-protection-enabled** 明确使用eisable reflected / Type-1 Cross-Site Scripting (XSS) 的保护。
- **xss-protection-block** 当true和XSS启用保护是真的，增加了mode=block的标头。这表明该页面不应该被加载的浏览器。当false和XSS启用保护是真的，页面仍然会呈现一个反映时检测到攻击但反应将被修改以防止攻击。请注意，有时有办法绕过这种模式，它可以经常次使阻塞页面更可取。

父元素 <xss-protection>

- headers

<content-type-options>

添加 X-Content-Type-Options 标头随着NOSNIFF值响应。 disables MIME-sniffing对 IE8+谷歌浏览器扩展。

<content-type-options> 属性

- **disabled** 如果选择的指定类型无法使用，默认FALSE。

父元素 <content-type-options>

- headers

<header>

向响应中添加额外的头文件，需要指定名称和值。

<header-attributes> 属性

- **header-name** 标头`name`的名字。
- **value** `value`添加的标头。
- **ref** 引用自定义实现的`HeaderWriter`接口。

父元素 <header>

- headers

<anonymous>

添加一个`AnonymousAuthenticationFilter` to the stack and an `AnonymousAuthenticationProvider`. 需要的如果你正在使用`IS_AUTHENTICATED_ANONYMOUSLY`属性.

父元素 <anonymous>

- http

<anonymous> 属性

- **enabled** 使用默认的命名空间设置，匿名“身份验证”设施将自动启用。您可以使用此属性禁用它。
- **granted-authority** 应分配给匿名请求的授予权限。通常，这是用来分配匿名请求特定的角色，它可以随后被用于授权决策。如果未设置，默认为`ROLE_ANONYMOUS`.
- **key** 提供者和过滤器之间的密钥共享。这一般不需要设置。如果未设置，则默认为一个安全的随机生成的值。这意味着设置这个值可以提高启动时间，当使用匿名功能，因为安全的随机值可以需要一段时间才能产生。
- **username** 应分配给匿名请求的用户名。这允许被确定的主要，这可能是重要的日志记录和审计。如果未设置，默认为`anonymousUser`.

<csrf>将添加元素

```
http://en.wikipedia.org/wiki/Cross-site_request_forgery[Cross Site Request Forger (CSRF)] protection to the application.
It also updates the default RequestCache to only replay "GET" requests upon successful authentication. Additional
information can be found in the <<csrf,Cross Site Request Forgery (CSRF)>> section of the reference.
```

父元素 <csrf>

- http

<csrf>属性

- **disabled** 可选属性，指定要禁用的Spring Security的CSRF保护。默认值为假（CSRF保护功能）。这是强烈建议关闭CSRF保护功能。
- **token-repository-ref** 使用的csrftokenrepository。默认值是`HttpSessionCsrfTokenRepository`.

- **request-matcher-ref** requestmatcher实例被用来确定是否应使用CSRF。除了默认的是“GET”任何HTTP方法，“Trace”、“Head”、“Opitions”。

<custom-filter>

此元素用于将一个过滤器添加到筛选器链中。它不会创建任何额外的bean类，但被用来选择一个类型的bean `javax.servlet.Filter` 已在应用程序上下文中定义，完整的细节可以在Spring Security维护的筛选器链中的特定位置处添加。 namespace chapter.

父元素 <custom-filter>

- http

<custom-filter>属性

- **after** 过滤器自定义过滤后，应放置在链中。此功能只需要高级用户，希望混合自己的过滤器到安全过滤器链，并有一些知识的标准Spring Security过滤器。将筛选器名称映射到特定的Spring Security实现筛选器。
- **before** 过滤器应立即在此之前，将自定义筛选器放置在链中。
- **position** 如果你正在更换一个标准的过滤器。自定义筛选器应放置在链中的显式位置。
- **ref** 定义了一个实现的Spring bean的参考 `Filter`.

<expression-handler>

定义 `SecurityExpressionHandler` 如果启用基于表达式的访问控制，则将使用实例，如果默认实现（没有ACL支持）将用于不提供。

父元素 <expression-handler>

- global-method-security
- http
- websocket-message-broker

<expression-handler> 属性

- **ref** 定义了一个实现的Spring Security的参考 `SecurityExpressionHandler`.

<form-login>

用来添加一个 `UsernamePasswordAuthenticationFilter` 到过滤器堆栈和一个 `LoginUrlAuthenticationEntryPoint` 到应用程序的环境， If no attributes are supplied, a login page will be generated automatically at the URL如果没有提供属性，将在网址中自动生成登录页 "/login" 脚注:[此功能是真的只是提供了方便，并不只是用于生产 (在那里，一个视图技术将被选择，并可以用来提供一个自定义的登录页面，这个类 `DefaultLoginPageGeneratingFilter` 是 负责显示登录页面，将正常形式的登录和/或OpenID如果需要提供登录形式。] 该行为可以使用自定义 `<form-login>` Attributes.

父元素 <form-login>

- http

<form-login> 属性

- **always-use-default-target** 如果设置 `true`，t用户将始终从给定的值开始 default-target-url, 何到达登录页面。映射到 `alwaysUseDefaultTargetUrl` 财产 `UsernamePasswordAuthenticationFilter`. 默认值是 `false`.
- **authentication-details-source-ref** 参考一 `AuthenticationDetailsSource` 将被认证过滤器使用。
- **authentication-failure-handler-ref** 可以作为一种替代 authentication-failure-url, 给您一个身份验证失败后的导航流的完全控制权。该值应该是一个 `AuthenticationFailureHandler` bean在应用程序上下文中。
- **authentication-failure-url** 映射到 `authenticationFailureUrl` 属性 `UsernamePasswordAuthenticationFilter`. 定义浏览器将被重定向到登录失败的网址。默认值为 `/login?error`,自动登录页面生成器自动处理，用一个错误信息重新绘制登录页面。
- **authentication-success-handler-ref** 这可以作为一种替代default-target-url 和 always-use-default-target, 给您一个成功的认证后的导航流的完全控制。该值应该是一个 `AuthenticationSuccessHandler` bean在应用程序上下文中。默认情况下，实现 `SavedRequestAwareAuthenticationSuccessHandler` 使用和注入 default-target-url.
- **default-target-url** 映射到 `defaultTargetUrl` 特性 `UsernamePasswordAuthenticationFilter`. 如果没有设置，默认值是“/”（应用程序的根）。登录后，用户将被带到这个网址，只要他们没有被要求登录，且他们试图访问一个有抵押的资源，他们将被带到最初的请求的网址。

- **login-page** 应该用来显示登录页面的网址。映射到` loginformurl 特性 `LoginUrlAuthenticationEntryPoint`. 默认值为 "/login".
- **login-processing-url** 映射到 `filterProcessesUrl` 特性 `UsernamePasswordAuthenticationFilter`。默认值为 "/login"。
- **password-parameter** 包含密码的请求参数的名称。默认为“password”。
- **username-parameter** 包含用户名的请求参数的名称。默认为“username”。
- **authentication-success-forward-url** 映射到 `ForwardAuthenticationSuccessHandler` `authenticationSuccessHandler`特性 `UsernamePasswordAuthenticationFilter`。
- **authentication-failure-forward-url** 映射到 `ForwardAuthenticationFailureHandler` `authenticationFailureHandler` 特性 `UsernamePasswordAuthenticationFilter`.

<http-basic>

添加一个 `BasicAuthenticationFilter` 和 `BasicAuthenticationEntryPoint` 对配置。后者将只用于作为配置入口点，如果未启用基于窗体的登录。

父元素 <http-basic>

- http

<http-basic> 属性

- **authentication-details-source-ref** 参考 `AuthenticationDetailsSource`将使用认证过滤器。
- **entry-point-ref** 设置 `AuthenticationEntryPoint` 用来 `BasicAuthenticationFilter`.

<http-firewall> 元

实现的顶层元素 `HttpFirewall` 进入 `FilterChainProxy` 由命名空间创建，默认实现应该适用于大多数应用程序。

<http-firewall> 属性

- **ref** 定义了一个实现的Spring Bean的参考 `HttpFirewall`.

<intercept-url>

此元素用于定义应用程序和配置如何处理它们的“网址”模式的集合 ，它被用来构建 `FilterInvocationSecurityMetadataSource` 通过使用 `FilterSecurityInterceptor`. 它还负责配置 `ChannelProcessingFilter` 如果特定的URL需要通过HTTPS访问，例如。 当匹配指定的模式对传入的请求时，匹配是在声明的元素的顺序中完成的。所以最具体的匹配模式应该是第一个，最普遍的应该是最后一个。

父元素 <intercept-url>

- filter-security-metadata-source
- http

<intercept-url> 属性

- **access** 列出将要存储在该目录中的访问属性 `FilterInvocationSecurityMetadataSource` 用于定义的网址模式/方法组合。这应该是一个逗号分隔的安全配置属性（如角色名称）的列表。
- **filters** 只能采取“none”的值，这将导致任何匹配的请求，完全绕过Spring Security过滤器链。没有其他的` <HTTP> ` 配置将有影响的请求，将没有安全上下文可供其持续时间。在请求期间访问安全方法将失败。

> 此属性无效 filter-security-metadata-source

- **method** HTTP方法将使用与模式匹配传入的请求合并。如果省略，任何方法将匹配。如果一个相同的模式被指定，而没有一个方法，该方法特定的匹配将优先。
- **pattern** 定义了网址路径的模式。内容将取决于 `request-matcher` 从包含HTTP元素属性，所以将默认的Ant路径语法。
- **requires-channel** “http”或“https”取决于一个特定的URL模式应该访问的HTTP或HTTPS区别。 lternatively 值“any”可以用来当没有偏好。如果这个属性是存在于any `<intercept-url>` 元, 然后一个 `ChannelProcessingFilter` 将添加到筛选器堆栈中，并将其添加到应用程序上下文中的附加依赖项添加到。

如果一个 `<port-mappings>` 配置添加，他将被用于 `SecureChannelProcessor` 和 `InsecureChannelProcessor` beans 用来重定向到 HTTP/HTTPS端口.

此属性无效 filter-security-metadata-source

<jee>

添加到过滤器链j2eepreauthenticatedprocessingfilter提供集成容器认证。

父元素 <jee>

- http

<jee> 属性

- **mappable-roles** 逗号隔开要查询传入的HttpServletRequest消息的角色列表。
- **user-service-ref** 对用户的服务（或userdetailsservice bean）ID

<logout>

添加一个 `LogoutFilter` 到过滤器堆栈。这是配置一个 `SecurityContextLogoutHandler`。

父元素 <logout>

- http

<logout> 属性

- **delete-cookies** 当用户登录时，应删除的一个逗号分隔的名称列表。
- **invalidate-session** 映射一个 `invalidateHttpSession` 的 `SecurityContextLogoutHandler`. 默认值为“真”，所以，会话将被作废注销。
- **logout-success-url** 登录后用户将采取目标网址。默认值为 <form-login-login-page>/?logout (i.e. /login?logout)

  设置此属性将注入 `SessionManagementFilter` 和一个 `SimpleRedirectInvalidSessionStrategy` 配置属性值。当一个无效的会话ID提交，该战略将调用重定向到配置的URL。
- **logout-url** URL会造成注销 (i.e. 将由过滤器处理). 默认值为 "/logout".
- **success-handler-ref** 可用于提供一个实例 `LogoutSuccessHandler` 将被调用来控制日志记录后的导航。

<openid-login>

类似于 `<form-login>`,并且具有相同的属性. `login-processing-url` 的默认值是"/login/openid". `OpenIDAuthenticationFilter` 和 `OpenIDAuthenticationProvider` 将被注册. 后者需要去引用一个 `UserDetailsService`. 同样, 这样可以被 `id` 指定, 使用 `user-service-ref` 属性, 或者将在应用程序上下文中被自动定位。

<openid-login>的父元素

- http

<openid-login>的属性

- **Always-use-the-default-target** 用户应该总是被重定向到登录后的默认目标网址.
- **authentication-details-source-ref** 对将要使用身份验证过滤引用一个身份验证详细信息源
- **authentication-failure-handler-ref** 引用一个AuthenticationFailureHandler bean,应用于处理身份验证失败的请求.不应该使用与身份验证失败的链接组合,实施执行处理导航到后续的目标.
- **authentication-failure-url** 登录失败页面的网址.如果没有指定登录失败的网址, Spring Security 将自动创建一个失败的登录网址，当请求登录失败的网址时会开出登录错误和一个相应的过滤器.
- **authentication-success-forward-url** 在 `UsernamePasswordAuthenticationFilter` 属性中将 `ForwardAuthenticationSuccessHandler` 映射到 `authenticationSuccessHandler` .
- **authentication-failure-forward-url** 在 `UsernamePasswordAuthenticationFilter` 属性中将 `ForwardAuthenticationSuccessHandler` 映射到 `authenticationSuccessHandler` .
- **authentication-success-handler-ref** 引用一个AuthenticationSuccessHandler bean应用于处理一个成功的身份验

证请求.不应与组合使用. default-target-url (or always-use-default-target) 实现执行是处理导航到后续的目标.

- **default-target-url** 如果用户的前一个动作无法恢复，将被重定向到成功认证后的URL.通常如果用户访问登录页面没有首先要求安全操作,将会触发身份验证.如果未指定，默认为应用程序的根.
- **login-page** 登录页面URL. 如果没有指定登录的URL, Spring Security将自动创建一个登录网址和一个相应的过滤器，以呈现被请求的登录网址.
- **login-processing-url** 登录窗口被发布的网址.如果未被指定，它默认为登录.
- **password-parameter** 包含密码的请求参数的名称。默认为"password".
- **user-service-ref** 参考用户服务（或用户详细信息服务）Id
- **username-parameter** 包含用户名的请求参数的名称。默认为 "username".

<openid-login>的子元素

- attribute-exchange

<attribute-exchange>

`attribute-exchange` 元素定义属性列表应该从身份提供程序中请求. 在OpenID Support 命名空间配置章节的部分可以找出一个例子. 不只一个可以使用, 在这种情况下每个必须有 `identifier-match` 属性, 对所提供的OpenID标识符匹配包含一个正则表达式. 这允许从不同的供应商(Google, Yahoo etc)获取不同的属性列表 .

<attribute-exchange>的父元素

- openid-login

<attribute-exchange>属性

- **identifier-match** 当决定在身份验证过程中使用哪些属性交换配置，将要与所请求的标识进行比较的正则表达式.

<attribute-exchange>的子元素

- openid-attribute

<openid-attribute>

用于制造一个OpenID Ax属性 Fetch Request

<openid-attribute>的父元素

- attribute-exchange

<openid-attribute> 属性

- **count** 指定要返回的属性的数量。例如，返回3个电子邮件. 默认值是 1.
- **name** 指定要返回的属性的名称。例如，电子邮件.
- **required** 指定此属性是否被要求对操作，但如果操作不返回属性，则不出错。默认为false.
- **type** 指定属性类型。例如, http://axschema.org/contact/email. 查看你的操作的有效属性类型的文档.

<port-mappings>

默认情况下，实例 `portmapperimpl` 将被添加到配置中用于安全和不安全的URL重定向到.此元素可以选择地用于重写该类定义的默认映射. 每个 `<port-mapping>` 元素定义了一对 HTTP:HTTPS 端口. 默认的映射是80:443和8080:8443.重写这些的例子可以在 namespace introduction找到.

<port-mappings>的父元素

- http

<port-mappings>的子元素

- port-mapping

<port-mappings>

当强制重定向提供了一个地图的HTTP端口HTTPS端口方式.

<port-mappings>父元素

- port-mappings

<port-mappings> 属性

- **http** HTTP端口使用.
- **https** HTTPS端口使用.

<remember-me>

添加 `RememberMeAuthenticationFilter` 到堆栈.根据属性设置，将依次配置一个 `TokenBasedRememberMeServices`,一个 `PersistentTokenBasedRememberMeServices` 或用户指定实现 `RememberMeServices` .

<remember-me>的父元素

- http

<remember-me> 属性

- **authentication-success-handler-ref** 如果需要自定义导航,在 `RememberMeAuthenticationFilter` 上设置 `authenticationSuccessHandler` .在应用程序上下文中该值应该是个` authenticationsuccesshandler ` bean的名称.
- **data-source-ref** 引用一个 `DataSource` bean. 如果这样设置，`persistenttokenbasedremembermeservices ` 将使用和配置一个 jdbctokenrepositoryimpl `实例.
- **remember-me-parameter** 切换 remember-me认证请求的参数名.默认为"remember-me". 映射到"parameter"的 `AbstractRememberMeServices` 属性.
- **remember-me-cookie** cookie的名称为存储remember-me身份验证的令牌. 默认为 "remember-me". `AbstractRememberMeServices` 属性映射到"cookieName".
- **key** `AbstractRememberMeServices` 属性映射到"key" . ，以确保remember-me cookies 仅在一个应用程序脚注中有效:[这不影响使用 `PersistentTokenBasedRememberMeServices` , 令牌存储在服务器端.]. 如果这不是设置一个将产生的安全的随机值. 因为生成安全的随机值可能需要一段时间, 当我们使用remember me功能时，设置一个明确的值可以改善启动时间.
- **services-alias** 输出内部定义的 `RememberMeServices` 作为 bean 别名,允许它在应用程序上下文中被其它beans使用.
- **services-ref** 将被使用的过滤器允许` remembermeservices `完全控制实施. 值应该是在应用程序上下文中实现此接口的bean的"id". 如果注销过滤器在使用也应该执行 logouthandler `.
- **token-repository-ref** 配置一个 `PersistentTokenBasedRememberMeServices` 但是允许使用一个自定义的 `PersistentTokenRepository` bean.
- **token-validity-seconds** `AbstractRememberMeServices` 属性映射到 `tokenValiditySeconds` . 指定remember-me cookie在几秒钟内应是有效的. 默认情况下，有效期为14天.
- **use-secure-cookie** 建议remember-me cookies只有通过HTTPS才能提交和因此应该被标记为"secure". 默认情况下，如果登录请求的链接是安全的（应该是），将使用一个secure cookie. 如果你把这个属性设置为 `false` , secure cookies 将不会被使用 . 设置它为 `true` 将始终在cookie上设置安全标志.这 `AbstractRememberMeServices` 属性映射到 `useSecureCookie` .
- **user-service-ref** remember-me服务的实现需要访问 `UserDetailsService` , 因此，在应用程序上下文中必须要有一个定义.如果只有一个，它将被选择和自动使用通过命名空间配置.如果有多个实例,你可以指定一个bean `id` 明确的使用这个属性.

<request-cache> 元素

集将被 `ExceptionTranslationFilter` 使用的 `RequestCache` 实例去存储在调用 `AuthenticationEntryPoint` 前的请求信息.

<request-cache>父元素

- http

<request-cache> 属性

- **ref** 对Spring bean定义一个引用，它是一个 `RequestCache` .

<session-management>

通过添加一个 `SessionManagementFilter` 到过滤器栈，Session-management 相关功能时被实施的.

<session-management>父元素

- http

<session-management> 属性

- **invalid-session-url** 设置此属性将会注入 `SessionManagementFilter` 一个 `SimpleRedirectInvalidSessionStrategy` 配置属性值.当提交一个无效的ID, 该战略将会被调用到重定向配置的URL.
- **session-authentication-error-url** 定义的错误页面应该显示在sessionauthenticationstrategy引发异常的URL. 如果没有设置，未经授权的（401）错误代码将返回到客户端. 请注意，如果错误发生在一个基于窗体的登录,此属性不适用.在身份验证失败的网址将优先.
- **session-authentication-strategy-ref** 允许加入被 SessionManagementFilter使用的 SessionAuthenticationStrategy 实例.
- **session-fixation-protection** 说明会话固定保护的应用将在用户认证. 如果设置为"none", 将不会应用保护. "newSession" 将新建一个新的空会话, 只有Spring Security相关属性迁移. "migrateSession" 将创建一个新的会话，并将所有会话属性复制到新会话中. In Servlet 3.1 (Java EE 7) 新的容器, 指定“changesessionid”将保持现有的会话和使用容器提供的会话固定保护（HttpServletRequest # changesessionid()）.默认为“changesessionid在Servlet 3.1和新的容器, “migratesession”在大容器.如果“changesessionid”用于大容器,抛出一个异常.

  如果启用会话固定保护, ` sessionmanagementfilter 注入一个适当的配置 defaultsessionauthenticationstrategy `.看到这类Javadoc详情.

<session-management>子元素

- concurrency-control

<concurrency-control>

添加支持并发会话控制, 允许将限制放置在用户可以拥有的活动会话的数量上. 一个` concurrentsessionfilter 将被创建，并且一个 `ConcurrentSessionControlAuthenticationStrategy` 将会可用于 `SessionManagementFilter`. 如果已声明 `form-login` 元素,则该策略对象也将被注入到所创建的验证筛选器中.一个 `SessionRegistry` (一个除非用户希望使用一个自定义bean的 `SessionRegistryImpl` 实例)将被创建供使用的战略实例.

<concurrency-control>父元素

- session-management

<concurrency-control>属性

- **error-if-maximum-exceeded** 如果设置为"true"，当用户试图超过允许的最大会话数时， `SessionAuthenticationException` 将会被引发. 这默认的行为是使一个原始的会话失效.
- **expired-url** 如果他们试图使用一个已被并发会话控制器"expired"的会话，URL用户将被重定向，因为用户已经超过了允许的会话数，并在其他地方再次登录. 除非 `exception-if-maximum-exceeded` 被设置.如果没有提供任何值，一个有效消息将直接返回到响应.
- **max-sessions** `ConcurrentSessionControlAuthenticationStrategy` 属性映射到 `maximumSessions`.
- **session-registry-alias** 他也可以是有用的，对内部会话注册表有一个参考，用于在你自己的beans或管理接口. 使用 `session-registry-alias` 属性，可以使内部bean公开, 给它一个名称，你可以在你的配置中的其他地方使用.
- **session-registry-ref** 用户可以提供自己的` sessionregistry 实现使用`session-registry-ref` 属性. 其他并发会话控制beans将被连接起来使用它.

<x509>

增加了支持X.509认证. 一个 `X509AuthenticationFilter` 将被添加到堆栈和 `Http403ForbiddenEntryPoint` bean 被创建. 后者只会在没有其他认证机制的时候使用（它唯一的功能是返回一个HTTP 403错误代码）. ` preauthenticatedauthenticationprovider`也将创建一个将用户权限加载到一个` userdetailsservice `的地方中.

<x509>父元素

- http

<x509> 属性

- **authentication-details-source-ref** 引用一个 `AuthenticationDetailsSource`
- **subject-principal-regex** 定义一个正则表达式，该表达式将用于从证书中提取用户名 (用于使用 `UserDetailsService`).
- **user-service-ref** 允许使用特定的 `UserDetailsService`X.509的情况下配置多个实例.如果没有设置，将试图自动找到一

个合适的实例，并使用.

<filter-chain-map>

用于显式配置filterchainproxy与filterchainmap实例

<filter-chain-map> 属性

- **request-matcher** 定义策略用于匹配传入请求的使用. 目前的选项是'ant'（蚂蚁路径模式），'regex'正则表达式和'ciRegex' 不区分大小写的正则表达式.

<filter-chain-map>子元素

- filter-chain

<filter-chain>

用于定义一个特定的URL模式和适用于该模式匹配的URL的过滤器列表.当多个filter-chain元素组合在一个列表中为了配置FilterChainProxy,最具体的模式必须放在列表的顶部，在底部最一般的模式.

<filter-chain>父元素

- filter-chain-map

<filter-chain> 属性

- **filters** 一个逗号分隔的列表引用Spring bean实现 `Filter`. 值"none"意味着没有 `Filter` 应该用于` FilterChain `.
- **pattern** A-pattern创造结合RequestMatcher request-matcher
- **request-matcher-ref** 引用一个` requestmatcher `将用于确定是否有一些`Filter`来自 `filters` 应该被调用的属性.

<filter-security-metadata-source>

用于显式配置FilterSecurityMetadataSource FilterSecurityInterceptor bean使用. 如果你正在配置FilterChainProxy,只需要明确的这一个，而不是使用<http>元素.截取只包含方法和访问属性模式的intercept-url.任何其他的将导致配置错误.

<filter-security-metadata-source> 属性

- **id** 一个bean标识符, 用于指在上下文中的其他地方的bean.
- **lowercase-comparisons** 比较后迫使小写
- **request-matcher** 定义用于匹配传入请求的策略.目前的选择是 'ant'（蚂蚁路径模式），'regex' 正则表达式和'ciRegex' 不区分大小写的正则表达式.
- **use-expressions** 可以使用在<intercept-url>元素中表达式'access'的属性,而不是传统的配置属性的列表.默认为 'true'. 如果启用，每个属性应该包含一个单一的布尔表达式. 如果表达式计算结果为'true'，则访问将被授予.

<filter-security-metadata-source>子元素

- intercept-url

WebSocket 安全

Spring Security 4.0+为授权消息提供支持.这是一个为WebSocket基础应用程序提供授权有用的例子.

<websocket-message-broker>

websocket-message-broker元素有两种不同的模式. 如果websocket-message-broker@id 没有指定，那么它会做以下事情:

- 确保任何SimpAnnotationMethodMessageHandler AuthenticationPrincipalArgumentResolver注册作为一个自定义参数解析器. 这允许使用 `@AuthenticationPrincipal` 来解决当前的主要`Authentication`
- 确保securitycontextchannelinterceptor自动注册为clientinboundchannel. 这与用户填充SecurityContextHolder消息中被发现.
- 确保channelsecurityinterceptor与clientinboundchannel注册. 这允许授权规则指定的消息.
- 确保CsrfChannelInterceptor与clientInboundChannel注册.这将确保只有从原来的域的请求被启用. *确保CsrfTokenHandshakeInterceptor与WebSocketHttpRequestHandler, TransportHandlingSockJsService,或DefaultSockJsService注册.这保证了预期的csrftoken来自消息复制到WebSocket Session属性.

如果额外的控制是必要的，可以指定ID和channelsecurityinterceptor将分配给指定的ID. 所有的布线与Spring的消息传递基础设施可以手动完成的.这是比较麻烦的，但提供了更大的配置控制.

<websocket-message-broker>属性

- **id** 一个bean的标识符，用于指在channelsecurityinterceptor bean的上下文中的任何地方. 如果指定，Spring Security需要在Spring Messaging内明确的配置. 如果没有指定，Spring Security会自动整合与通讯基础设施,如<websocket-message-broker>描述的.

- **same-origin-disabled** 禁用要求CSRF令牌出现在Stomp headers(默认错误). 如果有必要让其他起源SockJS连接，更改默认是有用的.

<websocket-message-broker>子元素

- expression-handler
- intercept-message

<intercept-message>

定义消息的授权规则.

<intercept-message>父元素

- websocket-message-broker

<intercept-message> 属性

- **pattern** 一种目的基于匹配Message的蚁群模式.例如, "**/" matches any Message with a destination; "/admin/**" 匹配任何有一个以"/admin/**"开头为目的地的Message".

- **type** 要匹配的消息的类型. 在simpmessagetype有效值的定义 (i.e. CONNECT, CONNECT_ACK, HEARTBEAT, MESSAGE, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT, DISCONNECT_ACK, OTHER).

- **access** 用于保护信息的表达式.例如,"DenyAll"将拒绝访问所有的匹配信息;"permitall"将授予访问所有的匹配信息;"hasrole（'admin'）"需要当前用户拥有的角色"role_admin"匹配的信息.

认证服务

在Spring Security 3.0之前, `AuthenticationManager` 在内部是自动注册的. 现在，你必须使用 `<authentication-manager>` 元素明确的注册一个. 这将创建Spring Security的 `ProviderManager` 类的一个实例, 需要配置一个或多个` AuthenticationProvider `实例. 这些都可以使用命名空间提供的语法元素来创建，也可以是标准的bean定义，标记为除了使用 `authentication-provider` 元素的列表.

<authentication-manager>

每个Spring Security应用程序使用名称空间必须包括这个元素.它负责注册的` authenticationManager `为应用提供认证服务. 所有元素的创建 AuthenticationProvider `实例应该是这个子元素.

<authentication-manager> 属性

- **alias** 此属性允许你为你自己的配置中使用的内部实例定义别名. 它的用途是描述在namespace introduction.

- **erase-credentials** 如果设置为真, AuthenticationManager将试图清除任何凭据数据在返回的验证对象中, 一旦用户已被身份验证.实际上它映射到`eraseCredentialsAfterAuthentication` 属性的 `ProviderManager`. 这是在 Core Services 章节中讨论的.

- **id** 此属性允许你为内部实例定义一个用于在自己的配置中使用的标识. 它是相同的别名元素, 但提供了一个更加一致的经验使用id属性的元素.

<authentication-manager>子元素

- authentication-provider
- ldap-authentication-provider

<authentication-provider>

除非用` REF 属性，这个元素是缩写配置一个 `DaoAuthenticationProvider`. `DaoAuthenticationProvider` 加载用户信息从一个` userdetailsservice 和用户名/密码组合所提供的登录时比较. ` userdetailsservice `实例可以通过使用可用的命名空间元素定义 (`jdbc-user-service` 或者通过使用 `user-service-ref` 属性指向一个bean定义在应用程序上下文). 在 namespace introduction中你可以找到这些变化的例子.

<authentication-provider>父元素

- authentication-manager

<authentication-provider> 属性

- **ref** 定义引用一个Spring bean实现`AuthenticationProvider`.

如果你写了自己的 `AuthenticationProvider` 实现(或者想配置一个Spring Security的实现作为一个传统的bean,然后，你可以使用下面的语法将它添加到` providermanager ` 内部列表:

```
  <security:authentication-manager>
<security:authentication-provider ref="myAuthenticationProvider" />
</security:authentication-manager>
<bean id="myAuthenticationProvider" class="com.something.MyAuthenticationProvider"/>
```

- **user-service-ref** 引用bean实现UserDetailsService可以创建使用标准的bean元素或自定义 ser-service 元素.

<authentication-provider>子元素

- jdbc-user-service
- ldap-user-service
- password-encoder
- user-service

<jdbc-user-service>

创建一个JDBC-based UserDetailsService.

<jdbc-user-service> 属性

- **authorities-by-username-query** 一个SQL语句查询用户授予政府给定的用户名.

默认为

```
  select username, authority from authorities where username = ?
```

- **cache-ref** 定义引用UserDetailsService缓存.
- **data-source-ref** bean ID 提供所需的表的数据源.
- **group-authorities-by-username-query** 一条SQL语句来查询用户组部门给定的用户名.

默认为

+

```
  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
```

- **id** bean标识符, 用于bean在上下文的任何地方.
- **role-prefix** 从永久存储(默认是 "ROLE_")中，一个非空字符串前缀字符串将被添加到角色加载.在默认为非空的情况下，使用没有前缀的值"none".
- **users-by-username-query** 一个SQL语句查询用户名、密码，并启用了一个用户名状态. 默认为

```
  select username, password, enabled from users where username = ?
```

<password-encoder>

如namespace introduction所描述的，身份验证提供者可以被配置为使用一个密码编码器. 这将导致bean注入到相应的` passwordencoder `实例中，可能可能伴随 `SaltSource` bean提供盐散列值.

<password-encoder>父元素

- authentication-provider
- password-compare

<password-encoder>属性

- **base64** 一个字符串是否应该被Base64编码
- **hash** 定义用于用户密码的散列算法.我们强烈建议你不要使用MD4,因为它是一个非常弱的散列算法.
- **ref** 定义引用一个Spring bean 实现`PasswordEncoder`.

<password-encoder>子元素

- salt-source

<salt-source>

口令保护策略. 可以使用来自UserDetails对象的系统常数和属性.

<salt-source>父元素

- password-encoder

<salt-source>属性

- **ref** 定义引用一个Spring bean Id.
- **system-wide** 一个单一的值，将被用来作为一个密码编码器的salt.
- **user-property** UserDetails对象的属性将被用作salt通过密码编码器. 通常情况下，像"username"可能会被使用.

<user-service>

从属性文件或列表中的"user"子元素创建一个UserDetailsService内存. 内部转换为小写,允许用户名不区分大小写的查询,所以这个不应使用是否需要区分大小写.

<user-service> 属性

- **id** bean标识符,用于指bean在上下文的任何地方.
- **properties** 其中属性文件的位置，每一行的格式为

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

<user-service>子元素

- user

<user>

表示应用程序中的用户.

<user>父元素

- user-service

<user> 属性

- **authorities** 一个或多个authorities授予的用户. 用逗号隔开authorities (但没有空间).例如, "ROLE_USER,ROLE_ADMINISTRATOR"
- **disabled** 可以设置为"true"来标记一个帐户禁用，无法使用.
- **locked** 可以设置为"true"来标记一个帐户锁定，无法使用.
- **name** 分配给用户的用户名.
- **password** 分配给用户的密码. 如果相应的认证供应商支持哈希(记得设置"user-service"的 "hash"属性元素)，这可能是散列.此属性被省略的情况下,该数据将不会被用于身份验证.但仅用于访问authorities.如果省略，命名空间将生成一个随机值，防止其意外使用身份验证。不能为空.

方法安全性

<global-method-security>

这个元素的主要手段是在Spring Security beans中添加支持安全方法.可以通过使用注释担保（在接口或类定义）或通过定义一组的切入点作为子元素的方法,使用AspectJ语法.

<global-method-security> 属性

- **access-decision-manager-ref** 使用与 `AccessDecisionManager` 相同的安全方法配置网络安全, 但这可以使用此属性覆盖.默认情况下AffirmativeBased实现用于RoleVoter和AuthenticatedVoter.
- **authentication-manager-ref** 引用一个 `AuthenticationManager` ,应该用于方法安全性.
- **jsr250-annotations** 指定是否使用JSR-250样式属性(例如"RolesAllowed"). 这将需要javax.annotation.安全类在类路径中.将此设置为真也增加一个 `Jsr250Voter` 到 `AccessDecisionManager` ,所以你需要确保你这样做，如果你正在使用一个自定义的实现和想要使用这些注释.
- **metadata-source-ref** 外部 `methodsecuritymetadatasource` 实例可以提供优先于其他来源（如默认的注释）.

- **mode** 该属性可以设置为"AspectJ"指定AspectJ应该用来代替默认的Spring AOP. 方法安全性必须被来自`spring-security-aspects`组件`AnnotationSecurityAspect`编排.

需要注意的是AspectJ遵循Java接口上的注释不是继承的.这意味着定义接口的Security annotaitons方法将是不安全的. 相反,当在使用AspectJ时,你必须把Security annotation放在类上.

- **order** 允许建议"order"将被设置为方法安全拦截.
- **pre-post-annotations** 指定是否使用Spring Security的前后调用注释 (@PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize) 应该支持这个应用程序上下文。默认为"disabled".
- **proxy-target-class** 如果这是真的,将使用基于类的代理,而不是基于接口的代理.
- **run-as-manager-ref** 引用一个可选的`RunAsManager`实现可通过配置的`MethodSecurityInterceptor`
- **secured-annotations** 指定是否使用Spring Security的 @Secured annotations 启用应用程序上下文. 默认为 "disabled".

<global-method-security>子元素

- after-invocation-provider
- expression-handler
- pre-post-annotation-handling
- protect-pointcut

<after-invocation-provider>

通过`<global-method-security>`的命名空间，这个元素可以用来装饰一个`AfterInvocationProvider`用于维护的安全拦截器.你可以定义零个或多个`global-method-security`内的元素,在应用程序上下文中,每一个都有`ref`属性指向一个`AfterInvocationProvider` bean实例.

<after-invocation-provider>父元素

- global-method-security

<after-invocation-provider> 属性

- **ref** 定义引用一个Spring bean 实现`AfterInvocationProvider`.

<pre-post-annotation-handling>

允许默认表达式的机制来处理Spring Security的前后调用注释 (@PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize) 去完全取代. 仅适用于这些被启用了的注释.

<pre-post-annotation-handling>父元素

- global-method-security

<pre-post-annotation-handling>子元素

- invocation-attribute-factory
- post-invocation-advice
- pre-invocation-advice

<invocation-attribute-factory>

定义了PrePostInvocationAttributeFactory实例用于生成前后调用元数据注释的方法.

<invocation-attribute-factory>父元素

- pre-post-annotation-handling

<invocation-attribute-factory> 属性

- **ref** 定义引用一个 Spring bean Id.

<post-invocation-advice>

对于<pre-post-annotation-handling> 元素，定制`PostInvocationAdviceProvider`编号为`PostInvocationAuthorizationAdvice`.

<post-invocation-advice>父元素

- pre-post-annotation-handling

<post-invocation-advice> 属性

- **ref** 定义引用一个Spring bean Id.

<pre-invocation-advice>

对于<pre-post-annotation-handling>元素,定制 `PreInvocationAuthorizationAdviceVoter` 编号为 `PreInvocationAuthorizationAdviceVoter` .

<pre-invocation-advice>父元素

- pre-post-annotation-handling

<pre-invocation-advice>属性

- **ref** 定义引用一个Spring bean Id.

方法安全性的使用

`<protect-pointcut>` 而不是定义一个单独的方法或类基础的安全属性使用 `@Secured` 注释, 你可以跨整个定义横切安全约束集的方法和接口服务层中使用 `<protect-pointcut>` 元素.你可以在 namespace introduction中找到一个例子.

<protect-pointcut>父元素

- global-method-security

<protect-pointcut> 属性

- **access** 访问适用于所有的匹配方法的切入点的配置属性列表, 例如. "ROLE_A,ROLE_B"
- **expression** 一个AspectJ的表达式, 包括'execution'关键字. 例如, 'execution(int com.foo.TargetObject.countLength(String))' (没有引号).

<intercept-methods>

可以使用在bean内的定义去添加一个安全拦截器到bean可以使用在bean定义安全拦截器添加到bean并，设置访问bean的配置属性的方法.

<intercept-methods> 属性

- **access-decision-manager-ref** 可选AccessDecisionManager bean ID创建的方法安全性拦截器使用.

<intercept-methods>子元素

- protect

<method-security-metadata-source>

创建一个MethodSecurityMetadataSource 实例

<method-security-metadata-source>属性

- **id** bean标识符, 用于指向bean的上下文中.
- **use-expressions** 可以使用 <intercept-url>元素 'access'属性表达式,而不是传统的配置属性列表. 默认为 'false'. 如果启用了, 每个属性应该包含一个boolean表达式. 如果表达式计算结果为'true', 访问将被授予.

<method-security-metadata-source>子元素

- protect

<protect>

定义一个应用于它的受保护方法和访问控制配置属性. 我们强烈建议你不要将"protect"申明与提供的"global-method-security"的任何服务混合在一起.

<protect>父元素

- intercept-methods
- method-security-metadata-source

<protect> 属性

- **access** 一种适用于该方法的访问配置属性列表,例如. "ROLE_A,ROLE_B".
- **method** 方法名称

LDAP命名空间选项

LDAP是覆盖一些细节在its own chapter. 我们将扩大在一些解释的名称空间映射到Spring bean的选项.LDAP 广泛的实现使

用Spring LDAP,因此,一些熟悉该项目的API可能是有用的.

定义LDAP服务器使用

`<ldap-server>` 元素 该元素设置了一个 Spring LDAP `ContextSource` 用于其他 LDAP beans, 定义LDAP服务器的位置和其他信息(如用户名和密码,如果它不允许匿名访问) 用于连接到它.它也可以被用来创建一个嵌入式服务器进行测试. 两个选项的语法的细节是在 LDAP chapter. 实际的 `ContextSource` 实施是 `DefaultSpringSecurityContextSource` 延伸 Spring LDAP的 `LdapContextSource` 类. `manager-dn` 和 `manager-password` 属性映射到后面的 `userDn` 和 `password` 属性分别.

如果你的应用程序上下文中定义了一个服务器, 其他的 LDAP namespace-defined beans将自动使用. 否则, 你可以给这个元素一个 "id"属性和从其他命名空间bean引用它使用 `server-ref` 属性. 这实际上是 `ContextSource` 的bean`id`实例, 如果你想在其它传统的Spring beans使用它.

<ldap-server> 属性

- **id** bean标识符, 用于指向bean的上下文中.
- **ldif** 显式指定LDIF文件资源加载到嵌入式LDAP服务器.ldif应该是一个Spring资源模式(i.e. classpath:init.ldiff). 默认为 classpath*:*.ldiff
- **manager-dn** 用户名 (DN) 的 "manager" 用户标识 将用于验证(non-embedded) LDAP 服务. 如果省略, 将使用匿名访问.
- **manager-password** 密码管理器DN. 如果 manager-dn被指定，这是必需的.
- **port** 指定一个IP端口号码. 用于配置嵌入式LDAP服务器, 例如. 默认值为 33389.
- **root** 对于嵌入式LDAP服务器可选根后缀. 默认为 "dc=springframework,dc=org"
- **url** 指定LDAP服务器URL时不使用嵌入式LDAP服务器.

<ldap-authentication-provider>

这元素是速记 `LdapAuthenticationProvider` 实例的创建.默认情况下这将配置一个` bindauthenticator `实例和一个`DefaultAuthoritiesPopulator`.和所有命名空间身份验证提供程序一样, 它必须包含作为 `authentication-provider` 元素的子元素.

<ldap-authentication-provider>父元素

- authentication-manager

<ldap-authentication-provider> 属性

- **group-role-attribute** 包含角色名的LDAP属性名将用于Spring Security. 映射到`DefaultLdapAuthoritiesPopulator`的 `groupRoleAttribute` 属性. 默认到 "cn".
- **group-search-base** 组成员搜索. 映射 `DefaultLdapAuthoritiesPopulator`的 `groupSearchBase` 构造函数的参数. 默认为 "" (从根搜索).
- **group-search-filter** 组搜索过滤器. 映射到 `DefaultLdapAuthoritiesPopulator`的 `groupSearchFilter` 属性. 默认为 (uniqueMember={0}). 替换参数是用户的DN.
- **role-prefix** 一个非空字符串前缀将被添加到从持久性加载的角色字符串中. 映射到`DefaultLdapAuthoritiesPopulator`的 `rolePrefix` 属性. 默认为 "ROLE_".在默认为非空的情况下,使用没有前缀的值"none".
- **server-ref** 可选的服务器使用. 如果省略, 和一个默认注册的LDAP 服务 (使用没ID的 <ldap-server>), that server will be used.
- **user-context-mapper-ref** 允许用户定制的加载对象明确指定userdetailscontextmapper bean将被称为来自用户目录项的上下文信息
- **user-details-class** 允许用户输入的类被指定。如果设置，框架会尝试加载标准属性的定义的类的对象返回到userdetails
- **user-dn-pattern** 如果你的用户在目录中的一个固定的位置（即你可以直接从用户名DN不做目录搜索），您可以使用此属性直接映射到DN. 它直接映射到`userDnPatterns` 属性 `AbstractLdapAuthenticator`. 值是一个特定的模式用于构建用户的DN, 例如"uid={0},ou=people". The key "{0}" 必须存在，并将被替换的用户名.
- **user-search-base** 搜索用户搜索库. 默认为 "". 只使用一个 'user-search-filter'.

如果你需要执行搜索来定位目录中的用户,然后，您可以设置这些属性来控制搜索. `BindAuthenticator` 将配置一个 `FilterBasedLdapUserSearch` 和属性值直接映射到bean的构造函数的前两个参数. 如果这些属性没有设置并没有"user-dn-pattern"提供作为替代, 然后默认搜索值 `user-search-filter="(uid={0})"` 和 `user-search-base=""` 将被使用.

- **user-search-filter** LDAP筛选器用于搜索用户（可选）.例如"(uid={0})".被替换的参数是用户的登录名.

  如果你需要执行搜索来定位目录中的用户,然后，你可以设置这些属性来控制搜索.`BindAuthenticator` 将配置一个 `FilterBasedLdapUserSearch` 并且属性值将直接映射到bean构造函数的前两个参数. 如果还没有设置这些属性和没有提供给 `user-dn-pattern` 作为替代, 然后默认搜索值 `user-search-filter="(uid={0})"` 和`user-search-base=""`将被使用.

<ldap-authentication-provider>子元素

- password-compare

<password-compare>

作为子元素被用于 `<ldap-provider>` 与从` bindauthenticator 到 passwordcomparisonauthenticator `交换机的认证策略.

<password-compare>父元素

- ldap-authentication-provider

<password-compare> 属性

- **hash** 定义用于用户密码的散列算法.我们强烈建议不要使用MD4,因为它是一个非常弱的散列算法.
- **password-attribute** 包含用户密码的目录中的属性. 默认为 "userPassword".

<password-compare>子元素

- password-encoder

<ldap-user-service>

这个元素配置一个LDAP `UserDetailsService`. 使用的类 `LdapUserDetailsService` 是 `FilterBasedLdapUserSearch` 和 `DefaultLdapAuthoritiesPopulator`的组合. 它支持的属性如在`<ldap-provider>`中有相同的用法.

<ldap-user-service> 属性

- **cache-ref** 定义引用一个缓存用于 UserDetailsService.
- **group-role-attribute** 包含角色名的LDAP属性名将用于Spring Security. 默认为 "cn".
- **group-search-base** 组成员搜索.默认为"" (从根搜索).
- **group-search-filter** 组过滤搜索器. 默认为 (uniqueMember={0}). 替代参数是用户的DN.
- **id** bean标识符, 用于指向bean的上下文中.
- **role-prefix** 一个非空字符串前缀将被添加到从持久性加载的角色字符串中(例如. "ROLE_"). 在默认为非空的情况下,使用没有前缀的值 "none" .
- **server-ref** 可选服务器使用. 如果省略, 和一个被默认注册的LDAP服务器(使用 没有ID的<ldap-server>), 该服务器将被使用.
- **user-context-mapper-ref** 通过指定一个可加载的用户对象的显式定制一个UserDetailsContextMapper bean，从用户的目录条目调用上下文信息.
- **user-details-class** 允许用户输入指定的对象类.如果设置，该框架将会尝试加载标准属性的定义的类的对象返回到 UserDetails
- **user-search-base** 搜索用户搜索库. 默认为 "". 只使用一个 'user-search-filter'.
- **user-search-filter** LDAP筛选器用于搜索用户(可选).例如"(uid={0})".被替换的参数是用户的登录名.

Spring Security依赖关系

T他的附录提供了一个 Spring Security参考模块和附加的依赖关系,他们需要在运行中的应用程序的功能. 我们不包括只用来构建或测试Spring Security 本身的依赖关系. 也不包括了依赖关系所必需的外部依赖.

在项目网站上列出了所需的Spring 版本, 因此，Spring依赖之下 特定的版本被省略.注意，下面列出的一些作为“可选”依赖关系下可能仍然需要在一个Spring应用程序中的其他非安全功能.如果它们在大多数应用中使用也被列为"optional"的依赖关系，实际上可能不被标记为在项目的Maven POM文件.只有在某种意义上他们是"optional" 他们是“可选的”只有在某种意义上,你不需要他们,除非你是使用指定的功能.

其中一个模块依赖于另一个Spring Security模块,模块的非可选依赖关系取决于被假定为是必需的和没有单独列出的.

spring-security-core

使用Spring Security核心模块必须包含在任何项目中.

Table 4. Core Depenendencies

Dependency	版本	描述
aopalliance	1.0	所需的安全实现方法.
ehcache	1.6.2	要求 ehcache-based基于用户缓存实现（可选）.
spring-aop		基于Spring AOP的方法安全性
spring-beans		Spring配置所需
spring-expression		基于表达式所需的方法安全性（可选）
spring-jdbc		如果需要使用一个数据库来存储用户数据(可选).
spring-tx		如果需要使用一个数据库来存储用户数据(可选).
aspectjrt	1.6.10	如果需要使用AspectJ支持(可选).
jsr250-api	1.0	如果需要你使用JSR-250方法安全性注释（可选）.

spring-security-remoting

这个模块通常需要在Web应用程序中使用Servlet API.

Table 5. Remoting Dependencies

Dependency	版本	描述
spring-security-core		
spring-web		使用HTTP的客户所需的远程支持.

spring-security-web

这个模块通常需要在Web应用程序中使用Servlet API.

Table 6. Web Dependencies

Dependency	版本	描述
spring-security-core		
spring-web		广泛使用Spring web支持类.
spring-jdbc		需要基于JDBC的免登陆标记库（可选）.
spring-tx		需要通过免登陆持久标记库实现（可选）.

spring-security-ldap

这个模块是如果需要你使用LDAP认证.

Table 7. LDAP Dependencies

Dependency	版本	描述

spring-security-core

Dependency	版本	描述
spring-ldap-core	1.3.0	LDAP仅支持于Spring LDAP.
spring-tx		数据异常类是必需的.
apache-ds [13]	1.5.5	如果需要你使用嵌入式LDAP服务器（可选）.
shared-ldap	0.9.15	如果需要你使用嵌入式LDAP服务器（可选）.
ldapsdk	4.1	Mozilla LdapSDK.如果你使用OpenLDAP的密码策略功能，用于解码LDAP密码策略控制,例如.

spring-security-config

这个模块是如果需要你使用Spring Security命名空间的配置.

Table 8. Config Dependencies

Dependency	版本	描述
spring-security-core		
spring-security-web		如果需要你使用任何与Web相关的命名空间配置（可选）.
spring-security-ldap		如果需要你使用LDAP命名空间选项（可选）.
spring-security-openid		如果需要你使用OpenID认证(可选).
aspectjweaver	1.6.10	如果需要你使用命名空间的语法点的保护（可选）.

spring-security-acl

ACL模块.

Table 9. ACL Dependencies

Dependency	版本	描述
spring-security-core		
ehcache	1.6.2	如果需要实现使用基于ACL的Ehcache缓存 (optional if you are using your own implementation).
spring-jdbc		如果需要你使用默认的基于JDBC的aclservice (optional if you implement your own).
spring-tx		如果需要你使用默认的基于JDBC的aclservice AclService (optional if you implement your own).

spring-security-cas

CAS模块提供集成JA-SIG CAS.

Table 10. CAS Dependencies

Dependency	版本	描述
spring-security-core		
spring-security-web		
cas-client-core	3.1.12	JA-SIG CAS客户端. 这是Spring

Dependency	版本	描述
		Security集成的基础.
ehcache	1.6.2	如果需要你基于缓存Ehcache标签 (可选).

spring-security-openid

OpenID模块.

Table 11. OpenID Dependencies

Dependency	版本	描述
spring-security-core		
spring-security-web		
openid4java-nodeps	0.9.6	Spring Security的OpenID集成使用OpenID4Java.
httpclient	4.1.1	openid4java-nodeps 取决于 HttpClient 4.
guice	2.0	openid4java-nodeps 取决于 Guice 2.

spring-security-taglibs

提供Spring Security的JSP标记的实现.

Table 12. Taglib Dependencies

Dependency	版本	描述
spring-security-core		
spring-security-web		
spring-security-acl		如果需要你使用 `accesscontrollist` 标记或 `hasPermission()` ACLs表达式 (可选).
spring-expression		如果需要你在标签访问限制中使用SPEL表达式

包括::_includes/faq.adoc[]

包括::_includes/migrating.adoc[]

---

1. 一旦响应已提交，不可能创建一个会话
2. 请注意，您需要在您的应用程序上下文中包括安全命名空间的XML文件，才能使用此语法。旧的语法仍然支持使用 `filter-chain-map`，但弃用构造函数参数的注入。
3. 而不是一个路径模式的要求， `request-matcher-ref` 属性可以用来指定一个更强大的匹配 `requestmatcher` 的实例
4. 你可能已经看过这浏览器不支持Cookie和` JSESSIONID 参数附加到URL分号后。然而，RFC允许这些参数的URL中的任何路径段的存在
5. 一旦请求离开 `FilterChainProxy` 的原始值将被退回，仍然会提供给应用程序。
6. 因此，例如，原来的请求路径 `/secure;hack=1/somefile.html;hack=2` 将返回为 `/secure/somefile.html`.
7. 我们使用了进行，这样SecurityContextHolder仍然包含主体，其可以是用于显示对用户有用的信息。在Spring Security的老版本中，我们让servlet容器处理403错误信息，缺乏这种有用的上下文信息。
8. 在Spring Security2.0和更早的版本，此过滤器被称为 `HttpSessionContextIntegrationFilter` 和存储方面的全部工作由过滤器本身进行执行。如果你熟悉这个类，那么其中大部分可用的配置选项现在可以在 `HttpSessionSecurityContextRepository` 找到。
9. 由于历史原因,Spring Security 3.0之前,这个过滤器被称为 `AuthenticationProcessingFilter` 和入口点被称为 `AuthenticationProcessingFilterEntryPoint`。由于框架现在支持多种不同形式的身份验证，他们在3.0都被给出更具体的名称。
10. 在版本3.0之前,在这一点上，应用程序流程已经演变到一个阶段由这个类和策略插件属性混合来控制，3.0重构代码的决定使这两个策略完全负责。
11. 有可能以十六进制编码的密码格式(MD5(username:realm:password))提供 `DigestAuthenticationFilter.passwordAlreadyEncoded` 设置为 `true`。然而,其他密码编码不会使用摘要式身份验证。
12. 本质上，用户名不包括在该cookie，以防止不必要的暴露有效登录名。有一个讨论这篇文章的评论部分。
13. The modules `apacheds-core`, `apacheds-core-entry`, `apacheds-protocol-shared`, `apacheds-protocol-ldap` and `apacheds-server-jndi` are required.