

目 录

致谢

介绍

Spring Security 概述

Hello World

单元测试

方法级别的安全

基于角色的登录

基于 JWT 的认证

JDBC 认证

使用 JPA 及 UserDetailsService

基本认证

摘要认证

摘要认证的密码加密

通用密码加密

Remember-Me（记住我）认证：基于散列的令牌方法

Remember-Me（记住我）认证：基于持久化的令牌方法

OAuth 2.0 认证的原理与实践

参考资料

致谢

当前文档《Spring Security 教程(Spring Security Tutorial)》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建, 生成于 2018-05-03。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能, 以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理, 书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候, 发现文档内容有不恰当的地方, 请向我们反馈, 让我们共同携手, 将知识准确、高效且有效地传递给每一个人。

同时, 如果您在日常生活、工作和学习中遇到有价值有营养的知识文档, 欢迎分享到 书栈(BookStack.CN) , 为知识的传承献上您的一份力量!

如果当前文档生成时间太久, 请到 书栈(BookStack.CN) 获取最新的文档, 以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/spring-security-tutorial>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享, 让知识传承更久远! 感谢知识的创造者, 感谢知识的分享者, 也感谢每一位阅读到此处的读者, 因为我们都将成为知识的传承者。

介绍

- [Spring Security Tutorial](#) 《Spring Security 教程》
 - [Get Started](#) 如何开始阅读
 - [Code](#) 源码
 - [Issue](#) 意见、建议
 - [Contact](#) 联系作者：

Spring Security Tutorial 《Spring Security 教程》



Spring Security Tutorial takes you to learn Spring Security step by step with a large number of samples. There is also a GitBook version of the book: <http://www.gitbook.com/book/waylau/spring-security-tutorial>.

Let's [READ!](#)

Spring Security Tutorial 是一本关于 Spring Security 学习的开源书。利用业余时间写了本书，图文并茂，用大量实例带你一步步走进 Spring Security 的世界。如有疏漏欢迎指正，欢迎提问。感谢您的参与！

注:在 Java 界,另外一个值得推荐的安全框架是 *Apache Shiro*。有关 *Apache Shiro* 的资料,可以看笔者的另外一本开源书《[Apache Shiro 1.2.x 参考手册](#)》。

为了避免繁琐的配置,本书所有实例基于 Spring Boot 之上进行构建。有关Spring Boot 相关的内容,可以参阅笔者的另外一本开源书《[Spring Boot 教程](#)》。

Get Started 如何开始阅读

选择下面入口之一:

- <https://github.com/waylau/spring-security-tutorial/> 的 SUMMARY.md (源码)
- <http://waylau.gitbooks.io/spring-security-tutorial/> 点击 Read 按钮 (同步更新,国内访问速度一般)

Code 源码

书中所有示例源码,移步至

<https://github.com/waylau/spring-security-tutorial>

[samples](#) 目录下

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖

<https://github.com/waylau/spring-security-tutorial/issues>

Contact 联系作者:

- Blog: waylau.com
- Gmail: [waylau521\(at\)gmail.com](mailto:waylau521(at)gmail.com)
- Weibo: waylau521

- Twitter: [waylau521](#)
- Github : [waylau](#)

Spring Security 概述

- Spring Security 概述
 - Spring Security 简介
 - Spring Security 的安装
 - 使用 Maven
 - 使用 Gradle
 - 模块
 - Core - spring-security-core.jar
 - Remoting - spring-security-remoting.jar
 - Web - spring-security-web.jar
 - Config - spring-security-config.jar
 - LDAP - spring-security-ldap.jar
 - ACL - spring-security-acl.jar
 - CAS - spring-security-cas.jar
 - OpenID - spring-security-openid.jar
 - Test - spring-security-test.jar
 - 源码

Spring Security 概述

Spring Security 简介

Spring Security为基于 Java EE 的企业软件应用程序提供全面的安全服务。特别是使用 Spring Framework 构建的项目，可以更好的使用 Spring Security 来加快构建的速度。

Spring Security 的出现有很多原因，但主要是基于 Java EE 的 Servlet 规范或 EJB 规范的缺乏对企业应用的安全性方面的支

持。而使用 Spring Security 克服了这些问题，并带来了数十个其他有用的可自定义的安全功能。

注：在 Java 界，另外一个值得推荐的安全框架是 *Apache Shiro*。有关 *Apache Shiro* 的资料，可以看笔者的另外一本开源书《[Apache Shiro 1.2.x 参考手册](#)》。

应用程序安全性的两个主要领域是

- 身份认证 (authentication)：“认证”是建立主体 (principal) 的过程。主体 通常是指可以在您的应用程序中执行操作的用户、设备或其他系统；
- 授权 (authorization)：或称为“访问控制 (access-control)”，“授权”是指决定是否允许主体在应用程序中执行操作。为了到达需要授权决定的点，认证过程已经建立了主体的身份。这些概念是常见的，并不是特定于 Spring Security。

在认证级别，Spring Security 支持各种各样的认证模型。这些认证模型中的大多数由第三方提供，或者由诸如因特网工程任务组的相关标准机构开发。此外，Spring Security 提供了自己的一组认证功能。具体来说，Spring Security 目前支持所有这些技术的身份验证集成：

- HTTP BASIC 认证头 (基于IETF RFC的标准)
- HTTP Digest 认证头 (基于IETF RFC的标准)
- HTTP X.509 客户端证书交换 (基于IETF RFC的标准)
- LDAP (一种非常常见的跨平台身份验证需求，特别是在大型环境中)
- 基于表单的身份验证 (用于简单的用户界面需求)
- OpenID 身份验证
- 基于预先建立的请求头的验证 (例如Computer Associates Siteminder)

- Jasig Central Authentication Service, 也称为CAS, 这是一个流行的开源单点登录系统
- 远程方法调用 (RMI) 和HttpInvoker (Spring远程协议) 的透明认证上下文传播
- 自动“remember-me”身份验证 (所以您可以勾选一个框, 以避免在预定时间段内重新验证)
- 匿名身份验证 (允许每个未经身份验证的调用, 来自动承担特定的安全身份)
- Run-as 身份验证 (如果一个调用应使用不同的安全身份继续运行, 这是有用的)
- Java认证和授权服务 (Java Authentication and Authorization Service, JAAS)
- Java EE 容器认证 (因此, 如果需要, 仍然可以使用容器管理身份验证)
- 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人群*

- 自己创建的认证系统

(其中加*是指由第三方提供, Spring Security 来集成)

许多独立软件供应商 (ISV) 采用 Spring Security, 是出于这种灵活的认证模型。这样, 他们可以快速地将他们的解决方案与他们的最终客户需要进行组合, 从而避免了进行大量的工作或者要求变更。如果上述认证机制都不符合您的需求, Spring Security 作为一个开放平台, 可以基于它很容易就实现自己的认证机制。

不考虑认证机制, Spring Security 提供了一组深入的授权功能。有三个主要领域:

- 对 Web 请求进行授权
- 授权某个方法是否可以被调用
- 授权访问单个领域对象实例

Spring Security 的安装

最小化依赖如下:

使用 Maven

使用 Maven 的最少依赖如下所示:

```
1. <dependencies>
2.     .....
3.     <dependency>
4.         <groupId>org.springframework.security</groupId>
5.         <artifactId>spring-security-web</artifactId>
6.         <version>4.2.2.RELEASE</version>
7.     </dependency>
8.     <dependency>
9.         <groupId>org.springframework.security</groupId>
```

```

10.         <artifactId>spring-security-config</artifactId>
11.         <version>4.2.2.RELEASE</version>
12.     </dependency>
13.     .....
14. </dependencies>

```

使用 Gradle

使用 Gradle 的最少依赖如下所示：

```

1. dependencies {
2.     .....
3.     compile 'org.springframework.security:spring-security-
      web:4.2.2.RELEASE'
4.     compile 'org.springframework.security:spring-security-
      config:4.2.2.RELEASE'
5.     .....
6. }

```

本书所有例子，将统一使用 Gradle 编写。各位读者如果需要了解 Gradle 方面的知识，可以参阅 [《Gradle 3 用户指南》](#)。

模块

自 Spring 3 开始，Spring Security 将代码划分到不同的 jar 中，这使得不同的功能模块和第三方依赖显得更加清晰。

Core - spring-security-core.jar

包含核心的 authentication 和 authorization 的类和接口、远程支持和基础配置API。任何使用 Spring Security 的应用都需要引入这个 jar。支持本地应用、远程客户端、方法级别的安全和 JDBC 用户配置。主要包含的顶级包为为：

- `org.springframework.security.core` : 核心
- `org.springframework.security.access` : 访问, 即 authorization 的作用
- `org.springframework.security.authentication` : 认证
- `org.springframework.security.provisioning` : 配置

Remoting - spring-security-remoting.jar

提供与 Spring Remoting 整合的支持, 你并不需要这个除非你需要使用 Spring Remoting 写一个远程客户端。主包为:

`org.springframework.security.remoting`

Web - spring-security-web.jar

包含 filter 和相关 Web安全的基础代码。如果我们需要使用 Spring Security 进行 Web 安全认证和基于URL的访问控制。主包为:

`org.springframework.security.web`

Config - spring-security-config.jar

包含安全命名空间解析代码和 Java 配置代码。如果您使用 Spring Security XML 命名空间进行配置或 Spring Security 的 Java 配置支持, 则需要它。主包为:

`org.springframework.security.config`。我们不应该在代码中直接使用这个jar中的类。

LDAP - spring-security-ldap.jar

LDAP 认证和配置代码。如果你需要进行 LDAP 认证或者管理 LDAP 用户实体。顶级包为:

`org.springframework.security.ldap`

ACL - spring-security-acl.jar

特定领域对象的ACL(访问控制列表)实现。使用其可以对特定对象的实例进行一些安全配置。顶级包为：`org.springframework.security.acls`

CAS - spring-security-cas.jar

Spring Security CAS 客户端集成。如果你需要使用一个单点登录服务器进行 Spring Security Web 安全认证，需要引入。顶级包为：`org.springframework.security.cas`

OpenID - spring-security-openid.jar

OpenId Web 认证支持。基于一个外部 OpenId 服务器对用户进行验证。顶级包为：`org.springframework.security.openid`，需要使用 OpenID4Java。

一般情况下，`spring-security-core` 和 `spring-security-config` 都会引入，在 Web 开发中，我们通常还会引入 `spring-security-web`。

Test - spring-security-test.jar

用于测试 Spring Security。在开发环境中，我们通常需要添加该包。

源码

如果，你对 Spring Security 的源码感兴趣，你代码托管于 <https://github.com/spring-projects/spring-security>。

本教程所使用的代码，在项目的根目录 `samples` 下。

Hello World

- [Hello World](#)
 - [环境](#)
 - [Gradle Wrapper](#)
 - [build.gradle](#)
 - [1. 自定义依赖的版本](#)
 - [2. 修改项目的名称](#)
 - [3. 修改项目的仓库地址](#)
 - [4. 指定依赖](#)
 - [配置类](#)
 - [运行](#)
 - [自定义 403 页面](#)
 - [处理登出](#)

Hello World

依照惯例，我们从一个 Hello World 项目入手。

我们新建了一个名为 `hello-world` 的 Gradle 项目。

基于Form 表单的登录认证。

环境

- Gradle 3.4.1
- Spring Boot 1.5.2.RELEASE
- Thymeleaf 3.0.3.RELEASE
- Thymeleaf Layout Dialect 2.2.0

Gradle Wrapper

修改 Gradle Wrapper 的配置 `gradle-wrapper.properties`，使用最新的Gradle：

```
1. distributionBase=GRADLE_USER_HOME
2. distributionPath=wrapper/dists
3. zipStoreBase=GRADLE_USER_HOME
4. zipStorePath=wrapper/dists
5. distributionUrl=https\://services.gradle.org/distributions/gradle-3.4.1-bin.zip
```

build.gradle

1. 自定义依赖的版本

我们可以自定义 Spring Boot 的版本，比如，我们使用了目前最新的 1.5.2.RELEASE 版本。

```
1. // buildscript 代码块中脚本优先执行
2. buildscript {
3.     .....
4.     ext {
5.         springBootVersion = '1.5.2.RELEASE'
6.     }
7.
8.     // 自定义 Thymeleaf 和 Thymeleaf Layout Dialect 的版本
9.     ext['thymeleaf.version'] = '3.0.3.RELEASE'
10.    ext['thymeleaf-layout-dialect.version'] = '2.2.0'
11.    .....
12. }
```

2. 修改项目的名称

修改 `build.gradle` 文件，让我们的 `hello-world` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'hello-world'  
3.     version = '1.0.0'  
4. }
```

3. 修改项目的仓库地址

为了加快构建速度，我们自定义了一个国内的仓库镜像地址：

```
1. repositories {  
2.     maven {  
3.         url 'http://maven.aliyun.com/nexus/content/groups/public/'  
4.     }  
5. }
```

4. 指定依赖

```
1. // 依赖关系  
2. dependencies {  
3.  
4.     // 该依赖对于编译发行是必须的  
5.     compile('org.springframework.boot:spring-boot-starter-web')  
6.  
7.     // 添加 Thymeleaf 的依赖  
8.     compile('org.springframework.boot:spring-boot-starter-  
9.         thymeleaf')  
10.  
11.    // 添加 Spring Security 依赖  
11.    compile('org.springframework.boot:spring-boot-starter-  
12.        security')
```

```

12.
13.     // 添加   Thymeleaf Spring Security 依赖, 与 Thymeleaf 版本一致都是
        3.x
14.     compile('org.thymeleaf.extras:thymeleaf-extras-
        springsecurity4:3.0.2.RELEASE')
15.
16.     // 该依赖对于编译测试是必须的, 默认包含编译产品依赖和编译时依
17.     testCompile('org.springframework.boot:spring-boot-starter-
        test')
18. }

```

配置类

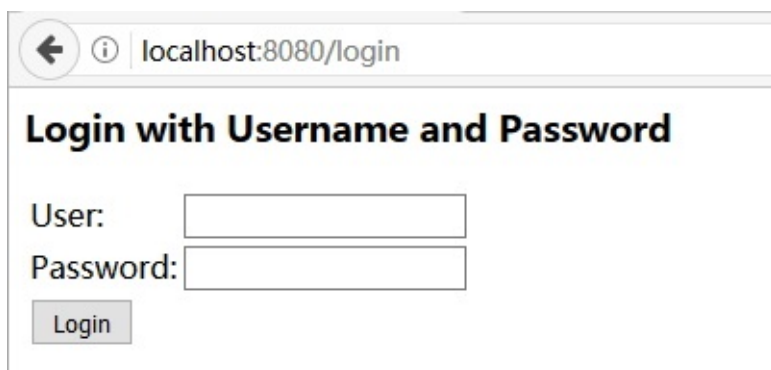
```

1. @EnableWebSecurity
2. public class SecurityConfig extends WebSecurityConfigurerAdapter {
3.     .....
4.     /**
5.      * 自定义配置
6.      */
7.     @Override
8.     protected void configure(HttpSecurity http) throws Exception {
9.         http
10.            .authorizeRequests()
11.                .antMatchers("/css/**", "/js/**", "/fonts/**",
"/index").permitAll() // 虽都可以访问
12.                .antMatchers("/users/**").hasRole("USER") // 需要
相应的角色才能访问
13.                .antMatchers("/admins/**").hasRole("ADMIN") // 需
要相应的角色才能访问
14.                .and()
15.                .formLogin() //基于 Form 表单登录验证
16.                .loginPage("/login").failureUrl("/login-error");
// 自定义登录界面
17.     }
18.     .....
19. }

```

这段代码内容很少，但事实上已经做了很多的默认安全验证，包括：

- 访问应用中的每个URL都需要进行验证
- 生成一个登陆表单
- 允许用户使用用户名和密码来登陆
- 允许用户注销
- [CSRF](#)攻击拦截
- [Session Fixation](#) 攻击
- 安全 Header 集成
 - 启用 [HTTP Strict Transport Security](#)
 - [X-Content-Type-Options.aspx](#)) 集成
 - Cache Control (缓存控制)
 - [X-XSS-Protection.aspx](#)) 集成
 - 集成 X-Frame-Options 来防止 [Clickjacking](#)
- 集成了以下 Servlet API :
 - [HttpServletRequest#getRemoteUser\(\)](#)
 - [HttpServletRequest.html#getUserPrincipal\(\)](#)
 - [HttpServletRequest.html#isUserInRole\(java.lang.String\)](#)
 - [HttpServletRequest.html#login\(java.lang.String, java.lang.String\)](#)
 - [HttpServletRequest.html#logout\(\)](#)
- 所有匹配 `/users/**` 的需要“USER”角色授权
- 所有匹配 `/admins/**` 的需要“ADMIN”角色授权
- 基于 Form 表单登录验证的方式。登录界面指定为 `/login`，登录失败等会重定向到 `/login-error` 页面。如果不指定登录界面，则会 Spring Security 会提供一个默认的登录页面：



← localhost:8080/login

Login with Username and Password

User:

Password:

Login

那么，上述安全设置是如何默认启用的呢？我们观察下 `SecurityConfig` 所继承的

`org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter` 类，其初始化的时候是这样的

```

1. http
2.     .csrf().and()
3.     .addFilter(new WebAsyncManagerIntegrationFilter())
4.     .exceptionHandling().and()
5.     .headers().and()
6.     .sessionManagement().and()
7.     .securityContext().and()
8.     .requestCache().and()
9.     .anonymous().and()
10.    .servletApi().and()
11.    .apply(new DefaultLoginPageConfigurer<HttpSecurity>()).and()
12.    .logout();

```

也就是说，它默认的时候就隐式的启用了多安全设置。

再看下其他几个方法：

```

1. /**
2.  * 用户信息服务
3.  */
4. @Bean
5. public UserDetailsService userDetailsService() {
6.     InMemoryUserDetailsManager manager = new

```

```

        InMemoryUserDetailsManager()); // 在内存中存放用户信息
7.
        manager.createUser(User.withUsername("waylau").password("123456").role("USER"));
8.
        manager.createUser(User.withUsername("admin").password("123456").role("ADMIN"));
9.        return manager;
10.    }
11.
12.    /**
13.     * 认证信息管理
14.     * @param auth
15.     * @throws Exception
16.     */
17.    @Autowired
18.    public void configureGlobal(AuthenticationManagerBuilder auth)
        throws Exception {
19.        auth.userDetailsService(userDetailsService());
20.    }

```

其含义为：

- 用户的认证信息是使用 InMemoryUserDetailsManager 来存储在内存中；
- 我们默认生成了两个用户，一个是“waylau” 是拥有“USER”角色权限；另一个是“admin” 是拥有“USER”以及“ADMIN”角色权限

运行

运行程序，访问<http://localhost:8080> 页面。

当我们试图访问以下页面时，提示需要用户登录：

- “用户管理”：对应 “/users” URL；
- “管理员管理”：对应 “/admins” URL

登录页面：



Spring Security Tutorial 首页 用户管理 管理员管理 登录 退出

登录页面

普通用户账户密码: waylau/123456

管理员账户密码: admin/123456

用户名

waylau

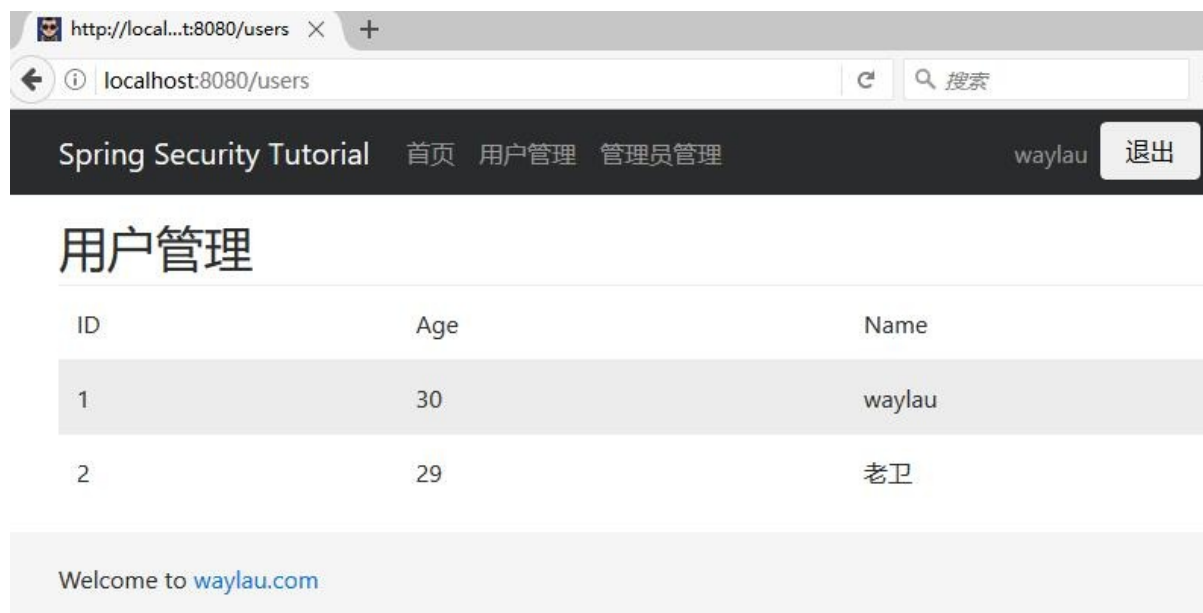
密码

.....

登录

Welcome to [waylau.com](#)

使用具有“USER”角色授权的“waylau”用户登录，可以访问“用户管理”页面：



Spring Security Tutorial 首页 用户管理 管理员管理 waylau 退出

用户管理

ID	Age	Name
1	30	waylau
2	29	老卫

Welcome to [waylau.com](#)

“waylau”用户登录可以访问“管理员管理”页面时，提示拒绝访问：



Whitelabel Error Page

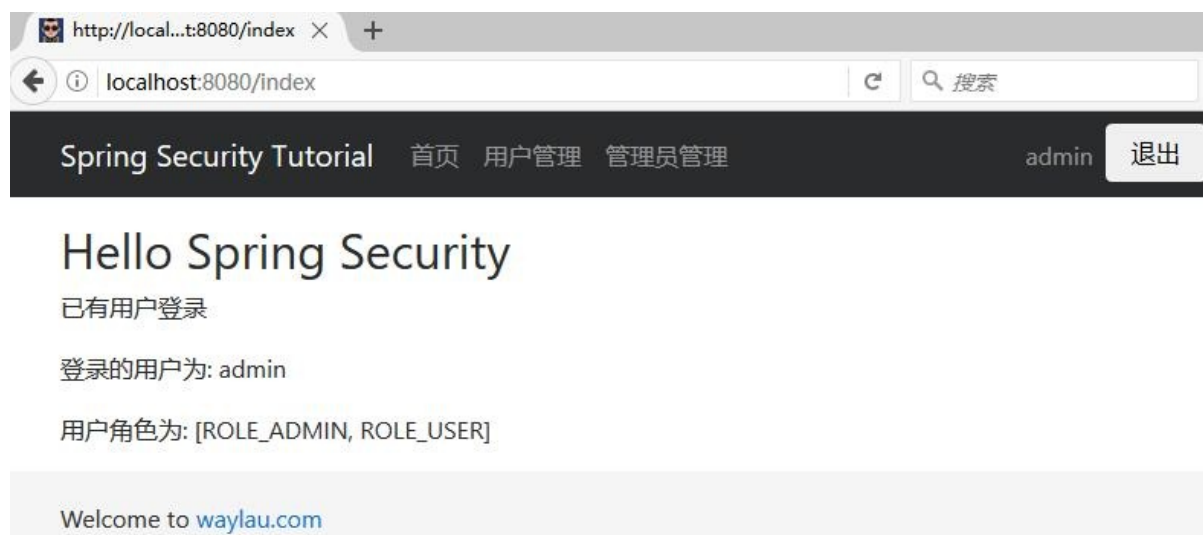
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Mar 17 23:25:07 CST 2017

There was an unexpected error (type=Forbidden, status=403).

Access is denied

此时，我们换用 “admin” 账号登录即可访问该页面。返回首页，我们能看到该用户的权限信息：



自定义 403 页面

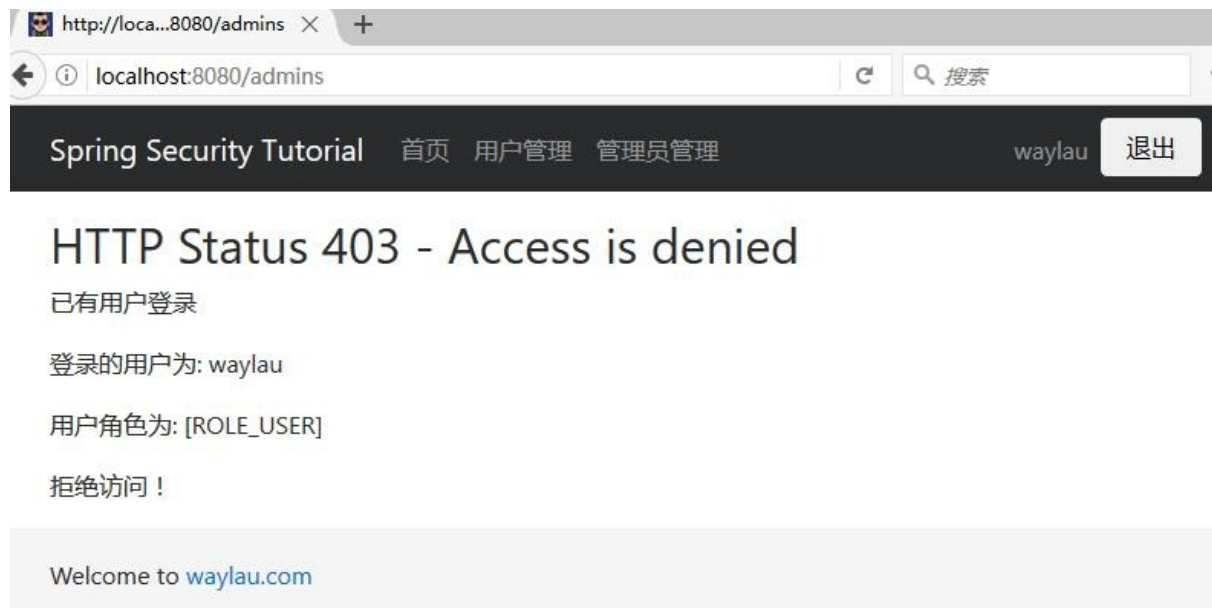
默认的提示拒绝访问页面太丑，我需要自定义一个页面。我们在返回的页面里面提示“拒绝访问！”。

同时，我要在配置类里面，配置重定向的内容：

```
1. ....
2. .formLogin()    //基于 Form 表单登录验证
3.     .loginPage("/login").failureUrl("/login-error") // 自定义登录界面
```

```
4.         .and()  
5.         .exceptionHandling().accessDeniedPage("/403"); // 处理异常，拒绝访问就  
           重定向到 403 页面  
6.         .....
```

最终效果：



处理登出

使用`WebSecurityConfigurerAdapter`时，应用会自动提供注销功能。默认情况下，访问 URL `/logout` 来注销用户。注销动作，做了以下几个事情：

- 使 HTTP 会话无效
- 清除配置了 RememberMe 身份认证的信息
- 清除 SecurityContextHolder
- 重定向到 `/login?logout`

与配置登录功能类似，我们也可以使用各种选项进一步自定义登出要求，比如：


```
1. http.logout().logoutSuccessUrl("/"); // 成功登出后, 重定向到 首页
```

这样, 成功登出后, 重定向到 首页。

其他的还可以选项还有很多:

```
1. protected void configure(HttpSecurity http) throws Exception {
2.     http
3.         .logout()
4.             .logoutUrl("/my/logout") // 1
5.             .logoutSuccessUrl("/my/index")
6.             .logoutSuccessHandler(logoutSuccessHandler) // 2
7.             .invalidateHttpSession(true) // 3
8.             .addLogoutHandler(logoutHandler) // 4
9.             .deleteCookies(cookieNamesToClear) // 5
10.         .and()
11.         ...
12. }
```

- (1) 自定义触发登出的 URL
- (2) 指定一个自定义LogoutSuccessHandler
- (3) 指定在注销时是否使 HttpSession 无效。默认情况下是 true
- (4) 添加 LogoutHandler。默认情况下, SecurityContextLogoutHandler 作为最后一个 LogoutHandler 被添加
- (5) 允许指定在注销成功时要删除的 Cookie 的名称。这是一个显式添加 CookieClearingLogoutHandler 的快捷方式

单元测试

- 集成测试
 - 环境
 - build.gradle
 - 1. 修改项目的名称
 - 2. 指定依赖
 - 编写测试类
 - 测试 `/users`

集成测试

我们从 Hello World 项目入手，增加单元测试功能。

我们新家了一个名为 `hello-world-test` 的 Gradle 项目。

环境

- Gradle 3.4.1
- Spring Boot 1.5.2.RELEASE
- Thymeleaf 3.0.3.RELEASE
- Thymeleaf Layout Dialect 2.2.0
- Spring Security Test 4.2.2.RELEASE

build.gradle

1. 修改项目的名称

修改 build.gradle 文件，让我们的 `hello-world` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {
2.     baseName = 'hello-world-test'
3.     version = '1.0.0'
4. }
```

2. 指定依赖

```
1. // 依赖关系
2. dependencies {
3.     .....
4.     // 该依赖对于编译测试是必须的，默认包含编译产品依赖和编译时依
5.     testCompile('org.springframework.security:spring-security-
6.         test:4.2.2.RELEASE')
7.     .....
7. }
```

编写测试类

测试

/users

首先我们测试 `/users` 接口，编写UserControllerTest.java：

```
1. @RunWith(SpringRunner.class)
2. @SpringBootTest
3. @AutoConfigureMockMvc
4. public class UserControllerTest {
5.
6.     @Autowired
7.     private MockMvc mockMvc;
8.
9.     @Test
10.    public void testList() throws Exception {
11.        mockMvc.perform(MockMvcRequestBuilders.get("/users"))
```

```

12.         .andExpect(status().isOk());
13.     }
14. }

```

运行 JUnit，结果是测试没有通过。在控制台，我们能看到如下请求和相应的结果：

```

1.
2. MockHttpServletRequest:
3.     HTTP Method = GET
4.     Request URI = /users
5.     Parameters = {}
6.     Headers = {}
7.
8. Handler:
9.     Type = null
10.
11. Async:
12.     Async started = false
13.     Async result = null
14.
15. Resolved Exception:
16.     Type = null
17.
18. ModelAndView:
19.     View name = null
20.     View = null
21.     Model = null
22.
23. FlashMap:
24.     Attributes = null
25.
26. MockHttpServletResponse:
27.     Status = 302
28.     Error message = null
29.     Headers = {X-Content-Type-Options=[nosniff], X-XSS-
        Protection=[1; mode=block], Cache-Control=[no-cache, no-store, max-

```

```

    age=0, must-revalidate], Pragma=[no-cache], Expires=[0], X-Frame-Options=[DENY], Location=[http://localhost/login]}
30.         Content type = null
31.             Body =
32.         Forwarded URL = null
33.         Redirected URL = http://localhost/login
34.         Cookies = []

```

可以看到响应的状态码为 302，重定向到了 `http://localhost/login` URL。

我们再次修改测试接口，这一次我们使用 mock 用户名为“waylau”角色权限为“USER”的用户：

```

1. @Test
2. @WithMockUser(username="waylau", password="123456", roles={"USER"})
   // mock 了一个用户
3. public void testListWithUser() throws Exception {
4.     mockMvc.perform(MockMvcRequestBuilders.get("/users"))
5.         .andExpect(status().isOk());
6. }

```

这次，我们的测试用例通过。如果我们把用的角色权限改为“ADMIN”，会怎么样呢？

```

1. MockHttpServletRequest:
2.     HTTP Method = GET
3.     Request URI = /users
4.     Parameters = {}
5.     Headers = {}
6.
7. Handler:
8.     Type = null
9.
10. Async:
11.     Async started = false

```

```
12.      Async result = null
13.
14.  Resolved Exception:
15.      Type = null
16.
17.  ModelAndView:
18.      View name = null
19.      View = null
20.      Model = null
21.
22.  FlashMap:
23.      Attributes = null
24.
25.  MockHttpServletResponse:
26.      Status = 403
27.      Error message = Access is denied
28.      Headers = {X-Content-Type-Options=[nosniff], X-XSS-
Protection=[1; mode=block], Cache-Control=[no-cache, no-store, max-
age=0, must-revalidate], Pragma=[no-cache], Expires=[0], X-Frame-
Options=[DENY]}
29.      Content type = null
30.      Body =
31.      Forwarded URL = null
32.      Redirected URL = null
33.      Cookies = []
```

看到测试失败后，控制台打印的信息，状态码为 403，错误信息为“Access is denied”。

方法级别的安全

- 方法级别的安全
 - `build.gradle`
 - 编写服务类
 - 控制器
 - 配置类
 - `@Secured`
 - `@PreAuthorize` / `@PostAuthorize`
 - 运行
 - 根据权限来显示或者隐藏操作

方法级别的安全

我们在 `hello-world-test` 的基础上，我们新建了一个名为 `method-security` 的 Gradle 项目。

本项目用于演示方法级别的安全设置。

build.gradle

修改 `build.gradle` 文件，让我们的 `method-security` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {
2.     baseName = 'method-security'
3.     version = '1.0.0'
4. }
```

编写服务类

创建 `com.waylau.spring.boot.security.service` 包，用于放置服务类。创建了 `UserService` 接口。接口比较简单，主要是用于查询和删除：

```
1. public interface UserService {  
2.  
3.     void removeUser(Long id);  
4.  
5.     List<User> listUsers();  
6. }
```

`UserServiceImpl` 是 `UserService` 接口的实现，在内存里面模拟了 `User` 的存储库。

```
1. @Service  
2. public class UserServiceImpl implements UserService {  
3.  
4.     private static final Map<Long, User> userRepository = new  
5.         ConcurrentHashMap<>();  
6.  
7.     public UserServiceImpl(){  
8.         userRepository.put(1L, new User(1L, "waylau", 30));  
9.         userRepository.put(2L, new User(2L, "老卫", 29));  
10.        userRepository.put(3L, new User(3L, "doufe", 109));  
11.    }  
12.  
13.    @Override  
14.    public void removeUser(Long id) {  
15.        userRepository.remove(id);  
16.    }  
17.  
18.    @Override  
19.    public List<User> listUsers() {  
20.        List<User> users = null;  
21.        users = new ArrayList<User>(userRepository.values());  
22.    }  
23. }
```



```

21.         return users;
22.     }
23.
24. }

```

控制器

在 `UserController` 中，增加了删除用户的方法：

```

1. @PreAuthorize("hasAuthority('ROLE_ADMIN')") // 指定角色权限才能操作方
   法
2. @GetMapping(value = "delete/{id}")
3. public ModelAndView delete(@PathVariable("id") Long id, Model
   model) {
4.     userService.removeUser(id);
5.     model.addAttribute("userList", userService.listUsers());
6.     model.addAttribute("title", "删除用户");
7.     return new ModelAndView("users/list", "userModel", model);
8. }

```

其中，我们使用了 `@PreAuthorize` 注解。

配置类

在配置类上，我们加上一个注解 `@EnableGlobalMethodSecurity`：

```

1. @EnableWebSecurity
2. @EnableGlobalMethodSecurity(prePostEnabled = true) // 启用方法安全设
   置
3. public class SecurityConfig extends WebSecurityConfigurerAdapter {
4.     .....
5. }

```

注意：`@EnableGlobalMethodSecurity` 可以配置多个参数：

- `prePostEnabled` : 决定 Spring Security 的前注解是否可用 `@PreAuthorize`、`@PostAuthorize` 等
- `securedEnabled` : 决定是否Spring Security的保障注解 `@Secured` 是否可用
- `jsr250Enabled` : 决定 JSR-250 注解 `@RolesAllowed` 等是否可用。

配置方式分别如下：

```

1. @EnableGlobalMethodSecurity(securedEnabled = true)
2. public class MethodSecurityConfig {
3.     // ...
4. }
5.
6. @EnableGlobalMethodSecurity(jsr250Enabled = true)
7. public class MethodSecurityConfig {
8.     // ...
9. }
10.
11.
12. @EnableGlobalMethodSecurity(prePostEnabled = true)
13. public class MethodSecurityConfig {
14.     // ...
15. }
```

在同一个应用程序中，可以启用多个类型的注解，但是只应该设置一个注解对于行为类的接口或者类。如果将2个注解同事应用于某一特定方法，则只有其中一个将被应用。

`@Secured`

此注释是用来定义业务方法的安全配置属性的列表。您可以在需要安全角色/权限等的方法上指定 `@Secured`，并且只有那些角色/权限的用户才可以调用该方法。如果有人不具备要求的角色/权限但试图调用此

方法，将会抛出 `AccessDenied` 异常。

`@Secured` 源于 Spring 之前版本。它有一个局限就是不支持 Spring EL 表达式。可以看看下面的例子：

如果你想指定 AND（和）这个条件，我的意思说 `deleteUser` 方法只能被同时拥有 ADMIN & DBA。但是仅仅通过使用 `@Secured` 注解是无法实现的。

但是你可以使用 Spring 的新的注

解 `@PreAuthorize` / `@PostAuthorize`（支持 Spring EL），使得实现上面的功能成为可能，而且无限制。

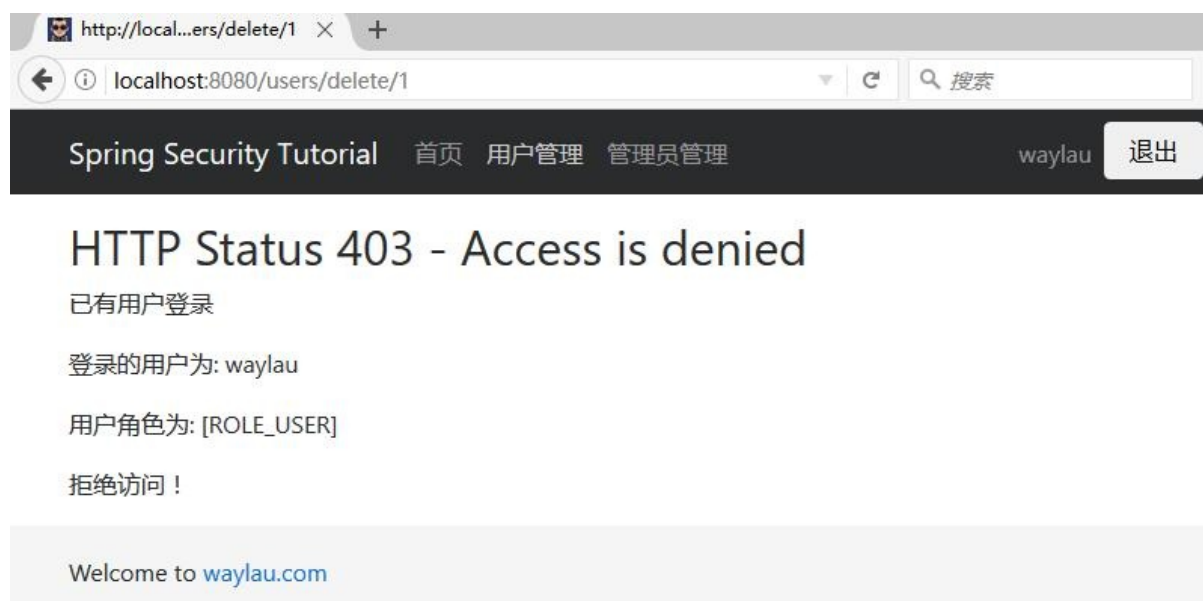
`@PreAuthorize` / `@PostAuthorize`

Spring 的 `@PreAuthorize` / `@PostAuthorize` 注解更适合方法级的安全，也支持 Spring EL 表达式语言，提供了基于表达式的访问控制。

- `@PreAuthorize` 注解：适合进入方法前的权限验证，
`@PreAuthorize` 可以将登录用户的角色/权限参数传到方法中。
- `@PostAuthorize` 注解：使用并不多，在方法执行后再进行权限验证。

运行

运行项目，我们查看下最终的效果：



我们用“USER”角色权限可以查看用户列表，但如果想删除用户，则提示“访问拒绝”。

根据权限来显示或者隐藏操作

实际上，如果用户不具备某个操作的权限，那么那个操作按钮就不应该显示出来。我们可以使用 `sec:authorize` 属性能达到这个目的。

修改 `list.html` 中的表格：

```

1.  .....
2.  <table class="table table-hover">
3.      <thead>
4.      <tr>
5.          <td>ID</td>
6.          <td>Age</td>
7.          <td>Name</td>
8.          <td sec:authorize="hasRole('ADMIN')">Operation</td>
9.      </tr>
10. </thead>
11. <tbody>
12. <tr th:if="${userModel.userList.size()} eq 0">
13.     <td colspan="3">没有用户信息!!</td>

```

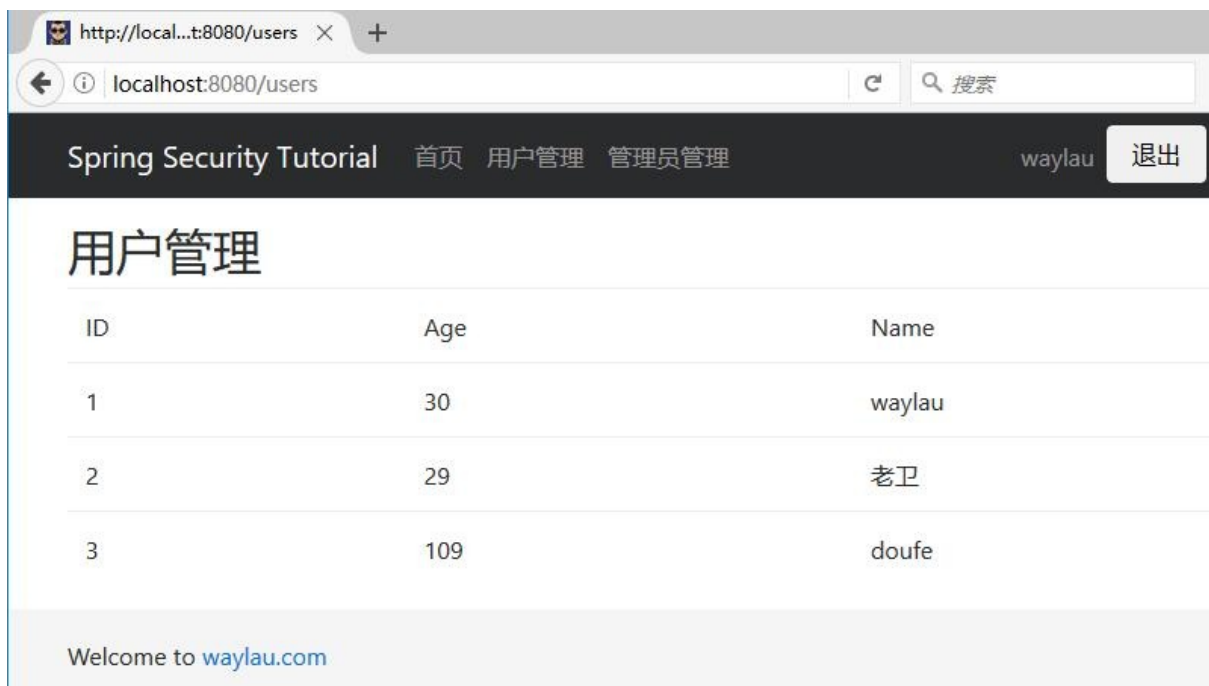
```

14.         </tr>
15.         <tr th:each="user : ${userModel.userList}">
16.             <td th:text="${user.id}">1</td>
17.             <td th:text="${user.age}">11</td>
18.             <td th:text="${user.name}">waylau</td>
19.             <td sec:authorize="hasRole('ADMIN')">
20.                 <div>
21.                     <a th:href="@{'/users/delete/' + ${user.id}}">
22.                         <i class="fa fa-times" aria-hidden="true"></i>
23.                     </a>
24.                 </div>
25.             </td>
26.         </tr>
27.     </tbody>
28. </table>
29. ....

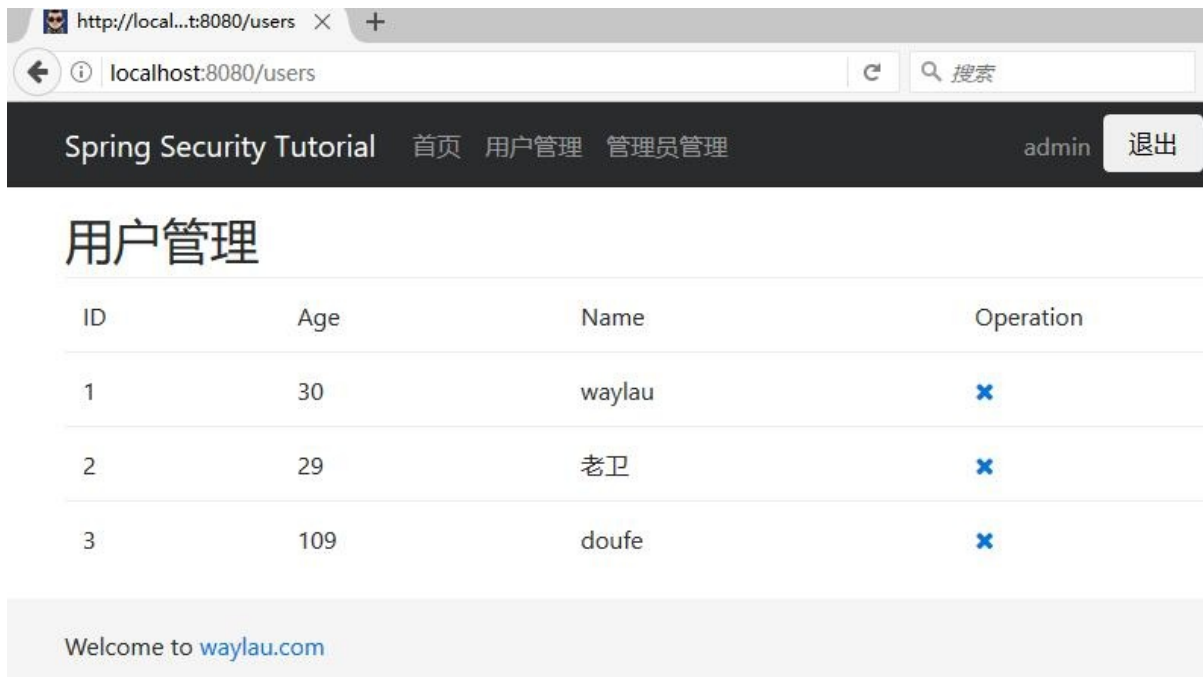
```

我们使用了 `<td sec:authorize="hasRole('ADMIN')">` 这样，只要是具备“ADMIN”权限的用户就能看到删除操作列，否则就看不到那一列的删除操作。效果如下：

不具备“ADMIN”权限的用户：



具备“ADMIN”权限的用户：



The screenshot shows a web browser at `http://localhost:8080/users`. The page has a dark header with the text "Spring Security Tutorial" and navigation links: "首页", "用户管理", and "管理员管理". On the right of the header, it shows the user "admin" and a "退出" (Logout) button. The main content area is titled "用户管理" (User Management) and contains a table with the following data:

ID	Age	Name	Operation
1	30	waylau	✕
2	29	老卫	✕
3	109	doufe	✕

At the bottom of the page, there is a light gray footer with the text "Welcome to waylau.com".

基于角色的登录

- [基于角色的登录](#)
 - [build.gradle](#)
 - [权限处理器](#)
 - [修改配置类](#)
 - [运行](#)

基于角色的登录

本文展示了如何根据不同的用户角色，在登录之后来重定向到不同的页面。

在 `method-security` 项目的基础上，我们构建了一个 `role-base-login` 项目。

build.gradle

修改 `build.gradle` 文件，让我们的 `role-base-login` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'role-base-login'  
3.     version = '1.0.0'  
4. }
```

权限处理器

新增 `com.waylau.spring.boot.security.auth` 包，用于存放权限处理器相

关的类。

新建 `AuthenticationSuccessHandler` ，继承自

`org.springframework.security.web.authentication.SimpleUrlAuthenticationSuccessHandler`

```
1. @Component
2. public class AuthenticationSuccessHandler extends
   SimpleUrlAuthenticationSuccessHandler {
3.     private RedirectStrategy redirectStrategy = new
   DefaultRedirectStrategy();
4.
5.     @Override
6.     public void handle(HttpServletRequest request,
   HttpServletResponse response,
7.         Authentication authentication) throws IOException,
   ServletException {
8.         String targetUrl = determineTargetUrl(authentication);
9.
10.        if (response.isCommitted()) {
11.            logger.debug("Response has already been committed.
   Unable to redirect to "
12.                + targetUrl);
13.            return;
14.        }
15.
16.        redirectStrategy.sendRedirect(request, response,
   targetUrl);
17.    }
18.
19.    private String determineTargetUrl(Authentication
   authentication) {
20.        String targetUrl = null;
21.
22.        Collection<? extends GrantedAuthority> authorities =
   authentication.getAuthorities();
23.
24.        List<String> roles = new ArrayList<String>();
```



```

25.
26.         for (GrantedAuthority a : authorities) {
27.             roles.add(a.getAuthority());
28.         }
29.
30.         // 根据不同的角色，重定向到不同页面
31.         if (isAdmin(roles)) {
32.             targetUrl = "/admins";
33.         } else if (isUser(roles)) {
34.             targetUrl = "/users";
35.         } else {
36.             targetUrl = "/403";
37.         }
38.
39.         return targetUrl;
40.     }
41.
42.     private boolean isUser(List<String> roles) {
43.         if (roles.contains("ROLE_USER")) {
44.             return true;
45.         }
46.         return false;
47.     }
48.
49.     private boolean isAdmin(List<String> roles) {
50.         if (roles.contains("ROLE_ADMIN")) {
51.             return true;
52.         }
53.         return false;
54.     }
55.
56. }

```

重写了 `handle()` 方法，这个方法简单地使用 `RedirectStrategy` 来重定向，由自定义的 `determineTargetUrl` 方法返回URL。此方法提取当前认证对象用户记录的角色，然后构造基于角色有相应的URL。最后由负责 `Spring Security` 框架内的所有重定向的

RedirectStrategy ，将请求重定向到指定的URL。

修改配置类

在配置类中，添加成功认证的处理器

AuthenticationSuccessHandler 的实例：

```
1. .formLogin()    //基于 Form 表单登录验证
2.     .loginPage("/login").failureUrl("/login-error") // 自定义登录界面
3.     .successHandler(authenticationSuccessHandler)
```

运行

当我们使用“ADMIN”角色的用户登录成功后，会被重定向

到 `/admins` 页面。而使用“USER”角色的用户登录成功后，会被重定向

到 `/users` 页面。

基于 JWT 的认证

- 基于 JWT 的认证
 - `build.gradle`
 - JWT 的认证

基于 JWT 的认证

我们在 `remember-me-hash` 上，基于 Form 表单的方式，来实现基于散列的令牌方法的 Remember-Me 认证，我们新建一个 `jwt-authentication` 项目。

build.gradle

修改 `build.gradle` 文件，让我们的 `remember-me-hash` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'jwt-authentication'  
3.     version = '1.0.0'  
4. }
```

添加 JWT 依赖：

```
1. // 添加 JSON Web Token Support For The JVM 依赖  
2. compile('io.jsonwebtoken:jjwt:0.7.0')
```

JWT 的认证

JWT 是 Json Web Token 的缩写。它是基于 [RFC 7519](#) 标准定义

的一种可以安全传输的 小巧 和 自包含 的 JSON 对象。由于数据是使用数字签名的，所以是可信任的和安全的。JWT 可以使用 HMAC 算法对密码进行加密或者使用RSA的公钥私钥对来进行签名。

JDBC 认证

- JDBC 认证
 - build.gradle
 - 处理依赖
 - 修改配置文件
 - 修改配置类 SecurityConfig
 - 使用Spring JDBC初始化数据库
 - 访问 H2 控制台

JDBC 认证

本文展示了如何使用 JDBC 的方式来进行认证。在本例，我们将认证信息存储于 H2 数据库中。

在 `method-security` 项目的基础上，我们构建了一个 `jdbc-authentication` 项目。

注：有关 H2 的更多内容，可以参阅作者的另外一部开源书《[H2 Database 教程](#)》

build.gradle

修改 build.gradle 文件，让我们的 `jdbc-authentication` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'jdbc-authentication'  
3.     version = '1.0.0'  
4. }
```

处理依赖

添加 `spring-boot-starter-jdbc` 和 `h2` 的依赖：

```
1. // 添加 Spring Boot JDBC 的依赖
2. compile('org.springframework.boot:spring-boot-starter-jdbc')
3.
4. // 添加 H2 的依赖
5. runtime('com.h2database:h2:1.4.193')
```

修改配置文件

修改配置文件 `application.properties`：

```
1. # 使用 H2 控制台
2. spring.h2.console.enabled=true
```

修改配置类 SecurityConfig

在配置类中，添加对 H2 的过滤策略：

```
1. http
2.     .authorizeRequests()
3.         .antMatchers("/css/**", "/js/**", "/fonts/**",
4.             "/index").permitAll() // 都可以访问
5.         .antMatchers("/h2-console/**").permitAll() // 都可以访问
6.         .antMatchers("/users/**").hasRole("USER") // 需要相应的角色
           才能访问
7.         .antMatchers("/admins/**").hasRole("ADMIN") // 需要相应的角色
           才能访问
8.         .and()
9.         .formLogin() // 基于 Form 表单登录验证
10.        .loginPage("/login").failureUrl("/login-error") // 自定义登录
           界面
```

```

11.      .exceptionHandling().accessDeniedPage("/403"); // 处理异常，拒绝访问就重定向到 403 页面
12. http.logout().logoutSuccessUrl("/"); // 成功登出后，重定向到 首页
13. http.csrf().ignoringAntMatchers("/h2-console/**"); // 禁用 H2 控制台的 CSRF 防护
14. http.headers().frameOptions().sameOrigin(); // 允许来自同一来源的H2 控制台的请求

```

其中：

- `http.csrf().ignoringAntMatchers` ： 用于配置匹配的 URL，不进行 CSRF 防护。这里，我们不需要对 H2 控制台做 CSRF 防护
- `http.headers().frameOptions().sameOrigin()` ： 用于配置同源策略。允许来自同一来源的H2 控制台的请求

使用Spring JDBC初始化数据库

Spring JDBC 有一个 `DataSource` 初始化功能。Spring Boot 默认启用它，并从标准位置（在类路径的根目录中）的 `schema.sql` 和 `data.sql` 脚本中加载SQL。此外，Spring Boot将加载 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件（如果存在的话），其中 `platform` 是 `spring.datasource.platform` 的值，例如。您可以选择将其设置为数据库的供应商名称（`hsqldb`、`h2`、`oracle`、`mysql`、`postgresql` 等）。Spring Boot 默认启用了 Spring JDBC 初始化程序中的快速失败（`fail-fast`）功能，因此，如果脚本导致异常，应用程序将无法启动。可以通过设置 `spring.datasource.schema` 和 `spring.datasource.data` 来更改脚本位置，如果 `spring.datasource.initialize=false`，则不会处理任何位置的脚本。

要禁用快速失败（`fail-fast`）功能，您可以设置 `spring.datasource.continue-on-error=true`。这在应用程序成熟并部署

了几次后可能很有用，因为插入失败意味着数据已经存在，因此不需要阻止应用程序运行。

如果要在 JPA 应用程序（如使用Hibernate）中使用schema.sql初始化，那么如果 Hibernate 尝试创建相同的表，`ddl-auto=create-drop` 将导致错误。要避免这些错误，请将 `ddl-auto` 显式设置为“”（首选）或“none”。无论是否使用 `ddl-auto=create-drop`，您都可以使用 `data.sql` 初始化新数据。

我们的表结构写在了 `schema.sql` 中：

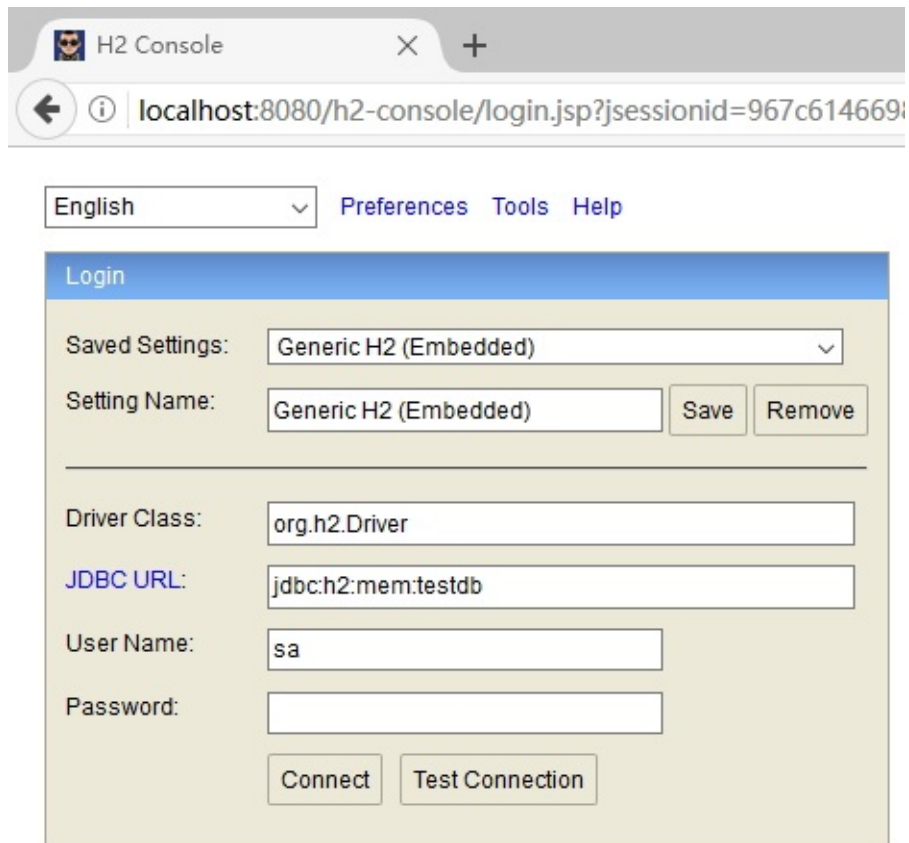
```
1. create table users (  
2.     username varchar(256),  
3.     password varchar(256),  
4.     enabled boolean  
5. );  
6.  
7. create table authorities (  
8.     username varchar(256),  
9.     authority varchar(256)  
10. );
```

我们的数据写在了 `data.sql` 中：

```
1. insert into users (username, password, enabled) values ('waylau',  
    '123456', true);  
2. insert into users (username, password, enabled) values ('admin',  
    '123456', true);  
3.  
4. insert into authorities (username, authority) values ('waylau',  
    'ROLE_USER');  
5. insert into authorities (username, authority) values ('admin',  
    'ROLE_USER');  
6. insert into authorities (username, authority) values ('admin',  
    'ROLE_ADMIN');
```


访问 H2 控制台

设置 JDBC URL 为 `jdbc:mem:testdb` :



可以看到我们新建的数据库表和初始化的数据：

The screenshot shows the H2 Console web interface in a browser. The address bar displays the URL: `localhost:8080/h2-console/login.do?jsessionid=967c61466981ff8a057`. The interface includes a toolbar with icons for database operations and settings like 'Auto commit', 'Max rows: 1000', 'Auto complete', and 'Auto select'. On the left, a tree view shows the database structure: `jdbc:h2:mem:testdb` containing `AUTHORITIES` (with `USERNAME` as `VARCHAR(256)` and `AUTHORITY`), `USERS` (with `USERNAME`, `PASSWORD`, and `ENABLED`), and `INFORMATION_SCHEMA`. The main area shows the SQL statement `SELECT * FROM AUTHORITIES` and its results in a table with 3 rows and 2 columns: `USERNAME` and `AUTHORITY`.

USERNAME	AUTHORITY
waylau	ROLE_USER
admin	ROLE_USER
admin	ROLE_ADMIN

(3 rows, 3 ms)

使用 JPA 及 UserDetailsService

- 使用 JPA 及 UserDetailsService
 - build.gradle
 - 实体
 - 修改 User
 - 新增 Authority
 - 新增存储库
 - 服务类
 - 修改配置类 SecurityConfig
 - 初始化数据库

使用 JPA 及 UserDetailsService

本文展示了如何使用 JPA 自定义 UserDetailsService 及数据库的方式来进行认证。在本例，我们将认证信息存储于 H2 数据库中。

在 `ldap-authentication` 项目的基础上，我们构建了一个 `jpa-userdetailsservice` 项目。

build.gradle

修改 build.gradle 文件，让我们的 `jpa-userdetailsservice` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'jpa-userdetailsservice'  
3.     version = '1.0.0'  
4. }
```

我们将 `spring-boot-starter-jdbc` 依赖改为了 `spring-boot-starter-data-jpa`

实体

修改 User

修改 User，增加了 username、password 两个字段，用于登录时使用。authorities 字段，用于存储该用户的权限信息。同时，User 也实现了

`org.springframework.security.core.userdetails.UserDetails` 接口。其中

User 与 Authority是多对多的关系。

```

1. @Entity // 实体
2. public class User implements UserDetails, Serializable {
3.
4.     private static final long serialVersionUID = 1L;
5.
6.     @Id // 主键
7.     @GeneratedValue(strategy = GenerationType.IDENTITY) // 自增长策略
8.     private Long id; // 用户的唯一标识
9.
10.    @Column(nullable = false, length = 50) // 映射为字段，值不能为空
11.    private String name;
12.
13.    @Column(nullable = false)
14.    private Integer age;
15.
16.    @Column(nullable = false, length = 50, unique = true)
17.    private String username; // 用户账号，用户登录时的唯一标识
18.
19.    @Column(length = 100)
20.    private String password; // 登录时密码
21.
22.    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

```

```

23.     @JoinTable(name = "user_authority", joinColumns =
        @JoinColumn(name = "user_id", referencedColumnName = "id"),
24.         inverseJoinColumns = @JoinColumn(name = "authority_id",
            referencedColumnName = "id"))
25.     private List<Authority> authorities;
26.
27.     protected User() { // JPA 的规范要求无参构造函数；设为 protected 防
        止直接使用
28.     }
29.
30.     public User(String name, Integer age) {
31.         this.name = name;
32.         this.age = age;
33.     }
34.
35.     public Long getId() {
36.         return id;
37.     }
38.
39.     public void setId(Long id) {
40.         this.id = id;
41.     }
42.
43.     public String getName() {
44.         return name;
45.     }
46.
47.     public void setName(String name) {
48.         this.name = name;
49.     }
50.
51.     public Integer getAge() {
52.         return age;
53.     }
54.
55.     public void setAge(Integer age) {
56.         this.age = age;
57.     }

```

```
58.
59.     @Override
60.     public Collection<? extends GrantedAuthority> getAuthorities()
61.     {
62.         return this.authorities;
63.     }
64.
65.     public void setAuthorities(List<Authority> authorities) {
66.         this.authorities = authorities;
67.     }
68.
69.     @Override
70.     public String getUsername() {
71.         return username;
72.     }
73.
74.     public void setUsername(String username) {
75.         this.username = username;
76.     }
77.
78.     @Override
79.     public String getPassword() {
80.         return password;
81.     }
82.
83.     public void setPassword(String password) {
84.         this.password = password;
85.     }
86.
87.     @Override
88.     public boolean isAccountNonExpired() {
89.         return true;
90.     }
91.
92.     @Override
93.     public boolean isAccountNonLocked() {
94.         return true;
95.     }
```

```

95.
96.     @Override
97.     public boolean isCredentialsNonExpired() {
98.         return true;
99.     }
100.
101.     @Override
102.     public boolean isEnabled() {
103.         return true;
104.     }
105.
106.     @Override
107.     public String toString() {
108.         return String.format("User[id=%d, username='%s', name='%s',
109.             age='%d', password='%s']", id, username, name, age,
110.                 password);
111.     }

```

新增 Authority

新增 Authority 用于存储权限信息。Authority 实现了 GrantedAuthority 接口，并重写了其 getAuthority 方法。

```

1.  @Entity // 实体
2.  public class Authority implements GrantedAuthority {
3.
4.      private static final long serialVersionUID = 1L;
5.
6.      @Id // 主键
7.      @GeneratedValue(strategy = GenerationType.IDENTITY) // 自增长策略
8.      private Long id; // 用户的唯一标识
9.
10.     @Column(nullable = false) // 映射为字段，值不能为空
11.     private String name;
12.
13.     public Long getId() {

```

```

14.         return id;
15.     }
16.
17.     public void setId(Long id) {
18.         this.id = id;
19.     }
20.
21.     /*
22.      * (non-Javadoc)
23.      *
24.      * @see
25.      org.springframework.security.core.GrantedAuthority#getAuthority()
26.      */
27.     @Override
28.     public String getAuthority() {
29.         return name;
30.     }
31.
32.     public void setName(String name) {
33.         this.name = name;
34.     }

```

新增存储库

新增 `com.waylau.spring.boot.security.repository.UserRepository`，由于继承了 `org.springframework.data.jpa.repository.JpaRepository` 接口，所以大部分与数据库操作的接口，我们在 `UserRepository` 都无需定义了。我们只需要定义一个

```

1. public interface UserRepository extends JpaRepository<User, Long>{
2.
3.     User findByUsername(String username);
4. }

```


`findByUsername` 是用来根据用户账号来查询用户，这个在查询用户权限时要用到。

服务类

我们的服务类继承自我们定义的UserService，以及 `org.springframework.security.core.userdetails.UserDetailsService`。其中，UserDetailsService是用来查询认证信息相关的接口，我们重写其 `loadUserByUsername` 方法。

```

1. @Service
2. public class UserServiceImpl implements
   UserService, UserDetailsService {
3.
4.     @Autowired
5.     private UserRepository userRepository;
6.
7.     .....
8.
9.     @Override
10.    public UserDetails loadUserByUsername(String username) throws
       UsernameNotFoundException {
11.        return userRepository.findByUsername(username);
12.    }
13.
14. }
```

修改配置类 SecurityConfig

在配置类中，我们把 UserDetailsService 的实现赋给 AuthenticationManagerBuilder：

```

1. @Autowired
2. private UserDetailsService userDetailsService;
```

```

3.  ....
4.
5.  /**
6.   * 认证信息管理
7.   * @param auth
8.   * @throws Exception
9.   */
10. @Autowired
11. public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception {
12.     auth.userDetailsService(userDetailsService);
13. }

```

初始化数据库

于是使用了 JPA ，所以，默认表结构是会根据实体逆向生成的。

至于数据，我们在项目的 `src/main/resources` 目录下，放置一个 `import.sql` 脚本文件。里面是我们要初始化的数据，只要应用启动，就会自动把 `import.sql` 的数据给导入数据库。

以下为我们要初始化的数据：

```

1. INSERT INTO user (id, username, password, name, age) VALUES (1,
    'waylau', '123456', '老卫', 30);
2. INSERT INTO user (id, username, password, name, age) VALUES (2,
    'admin', '123456', 'Way Lau', 29);
3.
4. INSERT INTO authority (id, name) VALUES (1, 'ROLE_USER');
5. INSERT INTO authority (id, name) VALUES (2, 'ROLE_ADMIN');
6.
7. INSERT INTO user_authority (user_id, authority_id) VALUES (1, 1);
8. INSERT INTO user_authority (user_id, authority_id) VALUES (2, 1);
9. INSERT INTO user_authority (user_id, authority_id) VALUES (2, 2);

```


基本认证

- 基本认证
 - `build.gradle`
 - `BasicAuthenticationFilter`
 - 配置
 - 运行
 - 如何注销账号

基本认证

基本认证在 Web 应用中是非常流行的认证机制。基本身份验证通常用于无状态客户端，它们在每个请求中传递其凭证。将其与基于表单的认证结合使用是很常见的，其中通过基于浏览器的用户界面和作为 Web 服务来使用应用。但是，基本认证将密码作为纯文本传输，是不安全的，所以它只能在真正通过加密的传输层（如 HTTPS）中使用。

在 `jpa-userdetailsservice` 项目的基础上，我们构建了一个 `basic-authentication` 项目。

`build.gradle`

修改 `build.gradle` 文件，让我们的 `basic-authentication` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'basic-authentication'  
3.     version = '1.0.0'  
4. }
```

BasicAuthenticationFilter

BasicAuthenticationFilter 负责处理 HTTP 标头中提供的基本身份认证的凭证。这可以用于认证由 Spring 远程协议（如 Hessian 和 Burlap）以及正常的浏览器用户代理（如 Firefox 和 Internet Explorer）进行的调用。管理 HTTP 基本认证的标准由 RFC 1945 第 11 节定义，并且 BasicAuthenticationFilter 符合该 RFC 标准。基本认证是一种有吸引力的身份验证方法，因为它在用户代理中的部署非常广泛，实现非常简单，它只是 `username:password` 的 Base64 编码，在 HTTP 头中指定即可。

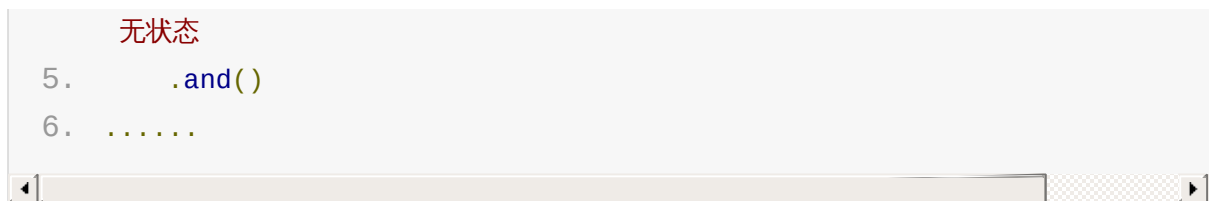
配置

AuthenticationManager 处理每个认证请求。如果认证失败，将使用配置的 AuthenticationEntryPoint 重试认证过程。通常，您将过滤器与 BasicAuthenticationEntryPoint 组合使用，它返回一个 401 响应与合适的头重试 HTTP 基本身份验证。如果认证成功，生成的 Authentication 对象将照常放入 SecurityContextHolder。

如果认证事件成功，或者未尝试认证，因为 HTTP 头不包含支持的认证请求，则过滤器链将正常继续。过滤器链将被中断的唯一时间点是认证失败并且调用 AuthenticationEntryPoint。

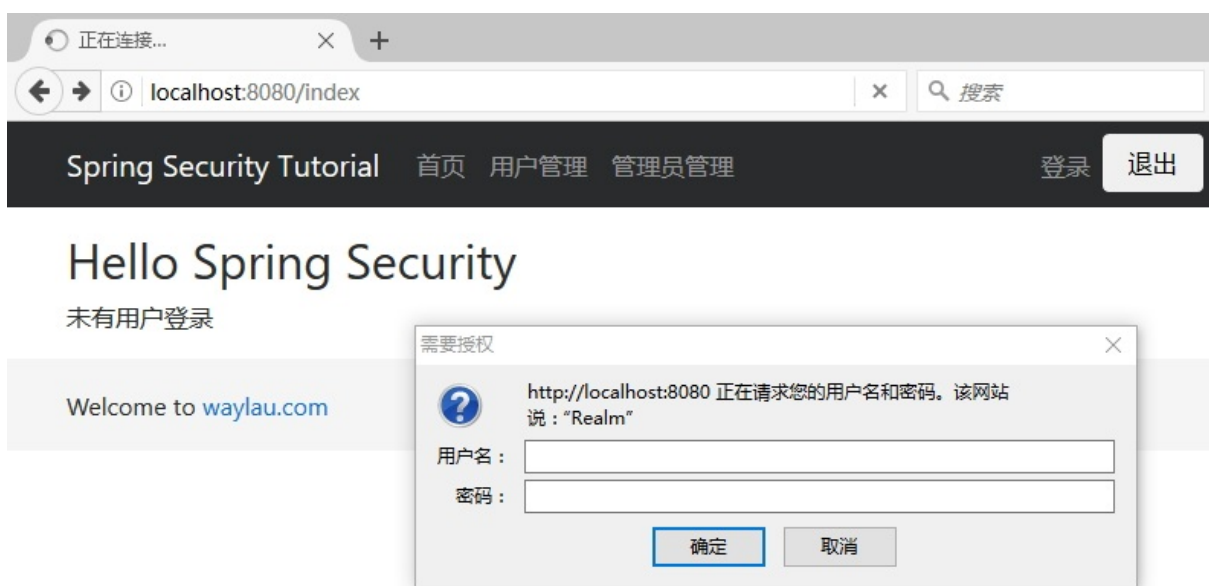
在配置类中，我们将 Form 表单认证改为 Basic 认证：

```
1. ....
2. .httpBasic()    // 使用 Basic 认证
3.     .and()
4. .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```



运行

当我们试图访问受限的资源时，浏览器会弹出输入框，要求我们输入账号密码：



输入之后，就可以在访问相关页面了。我们可以在响应头里面，看到账号密码经过 Base64 编码之后的认证信息：

▼ GET <http://localhost:8080/users>**头 响应 Cookie**

▼ 响应头

```

Cache-Control no-cache, no-store, max-age=0, must-revalidate
Content-Language zh-CN
Content-Type text/html; charset=UTF-8
Date Sun, 19 Mar 2017 14:14:07 GMT
Expires 0
Pragma no-cache
Set-Cookie JSESSIONID=B2D1B3C60FC9C1E6E72E537C2FE2A6C0; path=/; HttpOnly
Transfer-Encoding chunked
X-Content-Type-Options nosniff
X-Frame-Options SAMEORIGIN
x-xss-protection 1; mode=block

```

▼ 请求头

```

Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding gzip, deflate
Accept-Language zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
Authorization Basic d2F5bGF1OjEyMzQ1Ng==
Connection keep-alive
Cookie JSESSIONID=5CB717AB7C953CD166A48BD33A190A6B
DNT 1
Host localhost:8080
Referer http://localhost:8080/admins
Upgrade-Insecure-Requests 1

```

用户的状态信息都是保存在客户端（本例为浏览器），所以，即使后台服务器重启了，只要用户账号还在有效期内，就无需再次登录，即可再次访问服务。

如何注销账号

`HttpSecurity.logout()` 是清除 `HttpSession` 里面存储的用户信息。既然，我们是无状态（无会话），那么自然就无需调用 `logout()`。

如果是客户端是在浏览器，则直接关闭浏览器即可注销账号。

摘要认证

- 摘要认证
 - [build.gradle](#)
 - [Spring Security 的摘要认证](#)
 - [配置](#)
 - [运行](#)
 - [如何注销账号](#)

摘要认证

在 `basic-authentication` 项目的基础上，我们构建了一个 `digest-authentication` 项目。

build.gradle

修改 `build.gradle` 文件，让我们的 `digest-authentication` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'digest-authentication'  
3.     version = '1.0.0'  
4. }
```

Spring Security 的摘要认证

`DigestAuthenticationFilter` 能够处理在 HTTP 头中显示的摘要身份验证凭据。摘要认证尝试解决基本认证的许多弱点，特别是通过确保凭证不会以明文方式通过网络发送。许多用户代理支持摘要身份验

证，包括 Mozilla Firefox 和 Internet Explorer。管理 HTTP Digest 认证的标准由 RFC 2617 定义，RFC 2617 更新了较早版本的 RFC 2069 规定的摘要认证标准。大多数用户代理实现 RFC 2617。Spring Security 的 `DigestAuthenticationFilter` 与 RFC 2617 规定中的“auth”保护质量（qop）兼容，同时与 RFC 2069 向后兼。如果您需要使用未加密的 HTTP（即，没有 TLS/HTTPS）并希望最大化认证过程的安全性，摘要认证是非常有吸引力的选择。同时，摘要认证是 WebDAV 协议的强制性要求，如 RFC 2518 第 17.1 节所述：

你不应该在现代应用程序中使用摘要，因为它不被认为是安全的。最明显的问题是您必须以明文、加密或 MD5 格式存储密码。所有这些存储格式都被认为不安全。相反，您应该使用单向自适应密码散列（即 *bCrypt*、*PBKDF2*、*SCrypt* 等）。

摘要认证的中心是一个“随机数”，这是服务器生成的值。Spring Security 的随机数采用以下格式：

1. `base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))`
2. `expirationTime`: 随机数到期的日期和时间，以毫秒为单位
3. `key`: 用于防止随机数标记被修改的私钥

`DigestAuthenticationEntryPoint` 具有指定用于生成现时标记的密钥的属性，以及用于确定到期时间（默认 300，等于五分钟）的 `nonceValiditySeconds` 属性。虽然该随机数有效，但是通过连接各种字符串来计算摘要，这些字符串包括用户名、密码、随机数、被请求的 URI、客户端生成的随机数（只是用户代理生成每个请求的随机值）、领域名称等，然后执行 MD5 散列。服务器代理和用户代理都执行此摘要计算，如果它们对包含的值（例如密码）不同意，则导致不同的散列码。在 Spring Security 实现中，如果服务器生成的随机数已过期（但摘要有效），`DigestAuthenticationEntryPoint` 将发送一个“`stale=true`”头。这告诉用户代理没有必要打扰用户（因为密

码和用户名等是正确的)，而只是尝试再次使用一个新的随机数。

`DigestAuthenticationEntryPoint` 的 `nonceValiditySeconds` 参数的适当值取决于您的应用程序。在对安全非常重视的应用程序应该注意，截获的认证头可以用于模拟主体，直到到达随机数中包含的 `expirationTime`。这是选择适当设置的关键原则，但对于非常安全的应用程序，在第一次实例中不能通过 TLS / HTTPS 运行是不常见的。

由于 Digest 认证的更复杂的实现，常常有用户代理问题。例如，Internet Explorer 无法在同一会话中的后续请求上显示“opaque”标记。Spring Security 过滤器因此将所有状态信息封装到“nonce”令牌中。在我们的测试中，Spring Security 的实现可靠地使用 Mozilla Firefox 和 Internet Explorer，正确处理随机数超时等。

配置

要实现 HTTP 摘要认证，需要在过滤器链中定义 `DigestAuthenticationFilter`。同时需要配置 `UserDetailsService`，因为 `DigestAuthenticationFilter` 必须能够直接访问用户的明文密码。如果在 DAO 中使用编码密码，Digest 身份验证将不会工作（如果 `DigestAuthenticationFilter.passwordAlreadyEncoded` 设置为 true，则可以以 HEX (MD5 (username: realm: password)) 格式对密码进行编码。但是，其他密码编码将无法使用摘要身份验证。）。DAO 协作以及 `UserCache` 通常直接与 `DaoAuthenticationProvider` 共享。`authenticationEntryPoint` 属性必须为 `DigestAuthenticationEntryPoint`，以便

DigestAuthenticationFilter 可以获取正确的realmName 和摘要计算的键。

像 BasicAuthenticationFilter 一样，如果认证成功，那么认证请求令牌将被放入 SecurityContextHolder。如果认证事件成功，或者未尝试认证，因为HTTP头不包含摘要认证请求，则过滤器链将正常继续。过滤器链将被中断的唯一时机是如果认证失败并且调用了 AuthenticationEntryPoint。

摘要认证的 RFC 提供了一系列附加功能，以进一步提高安全性。例如，可以在每个请求时更改随机数。尽管如此，Spring Security 实现旨在最小化实现的复杂性（以及将出现的无疑的用户代理不兼容性），并避免需要存储服务器端状态。如果您想更详细地了解这些功能，请受邀查看RFC 2617。据我们所知，Spring Security的实现确实符合该 RFC 的最低标准。

在配置类中，我们启用摘要认证过滤器

DigestAuthenticationFilter，并自定义

DigestAuthenticationEntryPoint：

```
1. private static final String DIGEST_KEY = "waylau.com";
2. private static final String DIGEST_REALM_NAME = "spring security
   tutorial";
3. private static final int DIGEST_NONCE_VALIDITY_SECONDS = 240; // 过
   期时间 4 分钟
4.
5. @Autowired
6. private UserDetailsServiceImpl userDetailsServiceImpl;
7.
8. /**
9.  * 自定义 DigestAuthenticationEntryPoint
10.  *
11.  * @return
12.  */
```

```

13. @Bean
14. public DigestAuthenticationEntryPoint
    getDigestAuthenticationEntryPoint() {
15.     DigestAuthenticationEntryPoint digestEntryPoint = new
        DigestAuthenticationEntryPoint();
16.     digestEntryPoint.setKey(DIGEST_KEY);
17.     digestEntryPoint.setRealmName(DIGEST_REALM_NAME);
18.
        digestEntryPoint.setNonceValiditySeconds(DIGEST_NONCE_VALIDITY_SECONDS);
19.     return digestEntryPoint;
20. }
21.
22. /**
23.  * 摘要认证过滤器
24.  *
25.  * @param digestAuthenticationEntryPoint
26.  * @return
27.  * @throws Exception
28.  */
29. @Bean
30. public DigestAuthenticationFilter digestAuthenticationFilter(
31.     DigestAuthenticationEntryPoint
        digestAuthenticationEntryPoint) throws Exception {
32.
33.     DigestAuthenticationFilter digestAuthenticationFilter = new
        DigestAuthenticationFilter();
34.
        digestAuthenticationFilter.setAuthenticationEntryPoint(digestAuthenti
35.
        digestAuthenticationFilter.setUserDetailsService(userDetailsService);
36.     return digestAuthenticationFilter;
37. }

```

最终配置如下：

1.
2. http

```

3.         .authorizeRequests()
4.             .antMatchers("/css/**", "/js/**", "/fonts/**",
"/index").permitAll() // 都可以访问
5.             .antMatchers("/h2-console/**").permitAll() // 都可以访问
6.             .antMatchers("/users/**").hasRole("USER") // 需要相应的角色
才能访问
7.             .antMatchers("/admins/**").hasRole("ADMIN") // 需要相应的角
色才能访问
8.         .and()
9.
10.        .addFilter(digestAuthenticationFilter(getDigestAuthenticationEntryPoi
// 使用摘要认证过滤器
11.
12.        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STAT
无状态
13.        .and()
14.        .exceptionHandling().accessDeniedPage("/403") // 处理异常，拒绝访
问就重定向到 403 页面
15.        .authenticationEntryPoint(getDigestAuthenticationEntryPoint());
// 自定义 AuthenticationEntryPoint
16.        .....

```

运行

用户的状态信息都是保存在客户端（本例为浏览器），所以，即使后台服务器重启了，只要用户账号还在有效期内，就无需再次登录，即可再次访问服务。

当我们试图访问受限的资源时，浏览器会弹出输入框，要求我们输入账号密码：



输入之后，就可以在访问相关页面了。我们可以在响应头里面，看到摘要认证信息：

GET <http://localhost:8080/users> [HTTP/1.1 200 27ms]

头 响应 Cookie 调用堆栈

▼ 响应头

```

Cache-Control no-cache, no-store, max-age=0, must-revalidate
Content-Language zh-CN
Content-Type text/html; charset=UTF-8
Date Mon, 20 Mar 2017 07:26:08 GMT
Expires 0
Pragma no-cache
Set-Cookie JSESSIONID=60932307752E58A562F6EC889203E81E; path=/; HttpOnly
Transfer-Encoding chunked
X-Content-Type-Options nosniff
X-Frame-Options SAMEORIGIN
x-xss-protection 1; mode=block

```

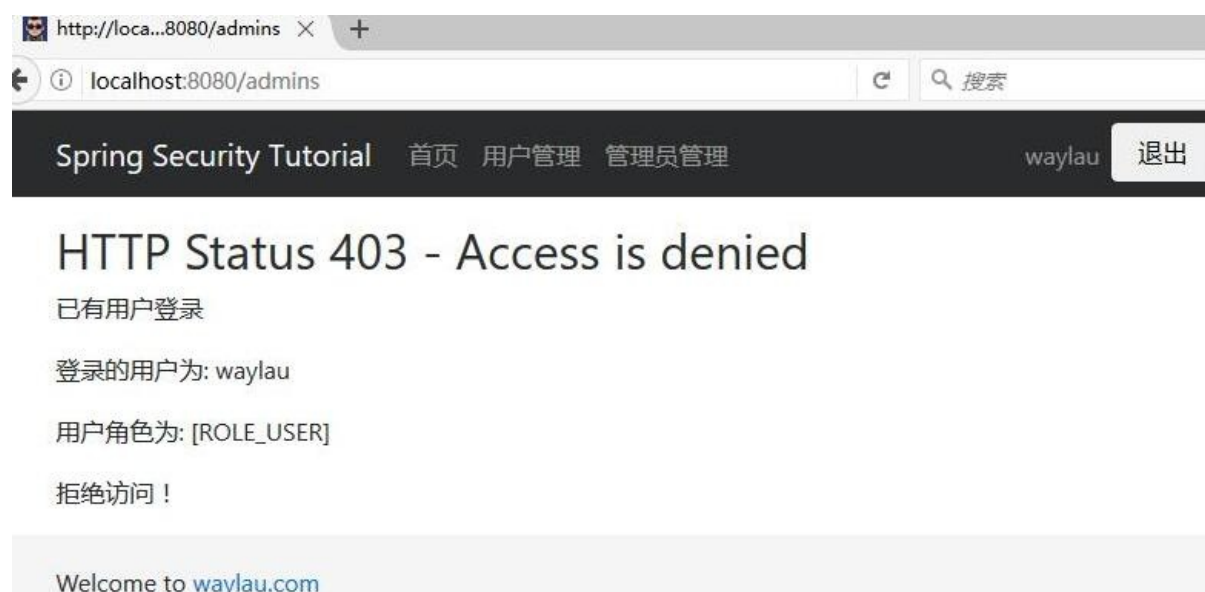
▼ 请求头

```

Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding gzip, deflate
Accept-Language zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
Authorization Digest username="waylau", realm="spring security tutorial", nonce="MTQ4OTk5NTAzMDEwMT03Y2M1YjA5YWESYjE4MjBmZmM4ZmY2MjczMWVjNTE2Nw==", uri="/users", response="f4f370741058341f8359bf202175664c", qop=auth, nc=00000010, cnonce="a2a02ab1a3de0d3d"
Connection keep-alive
Cookie JSESSIONID=7BC0603CD38222C9F9E439E94689E0E1
DNT 1
Host localhost:8080
Upgrade-Insecure-Requests 1
User-Agent Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0

```

当访问其他没有权限的资源时，会有相应的提示：



如何注销账号

`HttpSecurity.logout()` 是清除 `HttpSession` 里面存储的用户信息。既然，我们是无状态（无会话），那么自然就无需调用 `logout()`。

如果是客户端是在浏览器，则直接关闭浏览器即可注销账号。

摘要认证的密码加密

- 摘要认证的密码加密
 - `build.gradle`
 - 密码加密算法
 - 如果启用密码加密机制
 - 限制

摘要认证的密码加密

在之前的案例中，我们的密码都是以明文形式存储在数据库中，明文存储给系统安全带来了极大的风险。本节将演示在摘要认证中，实现对密码进行加密存储。

在 `digest-authentication` 项目的基础上，我们构建了一个 `digest-password-encode` 项目。

`build.gradle`

修改 `build.gradle` 文件，让我们的 `digest-password-encode` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'digest-password-encode'  
3.     version = '1.0.0'  
4. }
```

密码加密算法

Spring Security 所使用的密码加密算法格式为 `HEX(MD5(username`

`password))` 所以，当我们使用账号 waylau 密码 123456 时，生成的密码如下：

```
1. waylau:spring security tutorial:123456 ->
   b7ace5658b44f7295e7e8e36da421502
```

具体生成的密码的代码可以看 `ApplicationTests.java`：

```
1. @Test
2. public void testGenerateDigestEncodePassword() {
3.     String username = "waylau";
4.     String realm = "spring security tutorial";
5.     String password = "123456";
6.
7.     String a1Md5 = this.encodePasswordInA1Format(username, realm,
8. password);
9.
10.    System.out.println("a1Md5:" + a1Md5);
11. }
12. private String encodePasswordInA1Format(String username, String
13. realm, String password) {
14.     String a1 = username + ":" + realm + ":" + password;
15.
16.     return md5Hex(a1);
17. }
18. private String md5Hex(String data) {
19.     MessageDigest digest;
20.     try {
21.         digest = MessageDigest.getInstance("MD5");
22.     }
23.     catch (NoSuchAlgorithmException e) {
24.         throw new IllegalStateException("No MD5 algorithm
25.         available!");
26.     }
27. }
```

```

25.     }
26.
27.     return new String(Hex.encode(digest.digest(data.getBytes())));
28. }

```

这样，我们的数据库中存储加密后的密码，这样，就避免了明文存储的风险。

在初始化用户时，我们把加密后的密码存储进数据库：

```

1. INSERT INTO user (id, username, password, name, age) VALUES (1,
   'waylau', 'b7ace5658b44f7295e7e8e36da421502', '老卫', 30);
2. INSERT INTO user (id, username, password, name, age) VALUES (2,
   'admin', 'b7b20c789238e2a46e56b533c87e673c', 'Way Lau', 29);

```

如果启用密码加密机制

`DigestAuthenticationFilter.passwordAlreadyEncoded` 设置为 `true` 即可：

```

1. @Bean
2. public DigestAuthenticationFilter digestAuthenticationFilter(
3.     DigestAuthenticationEntryPoint
   digestAuthenticationEntryPoint) throws Exception {
4.
5.     DigestAuthenticationFilter digestAuthenticationFilter = new
   DigestAuthenticationFilter();
6.
7.     digestAuthenticationFilter.setAuthenticationEntryPoint(digestAuthenti
8.     digestAuthenticationFilter.setUserDetailsService(userDetailsService);
9.     digestAuthenticationFilter.setPasswordAlreadyEncoded(true); //
   密码已经加密
10.    return digestAuthenticationFilter;

```

限制

除默认的加密方式外，不能在摘要认证上采用其他的密码加密方式。

通用密码加密

- 通用密码加密
 - `build.gradle`
 - MD5 算法
 - 盐值加密
 - 生成加密后的密码

通用密码加密

在“摘要认证的密码加密”一节，我们讲到了如何在摘要认证中使用密码加密。但同时也提到了一些限制，比如，该加密方式只能适用于摘要认证，而且摘要认证，只能采用该种方式进行密码加密。那么，我们如何在其他认证类型（如基本认证、Form表单认证）中进行用户信息的加密呢？本节为你揭开谜底。

在 `basic-authentication` 项目的基础上，我们构建了一个 `password-encoder` 项目。

`build.gradle`

修改 `build.gradle` 文件，让我们的 `password-encoder` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'password-encoder'  
3.     version = '1.0.0'  
4. }
```

MD5 算法

任何一个正式的企业应用中，都不会在数据库中使用明文来保存密码的，我们在之前的章节中都是为了方便起见没有对数据库中的用户密码进行加密，这在实际应用中是极为幼稚的做法。可以想象一下，只要有人进入数据库就可以看到所有人的密码，这是一件多么恐怖的事情，为此我们至少要对密码进行加密，这样即使数据库被攻破，也可以保证用户密码的安全。

最常用的方法是使用MD5算法对密码进行摘要加密，这是一种单项加密手段，无法通过加密后的结果反推回原来的密码明文。

首先我们要把数据库中原来保存的密码使用MD5进行加密：

```
1. 123456 ---MD5--> e10adc3949ba59abbe56e057f20f883e
```

现在密码部分已经面目全非了，即使有人攻破了数据库，拿到这种“乱码”也无法登陆系统窃取客户的信息。

盐值加密

实际上，上面的实例在现实使用中还存在着一个不小的问题。虽然MD5 算法是不可逆的，但是因为它对同一个字符串计算的结果是唯一的，所以一些人可能会使用“字典攻击”的方式来攻破 MD5 加密的系统。这虽然属于暴力解密，却十分有效，因为大多数系统的用户密码都不会很长。

实际上，大多数系统都是用 admin 作为默认的管理员登陆密码，所以，当我们在数据库中看到“21232f297a57a5a743894a0e4a801fc3”时，就可以意识到 admin 用户使用的密码了。因此，MD5 在处理这种常用字符串时，并

不怎么奏效。

```
1. admin ---MD5--> 21232f297a57a5a743894a0e4a801fc3
```

为了解决这个问题，我们可以使用盐值加密“salt-source”。

修改配置文件：

1. ...
- 2.
3. 在password-encoder下添加了salt-source，并且指定使用 username 作为盐值。
- 4.
5. 盐值的原理非常简单，就是先把密码和盐值指定的内容合并在一起，再使用 MD5 对合并后的内容进行演算，这样一来，就算密码是一个很常见的字符串，再加上用户名，最后算出来的 MD5 值就没那么容易猜出来了。因为攻击者不知道盐值的值，也很难反算出密码原文。
- 6.
7. 我们这里将每个用户的 username 作为盐值，最后数据库中的密码部分就变成了这样：

admin123456 --MD5-->

a66abb5684c45962d887564f08346e8d

- 1.
2. **## PasswordEncoder 的实现**
- 3.
4. 加密实现方式都实现自
`org.springframework.security.crypto.password.PasswordEncoder`接口，
有AbstractPasswordEncoder、BCryptPasswordEncoder、
NoOpPasswordEncoder、
Pbkdf2PasswordEncoder、SCryptPasswordEncoder、StandardPasswordEncoder。
常用的有：
- 5.
6. * NoOpPasswordEncoder：按原文本处理，相当于不加密。
7. * StandardPasswordEncoder：1024次迭代的 SHA-256 散列哈希加密实现，并使用一个随机8字节的salt。
8. * BCryptPasswordEncoder：使用BCrypt的强散列哈希加密实现，并可以由客户端指定加密的强度strength，强度越高安全性自然就越高，默认为10。
- 9.

```

10. 在Spring Security 的注释中，明确写明了如果是开发一个新的项目，BCryptPasswordEncoder 是较好的选择。本节示例也是采用BCryptPasswordEncoder 方式。
11.
12. ```java
13. @Autowired
14. private PasswordEncoder passwordEncoder;
15.
16. @Bean
17. public PasswordEncoder passwordEncoder() {
18.     return new BCryptPasswordEncoder(); // 使用 BCrypt 加密
19. }

```

另外，我们需要自定义一个 DaoAuthenticationProvider，来将我们的加密方式进行注入：

```

1. @Autowired
2. private AuthenticationProvider authenticationProvider;
3.
4. @Bean
5. public AuthenticationProvider authenticationProvider() {
6.     DaoAuthenticationProvider authenticationProvider = new
        DaoAuthenticationProvider();
7.
8.     authenticationProvider.setUserDetailsService(userDetailsService);
9.     authenticationProvider.setPasswordEncoder(passwordEncoder); //
        设置密码加密方式
10.     return authenticationProvider;

```

同时，将上述的 DaoAuthenticationProvider 设置进 AuthenticationManagerBuilder ：

```

1. @Autowired
2. public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception {

```

```

3.     auth.authenticationProvider(authenticationProvider);
4. }

```

生成加密后的密码

具体生成的密码的代码可以看 `ApplicationTests.java`:

```

1. @Test
2. public void testBCryptPasswordEncoder() {
3.
4.     CharSequence rawPassword = "123456";
5.
6.     PasswordEncoder encoder = new BCryptPasswordEncoder();
7.     String encodePasswd = encoder.encode(rawPassword);
8.     boolean isMatch = encoder.matches(rawPassword, encodePasswd);
9.     System.out.println("encodePasswd:" + encodePasswd);
10.    System.out.println(isMatch);
11. }

```

这样，我们的数据库中存储加密后的密码，这样，就避免了明文存储的风险。

在初始化用户时，我们把加密后的密码存储进数据库：

```

1. INSERT INTO user (id, username, password, name, age) VALUES (1,
    'waylau',
    '$2a$10$N.zmdr9k7uOCQb376NoUnuTJ8iAt6Z5EHsM8lE9lB0sl7iKTVKIUi', '老
    卫', 30);
2. INSERT INTO user (id, username, password, name, age) VALUES (2,
    'admin',
    '$2a$10$N.zmdr9k7uOCQb376NoUnuTJ8iAt6Z5EHsM8lE9lB0sl7iKTVKIUi',
    'Way Lau', 29);

```


Remember-Me（记住我）认证：基于散列的令牌方法

- [Remember-Me（记住我）认证：基于散列的令牌方法](#)
 - [build.gradle](#)
 - [Remember-Me（记住我）认证](#)
 - [简单的基于散列的令牌方法](#)
 - [TokenBasedRememberMeServices](#)
 - [配置](#)
 - [设置 Remember-Me 选择项](#)

Remember-Me（记住我）认证：基于散列的令牌方法

我们在 `password-encoder` 上，基于 Form 表单的方式，来实现基于散列的令牌方法的 Remember-Me 认证，我们新建一个 `remember-me-hash` 项目。

build.gradle

修改 `build.gradle` 文件，让我们的 `remember-me-hash` 项目成为一个新的项目。

修改内容也比较简单，修改项目名称及版本即可。

```
1. jar {  
2.     baseName = 'remember-me-hash'  
3.     version = '1.0.0'  
4. }
```

Remember-Me（记住我）认证

Remember-Me或持久的登录身份验证是指网站能够记住身份之间的会话。这通常是通过发送 cookie 到浏览器，cookie 在未来会话中被检测到，并导致自动登录发生。Spring Security 为这些操作提供了必要的钩子，并且有两个具体的实现。

- 使用散列来保存基于 cookie 的令牌的安全性
- 使用数据库或其他持久存储机制来存储生成的令牌

需要注意的是，这些实现都需要 `UserDetailsService`。如果您使用的是一种身份验证提供程序不使用 `UserDetailsService`（例如，LDAP 程序），这样该机制就不会正常工作，除非在你的应用程序上下文中有 `UserDetailsService` bean。

简单的基于散列的令牌方法

这种方法使用散列实现一个有用的 Remember-Me 的策略。其本质是，在认证成功后，cookie 被发送到浏览器进行交互。cookie 的组成如下：

```
1. base64(username + ":" + expirationTime + ":" +
2. md5Hex(username + ":" + expirationTime + ":" password + ":" + key))
3.
4. username:      UserDetailsService 中的身份标识
5. password:      UserDetails 中的密码
6. expirationTime: 随机数到期的日期和时间，以毫秒为单位
7. key:           用于防止随机数标记被修改的私钥
```

因此，Remember-Me 令牌仅适用于指定的期间，并且提供的用户名、密码和密钥不会更改。值得注意的是，这有一个潜在的安全问题，即任何用户代理只要捕获了 Remember-Me 令牌就能一直使用直到令牌过期。这是与摘要验证存在的相同的问题。如果一个认证主体意识到令牌

已被截获，他们可以通过密码，来将之前的 Remember-Me 令牌作废。如果需要更重要的安全性，您应该使用下一节中描述的方法。另外 Remember-Me 的服务根本不应该使用。

TokenBasedRememberMeServices

TokenBasedRememberMeServices 产生 RememberMeAuthenticationToken，并由 RememberMeAuthenticationProvider 处理。这种身份验证提供者与 TokenBasedRememberMeServices 之间共享 key。此外，TokenBasedRememberMeServices 需要从它可以检索签名比较目的的用户名和密码 UserDetailsService，生成的 RememberMeAuthenticationToken 包含正确的 GrantedAuthority。当用户使用无效的 Cookie 发起请求时，注销命令将由应用程序提供。TokenBasedRememberMeServices 还实现了 Spring Security 的 LogoutHandler 接口，可以用 LogoutFilter 自动清除 Cookie。

配置

实现 Remember-Me 较为简单，只需要添加 `.rememberMe().key(KEY)` 即可，其中 KEY 是自定义的密钥值。完整配置如下：

```
1. private static final String KEY = "waylau.com";
2. ....
3. @Override
4. protected void configure(HttpSecurity http) throws Exception {
5.     http.authorizeRequests().antMatchers("/css/**", "/js/**",
6.         "/fonts/**", "/index").permitAll() // 都可以访问
7.         .antMatchers("/h2-console/**").permitAll() // 都可以访问
```

```

7.          .antMatchers("/users/**").hasRole("USER") // 需要相应的角色才能访问
8.          .antMatchers("/admins/**").hasRole("ADMIN") // 需要相应的角色才能访问
9.          .and()
10.         .formLogin() //基于 Form 表单登录验证
11.         .loginPage("/login").failureUrl("/login-error") // 自定义登录界面
12.         .and().rememberMe().key(KEY) // 启用 remember me
13.         .and().exceptionHandling().accessDeniedPage("/403");
// 处理异常，拒绝访问就重定向到 403 页面
14.         http.csrf().ignoringAntMatchers("/h2-console/**"); // 禁用 H2 控制台的 CSRF 防护
15.         http.headers().frameOptions().sameOrigin(); // 允许来自同一来源的 H2 控制台的请求
16.     }
17.     .....

```

设置 Remember-Me 选择项

在登录界面，我们设置一个设置 Remember-Me 选择项。

```

1.     .....
2.     <div>
3.         <label for="remember-me">记住我</label>
4.         <input type="checkbox" name="remember-me" id="remember-me">
5.     </div>
6.     .....

```

选择Remember-Me 选择项进行登录，则即使服务器重启，下次登录系统仍无需重新登录系统。

以下为登录界面：

登录后，可以看到 Cookie 里面看到 Remember-Me 的令牌值：

▼ POST <http://localhost:8080/login> [HTTP/1.1 302 486ms]

头 POST 响应 Cookie

▼ 响应头

```

Cache-Control no-cache, no-store, max-age=0, must-revalidate
Content-Length 0
Date Tue, 21 Mar 2017 14:29:31 GMT
Expires 0
Location http://localhost:8080/users
Pragma no-cache
Set-Cookie JSESSIONID=690C0EB109F439649BAD63E328119124;path=/;HttpOnly
remember-me=YwRtaW46MTQ5MTMxNjE3MTc0MTo4NmQ5OWZiYmQ1NzA2YmMyMTAxNzVkM2Y3YmFhZmZkMg;Max-Age=1209600;path=/;H
ttpOnly
X-Content-Type-Options nosniff
X-Frame-Options SAMEORIGIN
x-xss-protection 1; mode=block

```

▼ 请求头

```

Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding gzip, deflate
Accept-Language zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
Connection keep-alive
Cookie JSESSIONID=12A7B35012EE76ED713DA0676116D76E
DNT 1
Host localhost:8080
Referer http://localhost:8080/login
Upgrade-Insecure-Requests 1
User-Agent Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0

```


Remember-Me（记住我）认证：基于持久化的令牌方法

- Remember-Me（记住我）认证：基于持久化的令牌方法
 - 基于持久化的令牌方法
 - PersistentTokenBasedRememberMeServices
 - 配置
 - 运行

Remember-Me（记住我）认证：基于持久化的令牌方法

基于持久化的令牌方法

这种方法使用数据库来存储令牌信息。这种方法是基于http://jaspan.com/improved_persistent_login_cookie_best_practice 文章所做出的小修改。使用这种方法可以提供一个数据源引用：

```
1. create table persistent_logins (username varchar(64) not null,  
2.                               series varchar(64) primary key,  
3.                               token varchar(64) not null,  
4.                               last_used timestamp not null)
```

PersistentTokenBasedRememberMeService

这个类可以使用相同的方式 TokenBasedRememberMeServices，但它还需要配置一个 PersistentTokenRepository 来存储令牌。有两个标准实现。

- InMemoryTokenRepositoryImpl : 仅用于测试。
- JdbcTokenRepositoryImpl : 存储令牌到数据库中。

本例，我们将使用 JdbcTokenRepositoryImpl 将令牌存储到数据库中：

```

1. @Autowired
2. private DataSource dataSource;
3.
4. @Bean
5. public JdbcTokenRepositoryImpl tokenRepository() {
6.     JdbcTokenRepositoryImpl tokenRepository = new
       JdbcTokenRepositoryImpl();
7.     tokenRepository.setCreateTableOnStartup(true);    // 启动时自动建
       表，但重启数据会丢失
8.     tokenRepository.setDataSource(dataSource);
9.     return tokenRepository;
10. };

```

其中 setCreateTableOnStartup 方法是可选的，如果设置为 true ，则会自动创建 persistent_logins 表结构，但缺点是，原有的数据将会丢失。

配置

在 `.rememberMe().key(KEY).tokenRepository(tokenRepository())` 指明 PersistentTokenRepository 的实现方式。

完整配置如下：

```

1. @Override
2. protected void configure(HttpSecurity http) throws Exception {
3.     http.authorizeRequests().antMatchers("/css/**", "/js/**",
       "/fonts/**", "/index").permitAll() // 都可以访问
4.     .antMatchers("/h2-console/**").permitAll() // 都可以访问

```

```

5.          .antMatchers("/users/**").hasRole("USER") // 需要相应的角色才能访问
6.          .antMatchers("/admins/**").hasRole("ADMIN") // 需要相应的角色才能访问
7.          .and()
8.          .formLogin() //基于 Form 表单登录验证
9.          .loginPage("/login").failureUrl("/login-error") // 自定义登录界面
10.
11.         .and().exceptionHandling().accessDeniedPage("/403");
           // 处理异常，拒绝访问就重定向到 403 页面
12.         http.csrf().ignoringAntMatchers("/h2-console/**"); // 禁用 H2 控制台的 CSRF 防护
13.         http.headers().frameOptions().sameOrigin(); // 允许来自同一来源的 H2 控制台的请求
14.     }

```

运行

登录后，可以看到 Cookie 里面看到 Remember-Me 的令牌值：

▼ GET <http://localhost:8080/users> [HTTP/1.1 200 20ms]

头	响应	Cookie
▼ 响应头		
Cache-Control	no-cache, no-store, max-age=0, must-revalidate	
Content-Language	zh-CN	
Content-Type	text/html; charset=UTF-8	
Date	Tue, 21 Mar 2017 17:50:24 GMT	
Expires	0	
Pragma	no-cache	
Transfer-Encoding	chunked	
X-Content-Type-Options	nosniff	
X-Frame-Options	SAMEORIGIN	
x-xss-protection	1; mode=block	
▼ 请求头		
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8	
Accept-Encoding	gzip, deflate	
Accept-Language	zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3	
Connection	keep-alive	
Cookie	JSESSIONID=10FB8AC552ABE907ADE64273C960354D; remember-me=ZnJodk4xQ1FZaFN5MTdjamI2NX1qZz09OnMvbHN6MEpjcDVseDcvZ11ONXh2aWc9PQ	
DNT	1	
Host	localhost:8080	
Referer	http://localhost:8080/users	
Upgrade-Insecure-Requests	1	
User-Agent	Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0	

我们也可以在数据中看到这个登录信息：



The screenshot shows the H2 console interface. The address bar displays the URL: `localhost:8080/h2-console/login.do?jsessionid=30d2c07ca3603d3f5f9b59a87980beb7`. The left sidebar shows the database structure for `jdbc:h2:mem:testdb`, including tables like `AUTHORITY`, `PERSISTENT_LOGINS`, `USER`, `USER_AUTHORITY`, and the `INFORMATION_SCHEMA` folder. The main area shows the SQL statement `SELECT * FROM PERSISTENT_LOGINS` being executed. The result is displayed as a table with 4 columns: `USERNAME`, `SERIES`, `TOKEN`, and `LAST_USED`. The data shows a single row for user `waylau` with a specific series and token, last used on 2017-03-22. Below the table, it indicates `(1 row, 3 ms)`. There is an `Edit` button at the bottom.

USERNAME	SERIES	TOKEN	LAST_USED
waylau	frhvN1CQYhSy17cjb65yjjg==	s/lSz0Jcp5lx7/fYN5xvig==	2017-03-22 01:50:19.591

当用户注销后，该数据信息自动会删除。

OAuth 2.0 认证的原理与实践

- OAuth 2.0 认证的原理与实践

OAuth 2.0 认证的原理与实践

详见<https://waylau.com/principle-and-practice-of-oauth2>

参考资料

- [参考文献](#)

参考文献

- <http://docs.spring.io/spring-security/site/docs/4.2.2.RELEASE/reference/htmlsingle/#what-is-acegi-security>
- [https://msdn.microsoft.com/en-us/library/ie/gg622941\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx)
- <https://en.wikipedia.org/wiki/Clickjacking>
- [https://msdn.microsoft.com/en-us/library/dd565647\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx)
- <http://docs.spring.io/spring-security/site/docs/4.2.2.RELEASE/reference/htmlsingle/#digest-config>
- http://jaspan.com/improved_persistent_login_cookie_best_practice
- <https://tools.ietf.org/html/rfc7519>
- <https://developer.github.com/v3/oauth/>
- <https://spring.io/guides/tutorials/spring-boot-oauth2/>
- <https://tools.ietf.org/html/rfc6749>