# GIT: https://github.com/troger19/fitness-tracker.git

# Intro

**Authentication**: Who Am I?

**Authorization**: What can I do?

# MAVEN

Need to add three jars in addition to the Spring jars

□ spring-security-web

□ spring-security-config

□ commons-logging

<**dependency**>  
 <**groupId**>org.springframework.security</**groupId**>  
 <**artifactId**>spring-security-web</**artifactId**>  
 <**version**>4.2.3.RELEASE</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.springframework.security</**groupId**>  
 <**artifactId**>spring-security-config</**artifactId**>  
 <**version**>4.2.3.RELEASE</**version**>  
</**dependency**>

<**dependency**>  
 <**groupId**>commons-logging</**groupId**>  
 <**artifactId**>commons-logging</**artifactId**>  
 <**version**>1.2</**version**>  
</**dependency**>

We need to bootstrap our context.Bootstrap mean start by itself, start reading own configuration.

Put this inside webapp/WEB-INF/web.xml

1. Create filter with name you want, instance of DelegatingFilterProxy. Create also mapping to this filter. /\* means every request go through this filter

<**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**>

1. Create context listener. It bootstrap the spring configuration for us.   
     
   <**listener**>  
    <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  
   </**listener**>
2. Create context params. They are used by listener to find the configuration. contextConfigLocation is a name of the param, the ContextLoaderListener is looking for, don’t change the name.   
     
   <**context-param**>  
    <**param-name**>contextConfigLocation</**param-name**>  
    <**param-value**>/WEB-INF/config/security-config.xml</**param-value**>  
   </**context-param**>

To keep the Separation of Concern design pattern, we define security configurion separate to web configuration. WEB-INF/config/security-config.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans:beans xmlns:beans="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:p="http://www.springframework.org/schema/p"  
 xmlns="http://www.springframework.org/schema/security"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/security  
 http://www.springframework.org/schema/security/spring-security.xsd"**>  
  
  
 <**http auto-config="true"**>  
 <**intercept-url pattern="/\*\*" access="ROLE\_USER"** />  
 </**http**>  
  
 <**authentication-manager**>  
 <**authentication-provider**>  
 <**user-service**>  
 <**user name="bryan" password="secret" authorities="ROLE\_USER"**/>  
 </**user-service**>  
 </**authentication-provider**>  
 </**authentication-manager**>  
  
</**beans:beans**>

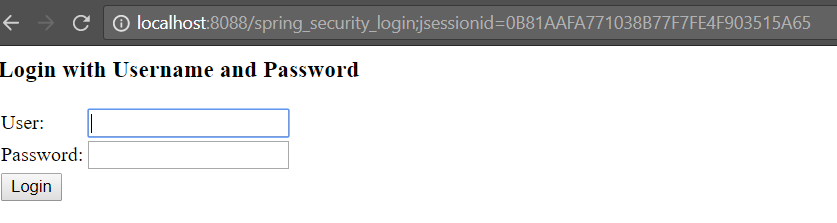
This specifies that every URL must have be accessed by someone that has the authority of ROLE\_USER.

<**intercept-url pattern="/\*\*" access="ROLE\_USER"** />

This creates user, which must have the right assess type to access the URL.

<**user name="bryan" password="secret" authorities="ROLE\_USER"**/>

Now the website is secured and Login and logout forms are ready and protect every URL. Once the user logs in, the is no need to login again and he can surf the site.



# Database setup

## Authentification

1. Create authentification tables:

**CREATE TABLE** users (  
 **username VARCHAR**(50) **NOT NULL PRIMARY KEY**,  
 **password VARCHAR**(50) **NOT NULL**,  
 **enabled BOOLEAN NOT NULL**);  
  
**CREATE TABLE** authorities (  
 **username VARCHAR**(50) **NOT NULL**,  
 **authority VARCHAR**(50) **NOT NULL**,  
 **CONSTRAINT** fk\_authorities\_users  
 **FOREIGN KEY** (**username**) **REFERENCES** users (**username**)  
);  
  
**CREATE UNIQUE INDEX** ix\_auth\_username  
 **ON** authorities (**username**, **authority**);  
  
**INSERT INTO** users (**username**, **password**, **enabled**) **VALUES** ("bryan", "secret", **TRUE**);  
**INSERT INTO** authorities (**username**, **authority**) **VALUES** ("bryan", "ROLE\_USER");

1. Add Maven dependencies

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.1.44</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-jdbc</**artifactId**>  
 <**version**>3.2.0.RELEASE</**version**>  
</**dependency**>

1. Change security-config.xml
   1. Create datasource

<**beans:bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  
 <**beans:property name="driverClassName" value="com.mysql.jdbc.Driver"**/>  
 <**beans:property name="url" value="jdbc:mysql://localhost:3306/fitness\_tracker"**/>  
 <**beans:property name="username" value="root"**/>  
 <**beans:property name="password" value="Miesacka22"**/>  
</**beans:bean**>

* 1. Create service for login : user details and assign the datasource to it

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

* 1. Link the service to authentication provider

<**authentication-manager**>  
 <**authentication-provider user-service-ref="userDetailService"**/>  
</**authentication-manager**>

## Authorization

1. Add taglibs dependencies to pom.xml

<**dependency**>  
 <**groupId**>org.springframework.security</**groupId**>  
 <**artifactId**>spring-security-taglibs</**artifactId**>  
 <**version**>3.2.0.RELEASE</**version**>  
</**dependency**>

1. Add standard tag library in jsp for spring security.. JSTL in index.jsp

<%@**taglib prefix**="**sec**" **uri**="**http://www.springframework.org/security/tags**" %>

Now, you can use this prefix to get some info about the logged in user. This will return the name of the user.

<**sec:authentication property="name"**/>

1. Wrap the elements that you want to restrict the access

<**sec:authorize ifAnyGranted="ROLE\_ADMIN"**>  
<**a class="btn btn-primary" href="editGoal.html"**> Edit Goal »</**a**>  
</**sec:authorize**>

1. Create the database user with ADMIN role. Add him also USER role, because we have restriction for all pages to have at least USER role. Or you can extend the filter in spring-security.xml. Otherwise you will get 403.

<**intercept-url pattern="/\*\*" access="ROLE\_USER"** />

**INSERT INTO** users (**username**, **password**, **enabled**) **VALUES** ("cris", "secret", **TRUE**);  
**INSERT INTO** authorities (**username**, **authority**) **VALUES** ("cris", "ROLE\_USER");

**INSERT INTO** authorities (**username**, **authority**) **VALUES** ("cris", "ADMIN\_USER");

## Encryption

Passwords are stored in DB as a plain text, so at least encrypt them somehow.

### MD5

1. You can write unites to check the MD5 values of some passwords.   
     
    **public void** testMD5Hash() {  
    String password = **"secret"**;  
    Md5PasswordEncoder passwordEncoder = **new** Md5PasswordEncoder();  
    String hashedPassword = passwordEncoder.encodePassword(password, **null**);  
    System.***out***.println(hashedPassword);  
   }

Output: 5ebe2294ecd0e0f08eab7690d2a6ee69

1. Change the password in DB. From secret to 5ebe2294ecd0e0f08eab7690d2a6ee69
2. Define the password-encoder for authentication-provider

<**authentication-manager**>  
 <**authentication-provider user-service-ref="userDetailService"**>  
 **<password-encoder hash="md5"/>**  
 </**authentication-provider**>  
</**authentication-manager**>

1. Now, when you try to login with bryan/secret, it should work. But cris/secret doesn’t work, because we haven’t change its value to MD5 value.

### BCrypt

MD5 is still weak, you can google it to find the decrypted value. BCrypt add salt which helps security but also add complexity because password are 60 character long.

1. Alter table

**ALTER TABLE** users **MODIFY password VARCHAR**(60) **NOT NULL**;

1. Change password encoder for bcrypt

<**password-encoder hash="bcrypt"**/>

1. Create unit test to get Bcrypt value for our password.

**public void** testBCryptHash() {  
 String password = **"secret"**;  
 BCryptPasswordEncoder passwordEncoder = **new** BCryptPasswordEncoder();  
 String hashedPassword = passwordEncoder.encode(password);  
 System.***out***.println(hashedPassword);  
}

# Customizing Login

1. For basic login just add http-basic element in http

<**http auto-config="true"**>  
 <**intercept-url pattern="/\*\*" access="ROLE\_USER"** />  
 **<http-basic/>**  
</**http**>

1. Add form login page and second interceptor, to have login page accessible

<**http auto-config="true"**>  
 **<intercept-url pattern="/login.html" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>** <**intercept-url pattern="/\*\*" access="ROLE\_USER"** />  
 **<form-login login-page="/login.html"/>**  
 <**http-basic**/>  
</**http**>

1. Create Login Controller

@Controller  
**public class** LoginController {  
  
 @RequestMapping(value = **"/login"**, method = RequestMethod.***GET***)  
 **public** String login(ModelMap model){  
 System.***out***.println(**"We are in login "**);  
 **return "login"**;  
 }  
}

value = **"/login"**, 🡪 we don’t need to put .html, because we tell spring to route each request through Spring MVC

<**servlet-mapping**>  
 <**servlet-name**>fitTrackerServlet</**servlet-name**>  
 <**url-pattern**>\*.html</**url-pattern**>  
</**servlet-mapping**>

**return "login"**; 🡪 this tell Spring to use InternalResolver to look for whatever String we returned, add .jsp and look in directory WEB/jsp

servlet-config.xml

<**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"   
 p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" p:order="2"**/>

1. Create login.jsp page

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<html>  
<head>  
 <title>Fitness Tracker Custom Login Page</title>  
 <style type="text/css">  
 .errorblock {  
 color: #ff0000;  
 background-color: #ffEEEE;  
 border: 3px solid #ff0000;  
 padding: 8px;  
 margin: 16px;  
 }  
 </style>  
</head>  
  
<body onload='*document*.f.j\_username.focus();'>  
<h3>Fitness Tracker Custom Login Page</h3>  
<c:if test="${not empty error}">  
 <div class="errorblock">  
 Your login was unsuccessful. <br />  
 Caused: ${sessionScope["SPRING\_SECURITY\_LAST\_EXCEPTION"].message }  
 </div>  
</c:if>  
  
<form action="j\_spring\_security\_check" name="f" method="post">  
 <table>  
 <tr>  
 <td>User:</td>  
 <td><input type="text" name="j\_username" value=""></td>  
 </tr>  
 <tr>  
 <td>Password:</td>  
 <td><input type="password" name="j\_password" ></td>  
 </tr>  
 <tr>  
 <td colspan="2"><input type="submit" name="Submit" value="Submit"></td>  
 </tr>  
 <tr>  
 <td colspan="2"><input type="reset" name="reset" > </td>  
 </tr>  
 </table>  
</form>  
</body>  
</html>

1. Error page. In security-config.xml, in HTTP tag, add error page routing and interceptor

<**http auto-config="true"**>

..  
**<intercept-url pattern="/loginFailed.html" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>**  
 <**form-login login-page="/login.html" authentication-failure-url="/loginFailed.html"/>**

..  
</**http**>

1. New method in Login Controller to control the error case

@RequestMapping(value=**"/loginFailed"**, method=RequestMethod.***GET***)  
**public** String loginFailed(ModelMap model) {  
 System.***out***.println(**"Login Failed"**);  
  
 model.addAttribute(**"error"**, **"true"**);  
 **return "login"**;  
}

This model.addAttribute(**"error"**, **"true"**); is passing the attribute with name error (value doesn’t matter) This attribute is checked in login.jsp page :

<**c:if test="${not empty error}"**>  
 <**div class="errorblock"**>  
 Your login was unsuccessful. <**br** />  
 Caused: **${**sessionScope[**"SPRING\_SECURITY\_LAST\_EXCEPTION"**].message **}** </**div**>  
</**c:if**>

1. Logout . Create Logout JSP, which is just some dummy jsp
2. Add logout method to Login Controller

@RequestMapping(value=**"/logout"**, method=RequestMethod.***GET***)  
**public** String logout(ModelMap model) {  
 **return "logout"**;  
}

1. Add logout and interceptor in security-config.xml in HTTP tag.

**<intercept-url pattern="/logout.html" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>  
 <logout logout-success-url="/logout.html"/>**

1. Add Logout button to index.jsp page, so we can Log out.

**<a class="btn btn-warning" href="j\_spring\_security\_logout"> Logout >> </a>**

**" href="j\_spring\_security\_logout"> this is doing all the magic, we call this URL to logout, Spring Security will call Session invalidate, eliminate any cookies we have,**

1. Create 403 JSP page. It’s just dummy page with text
2. Add handler to security-config.xml inside HTTP tag

<**access-denied-handler error-page="/403.html"**/>

1. Add method to Login Controller

@RequestMapping(value=**"/403"**, method=RequestMethod.***GET***)  
**public** String error403(ModelMap model) {  
 **return "403"**;  
}

1. Insert user with incorrect role

In users table insert user mark:

*mark $2a$10$m8le2GqwBcynPj8m03Rucucmpe3NWc1RnOZLAhkJzOBDpPsAc3AqK 1*

in authorities table insert user with some role:

*mark ROLE\_BAD*

1. Try to login with user mark, with correct password. He should have permission to access the index.jsp page.

# Expressions

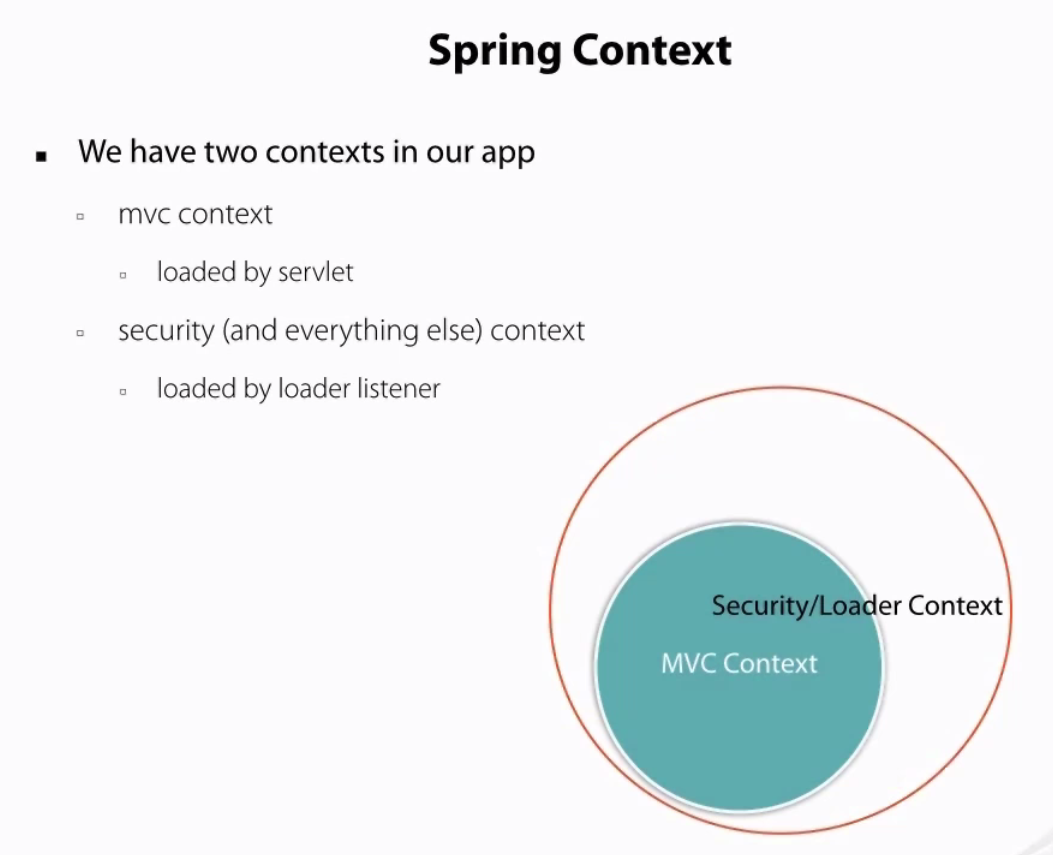
* Attribute of http element
* Simplify Boolean login into an expression
* Built-in expression:
  + hasRole
  + hasAnyRole
  + permitAll
  + hasPermission

Change security-config.xml

*<!--Expression config-->*<**http auto-config="true" use-expressions="true"** >  
 <**intercept-url pattern="/login.html" access="permitAll"**/>  
 <**intercept-url pattern="/loginFailed.html" access="permitAll"**/>  
 <**intercept-url pattern="/logout.html" access="permitAll"**/>  
 <**intercept-url pattern="/403.html" access="permitAll"**/>  
 <**intercept-url pattern="/\*\*" access="hasRole('ROLE\_USER')"** />  
 <**form-login login-page="/login.html" authentication-failure-url="/loginFailed.html"**/>  
 <**logout logout-success-url="/logout.html"**/>  
 <**access-denied-handler error-page="/403.html"**/>  
</**http**>

Method Level Security

* @PreAuthorize – most common, used to verify if someone has a role or permission before method can execute
* @PostAuthorize – after method is executed, if we want to filter result, based on a user rights
* <global-method-security/>
  + Pre-post-annotations
  + Secured-annotations
  + Jsr250-annotations
* Context matters



In our application we have 2 contexts:

MVC:

<**servlet**>  
 <**servlet-name**>fitTrackerServlet</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>/WEB-INF/config/servlet-config.xml</**param-value**>  
 </**init-param**>  
</**servlet**>

Security:

<**context-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>/WEB-INF/config/security-config.xml</**param-value**>  
</**context-param**>

## Roles

How to enabled them?

1. configuration

In **servlet-config.xml** add namespace for security

.

**xmlns:security=**<http://www.springframework.org/schema/security>

**.**

**.**

**.**

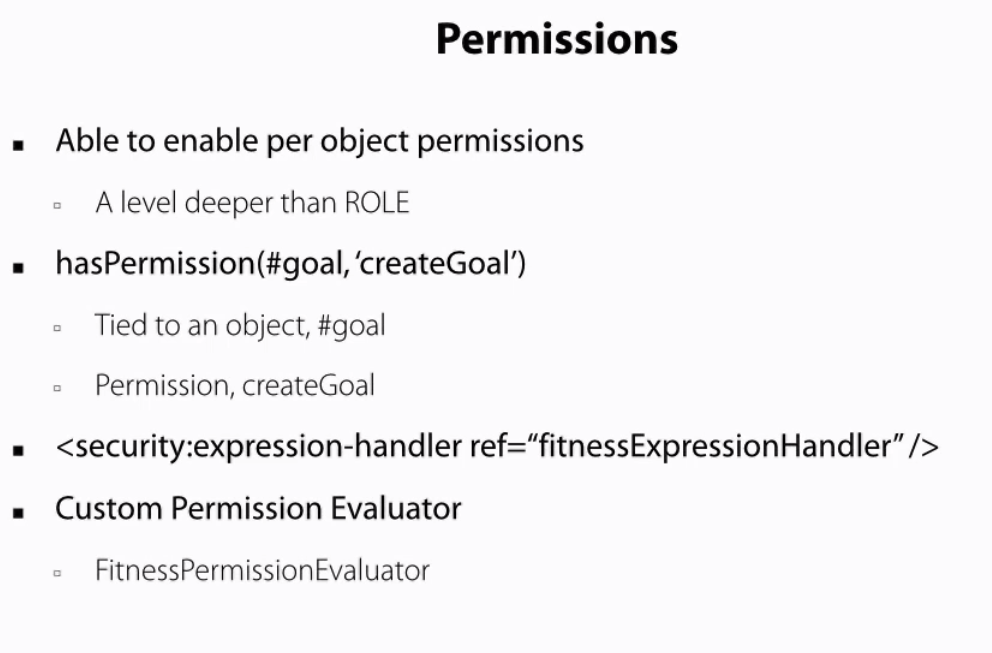
**http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security-3.2.xsd**

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

1. Annotate method you want to restrict. This will do the same as using **intercept-url pattern ,** but its more clean to use it on method, because same URL can have different role permission, for example allow GET, but not POST on the same URL.

@PreAuthorize(**"hasRole('ROLE\_ADMIN')"**)  
@RequestMapping(value = **"addGoal"**, method = RequestMethod.***POST***)  
**public** String updateGoal(@Valid @ModelAttribute(**"goal"**) Goal goal, BindingResult result) {

## Permissions



1. Annotate the method with permission expression

@PreAuthorize(**"hasRole('ROLE\_ADMIN') and hasPermission(#goal, 'createGoal')"**)  
@RequestMapping(value = **"addGoal"**, method = RequestMethod.***POST***)

1. Create permission table and insert values

**CREATE TABLE** permissions (  
 **username varchar**(50) **NOT NULL**,  
 **target varchar**(50) **NOT NULL**,  
 **permission varchar**(50) **NOT NULL**,  
 **CONSTRAINT** fk\_permissions\_users **FOREIGN KEY** (**username**) **REFERENCES** users (**username**));  
  
**create UNIQUE INDEX** ix\_perm\_username **on** permissions (**username**,**target**,**permission**)

**INSERT into** permissions (**username**, **target**, **permission**) **VALUES** ("cris","com.pluralsight.model.Goal","createGoal")

1. Create Expression Evaluator class. The method hasPermission control the access. If it return true, than the method is allowed to be called.

**public class FitnessPermissionEvaluator** **implements PermissionEvaluator** {  
  
 **private** DataSource **datasource**;  
  
 **public** DataSource getDatasource() {  
 **return datasource**;  
 }  
  
 **public void** setDatasource(DataSource datasource) {  
 **this**.**datasource** = datasource;  
 }  
  
 **public boolean** hasPermission(Authentication auth, Object targetDomainObject, Object permission) {  
  
 JdbcTemplate template = **new** JdbcTemplate(**datasource**);  
  
 **Object[] args = {((User) auth.getPrincipal()).getUsername(),  
 targetDomainObject.getClass().getName(),  
 permission.toString()};  
  
 int count = template.queryForObject("select *count*(*\**) from permissions p where " +  
 "p.username = ? and p.target = ? and p.permission = ?", args, Integer.class);** **if** (count == 1) {  
 **return true**;  
 } **else** {  
 **return false**;  
 }  
 }  
  
 **public boolean** hasPermission(Authentication arg0, Serializable id, String type, Object permission) {  
 **return false**;  
 }  
  
}

1. Adjust servlet-config.xml

<**security:global-method-security pre-post-annotations="enabled"**>  
 <**security:expression-handler ref="fitnessExpressionHandler"**/>  
</**security:global-method-security**>  
  
<**bean id="fitnessExpressionHandler"  
 class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler"**>  
 <**property name="permissionEvaluator"**>  
 <**bean id="permissionEvaluator" class="com.pluralsight.security.FitnessPermissionEvaluator"**>  
 <**property name="datasource" ref="dataSource"** />  
 </**bean**>  
 </**property**>  
</**bean**>

# **LDAP**

## ldap-authentication-provider

Spring security has built in ldap-authentication-provider

▪ Similar to jdbc-user-service

▪ Contains attributes for most common configurations:

□ group-search-filter

□ group-search-base

□ user-search-base

□ user-search-filter

▪ Can be combined with a user-details-service element

## ldap-server

▪ Helper element to connect to ldap server

▪ Can be used to create a test server

□ Loads an ldif file

□ Default port 389 can be overridden

1. Add Maven dependencies

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>**slf4j-simple**</artifactId>

<version>1.5.6</version>

</dependency>

<dependency>

<groupId>org.apache.directory.server</groupId>

<artifactId>**apacheds-all**</artifactId>

<version>1.5.5</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>**spring-security-ldap**</artifactId>

<version>3.2.0.RELEASE</version>

</dependency>

1. Add ldap provider in security-config.xml. Specify the ldap server. It scan your classpath for the ldif file, which will populate during the startup to LDAP. Comment out if you have other authentification-manager.

<**ldap-server ldif="classpath:users.ldif"**/>  
  
<**authentication-manager**>  
 <**ldap-authentication-provider  
 group-search-filter="member={0}"  
 group-search-base="ou=groups"  
 user-search-base="ou=people"  
 user-search-filter="uid={0}"** />  
</**authentication-manager**>

1. Create users.ldif file inse src/main/resources

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=rod,ou=people,dc=springframework,dc=org  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: Rod Johnson  
sn: Johnson  
uid: rod  
userPassword: koala  
  
dn: uid=dianne,ou=people,dc=springframework,dc=org  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: Dianne Emu  
sn: Emu  
uid: dianne  
userPassword: emu  
  
dn: uid=**scott**,ou=people,dc=springframework,dc=org  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: Scott  
sn: Wombat  
uid: **scott**  
userPassword: **wombat**  
  
dn: cn=user,ou=groups,dc=springframework,dc=org  
objectclass: top  
objectclass: groupOfNames  
cn: user  
member: uid=rod,ou=people,dc=springframework,dc=org  
member: uid=dianne,ou=people,dc=springframework,dc=org  
member: uid=scott,ou=people,dc=springframework,dc=org  
  
dn: cn=teller,ou=groups,dc=springframework,dc=org  
objectclass: top  
objectclass: groupOfNames  
cn: teller  
member: uid=rod,ou=people,dc=springframework,dc=org  
member: dianne=rod,ou=people,dc=springframework,dc=org  
  
dn: cn=supervisor,ou=groups,dc=springframework,dc=org  
objectclass: top  
objectclass: groupOfNames  
cn: supervisor  
member: uid=rod,ou=people,dc=springframework,dc=org

1. Start the application, check the logs, use to login with the user specified in ldif file, scott / wombat

# HTTPS

## Man In the Middle Attack

□ OWASP : if someone is monitoring the traffic in your environment using f.i. keylogger.

Saffer way is to submit request using HTTPS.

▪ Spring can force requests over HTTPS

▪ requires-channel="https"

## Certificate

▪ Requires a certificate

▪ For development a self signed cert is fine, but not for production

□ keytool

□ connector

□ requires-channel

1. Generate certificate inside Tomcat directory. Username and password is changeit

c:\Java\tomcat\bin>**keytool -genkey -alias tomcat -keyalg RSA -keystore C:\Java\tomcat\bin\tomcat**

1. Uncomment or change this in Tomcat’s server.xml

**<Connector**

**protocol="HTTP/1.1" port="8443" maxThreads="200" scheme="https" secure="true" SSLEnabled="true"**

**keystoreFile="C:\Java\tomcat\bin\tomcat" keystorePass="changeit" clientAuth="false" sslProtocol="TLS"/>**

1. Add the requires-channel in security-config.xml for the interceptors.

<**intercept-url pattern="/login.html" access="permitAll" requires-channel="https"**/>  
<**intercept-url pattern="/loginFailed.html" access="permitAll" requires-channel="https"**/>  
<**intercept-url pattern="/logout.html" access="permitAll" requires-channel="https"**/>  
<**intercept-url pattern="/403.html" access="permitAll" requires-channel="https"**/>

1. Access the page. https is crossed because it recognized the sef-signed certificate.

