
Part 1: Functional Specs and the Application Database

We'll develop a simple Bulletin application where various users can create, add, edit, and delete posts depending on their access levels.

In our system we have three roles:

- ROLE_ADMIN - provides administrative access
- ROLE_USER - provides regular access
- ROLE_VISITOR - provides visitor access

We also have three concrete users along with their roles:

- john - ROLE_ADMIN
- jane - ROLE_USER
- mike - ROLE_VISITOR

When john logs-in, he is given the ROLE_ADMIN. When jane logs-in, she is given the ROLE_USER. And when mike logs-in, he gets the ROLE_VISITOR.

Our Bulletin application has three types of posts:

- AdminPost - contains an id, date, and message
- PersonalPost - contains an id, date, and message
- PublicPost - contains an id, date, and message

Here are the **simple** rules:

1. Only users with ROLE_ADMIN can create AdminPost
2. Only users with ROLE_USER can create PersonalPost
3. Only users with ROLE_ADMIN or ROLE_USER can create PublicPost
4. Users with ROLE_VISITOR cannot create any post

Note: When we use the word 'create', we mean adding a new post.
Here are the **complex** rules:

1. A user can edit and delete posts that belongs only to them regardless of the role.
2. A user with ROLE_ADMIN or ROLE_USER can edit and delete PublicPosts.
3. We are required to show all posts in the main Bulletin page
a. ROLE_ADMIN can see all posts
b. ROLE_USER can see Personal and Public posts
c. ROLE_VISITOR can only see Public posts

Let's visualize the rules using tables:

An admin has READ and WRITE access to everything, but only READ access to the Personal Posts.

Admin

Post Type	View	Add	Edit	Delete
Admin	x	x	x	x
Personal	x			
Public	x	x	x	x

A regular user has READ and WRITE access to Personal Posts and Public Posts but only READ access to *Admin Posts*.

User

Post Type	View	Add	Edit	Delete
Admin				
Personal	x	x	x	x
Public	x	x	x	x

A visitor can only read Admin and Public Posts but no access of whatsoever in the Personal Posts section.

Visitor

Post Type	View	Add	Edit	Delete
Admin				
Personal				
Public	x			

The main problem:

If we focus on the simple rules, the solution looks easy. Just configure a simple http tag with a couple of intercept-url declarations. Here's how we may tackle this problem:

Admin Posts

```
<security:intercept-url pattern="/krams/admin/view"
access="hasRole('ROLE_ADMIN')"/>

<security:intercept-url pattern="/krams/admin/add"
access="hasRole('ROLE_ADMIN')"/>

<security:intercept-url pattern="/krams/admin/edit"
access="hasRole('ROLE_ADMIN')"/>

<security:intercept-url pattern="/krams/admin/delete"
access="hasRole('ROLE_ADMIN')"/>
```

Personal Posts

```
<security:intercept-url pattern="/krams/personal/view"
access="hasRole('ROLE_ADMIN') or hasRole('ROLE_USER')"/>

<security:intercept-url pattern="/krams/personal/add"
access="hasRole('ROLE_USER')"/>

<security:intercept-url pattern="/krams/personal/edit"
access="hasRole('ROLE_USER')"/>

<security:intercept-url pattern="/krams/personal/delete"
access="hasRole('ROLE_USER')"/>
```

Public Posts

```
<security:intercept-url pattern="/krams/public/view"
access="hasRole('ROLE_ADMIN') or hasRole('ROLE_USER') or
hasRole('ROLE_VISITOR')"/>

<security:intercept-url pattern="/krams/public/add"
access="hasRole('ROLE_ADMIN') or hasRole('ROLE_USER')"/>

<security:intercept-url pattern="/krams/public/edit"
access="hasRole('ROLE_ADMIN') or hasRole('ROLE_USER')"/>
```

```
<security:intercept-url pattern="/krams/public/delete"
access="hasRole('ROLE_ADMIN') or hasRole('ROLE_USER')"/>
```

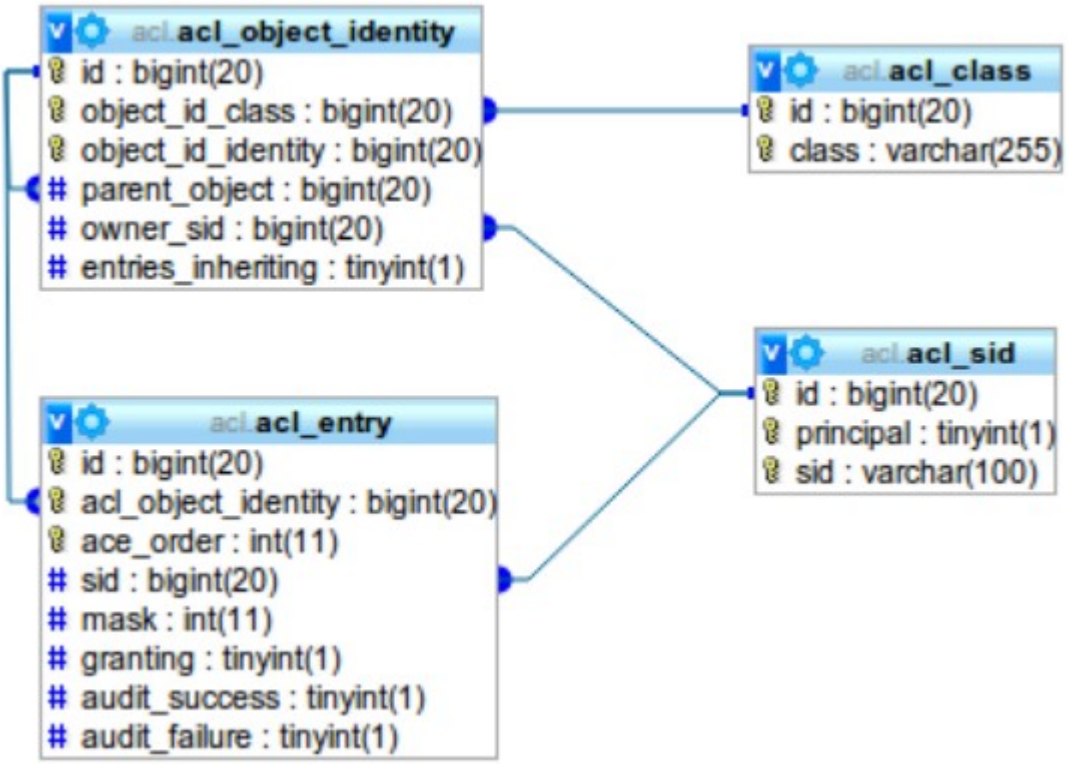
However, if we consider the complex rules, the intercept-url is unable to cope with the complex rules. Why? Because intercept-url is meant to secure at the URL-level. The complex rules are operating at the domain level.

The solution is to use ACL at the object level and intercept-url at the URL-level.

The ACL Database

We'll start our multi-part tutorial by creating a new MySQL database named acl. This database will contain our access control list. It's composed of four tables:

- acl_class
- acl_sid
- acl_object_identity
- acl_entry



Create database and populate data

acl_class

The table *acl_class* stores the fully qualified name of domain objects. It is made up of the package name and class name of the object.

In the table below we have declared three fully qualified names that pertain to our three domain objects:

			id	class
☐	✎	✕	1	org.krams.tutorial.domain.AdminPost
☐	✎	✕	2	org.krams.tutorial.domain.PersonalPost
☐	✎	✕	3	org.krams.tutorial.domain.PublicPost

Field	Description
id	The primary key
class	The fully qualified name of the domain object

acl_sid

The table *acl_sid* stores the name of the users which can be a principal (like usernames john, james, mark) or an authority (like roles ROLE_ADMIN, ROLE USER, ROLE_ANYONE).

In the table below we have declared three sid objects:

			id	principal	sid
☐	✎	✕	2	1	jane
☐	✎	✕	1	1	john
☐	✎	✕	3	1	mike

Field	Description
id	The primary key
principal	A flag to indicate if the *sid* field is a username or a role
sid	The actual username (ie. john) or role (ie. ROLE_ADMIN)

acl_object_identity

The table *acl_object_identity* stores the actual identities of the domain objects. The identities are referenced via a unique id which is retrieved from another database: the application

database.

id	object_id_class	object_id_identity	parent_object	owner_sid	entries_inheriting
1	1	1	NULL	1	0
2	1	2	NULL	1	0
3	1	3	NULL	1	0
4	2	1	NULL	1	0
5	2	2	NULL	1	0
6	2	3	NULL	1	0
7	3	1	NULL	1	0
8	3	2	NULL	1	0
9	3	3	NULL	1	0

Field	Description
id	The primary key
object_id_class	Refers to the id field in the *acl_class*. This is a reference to the fully qualified name of the class
object_id_identity	Refers to the primary id of the domain object. The id is assigned from another database: the application database. Every domain object in the application needs to have a unique id.
parent_object	Refers to the id of the parent object if existing
owner_sid	Refers to the id field in the *acl_sid*. This is a reference to the username or role
entries_inheriting	A flag to indicate whether the object has inherited entries

acl_entry

The table *acl_entry* stores the actual permissions assigned for each user and domain object.

id	acl_object_identity	ace_order	sid	mask	granting	audit_success	audit_failure
1	1	1	1	1	1	1	1
2	2	1	1	1	1	1	1
3	3	1	1	1	1	1	1
4	1	2	1	2	1	1	1
5	2	2	1	2	1	1	1
6	3	2	1	2	1	1	1

Field	Description
id	The primary key
acl_object_identity	Refers to the *id* field in the *acl_object_identity* table
ace_order	Refers to the ordering of the access control entries
sid	Refers to the *id* field in the *acl_sid* table

Field	Description
mask	A bitwise mask to indicate the permissions. A value of 1 is equivalent to READ permission, 2 for WRITE, and so forth.
granting	A flag to indicate whether the mask should be interpreted as granting access or deny access
audit_success	A flag to indicate whether to audit a successful permission
audit_failure	A flag to indicate whether to audit a failed permission

Part 2: Spring Security Configuration

We'll be declaring two configuration files:

1. spring-security.xml
2. acl-context.xml

spring-security.xml

This contains standard Spring Security configuration. It declares the following:
1. A set of intercept-url patterns.
2. An authentication manager
3. An Md5 password encoder
4. An in-memory user service

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:security="http://www.springframework.org/schema/security"
 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-3.0.xsd
">

 <!-- Loads ACL related configurations -->
 <import resource="acl-context.xml" />

 <!-- This is where we configure Spring-Security  -->
 <security:http auto-config="true" use-expressions="true"
access-denied-page="/krams/auth/denied" >
```

```
  <security:intercept-url pattern="/krams/auth/login"
access="permitAll"/>
  <security:intercept-url pattern="/krams/bulletin/view"
access="hasRole('ROLE_VISITOR')"/>
  <security:intercept-url pattern="/krams/role/admin"
access="hasRole('ROLE_ADMIN')"/>
  <security:intercept-url pattern="/krams/role/user"
access="hasRole('ROLE_USER')"/>
  <security:intercept-url pattern="/krams/role/visitor"
access="hasRole('ROLE_VISITOR')"/>

  <security:form-login
    login-page="/krams/auth/login"
    authentication-failure-url="/krams/auth/login?error=true"
    default-target-url="/krams/bulletin/view"/>

  <security:logout
    invalidate-session="true"
    logout-success-url="/krams/auth/login"
    logout-url="/krams/auth/logout"/>

 </security:http>

 <!-- Declare an authentication-manager to use a custom userDetailsService
-->
 <security:authentication-manager>
         <security:authentication-provider
user-service-ref="userDetailsService">
           <security:password-encoder ref="passwordEncoder"/>
         </security:authentication-provider>
 </security:authentication-manager>

 <!-- Use a Md5 encoder since the user's passwords are stored as Md5 in
the database -->
 <bean
class="org.springframework.security.authentication.encoding.Md5Passwor
dEncoder" id="passwordEncoder"/>

  <!-- An in-memory list of users. No need to access an external database
layer.
      See Spring Security 3.1 Reference 5.2.1 In-Memory Authentication -->
  <!-- john's password: admin
    jane's password: user
    mike's password: visitor  -->
  <security:user-service id="userDetailsService">
```

```
     <security:user name="john"
password="21232f297a57a5a743894a0e4a801fc3" authorities="ROLE_ADMIN,
ROLE_USER, ROLE_VISITOR" />
     <security:user name="jane"
password="ee11cbb19052e40b07aac0ca060c23ee" authorities="ROLE_USER,
ROLE_VISITOR" />
     <security:user name="mike"
password="127870930d65c57ee65fcc47f2170d38" authorities="ROLE_VISITOR"
/>
   </security:user-service>

</beans>
```

acl-context.xml

This contains ACL-related configuration. It declares the following:
1. A *global-method-security* tag which enables method security expressions
2. An expression handler
3. A permission evaluator
4. An ACL service
5. A lookup strategy
6. A datasource
7. An ACL cache
8. An ACL authorization strategy
9. A role hierarchy

The Method Security:

```
<security:global-method-security pre-post-annotations="enabled">
  <security:expression-handler ref="expressionHandler" />
</security:global-method-security>
```

There are three types of method security annotations available in Spring Security
1. @Secured annotation
2. JSR-250 annotation
3. Expression-based access control

The Expression Handler

The expression-handler property defines a custom expression handler instance. Without this property Spring Security will declare a default expression handler with no ACL support. We need to declare a custom handler because we need ACL support. It turns out Spring provides a

default implementation that we can customize so that we don't have to create one from scratch.

```
<bean id="expressionHandler"
class="org.springframework.security.access.expression.method.DefaultMe
thodSecurityExpressionHandler">

  <property name="permissionEvaluator" ref="permissionEvaluator" />
  <property name = "roleHierarchy" ref="roleHierarchy"/>

 </bean>
```

Here we declared a reference to a customized expression handler: DefaultMethodSecurityExpressionHandler. This is actually the default expression handler but it needs to be declared manually so that we can provide a customized permission evaluator.

The permissionEvaluator property defines a reference to a custom permission evaluator, while the roleHierarchy allows us to define the hierarchy of our roles.

The Role Hierarchy

Role hierarchy is a way of declaring which role is the boss of other roles. In our sample configuration ROLE_ADMIN > ROLE USER, means whenever a user has a ROLE_ADMIN, he also gets the ROLE_USER. And because we declared ROLE_USER > ROLE_VISITOR, he also gets the ROLE_VISITOR.

```
<bean id="roleHierarchy"
class="org.springframework.security.access.hierarchicalroles.RoleHiera
rchyImpl">

     <property name="hierarchy">
         <value>
             ROLE_ADMIN > ROLE_USER
             ROLE_USER > ROLE_VISITOR
         </value>
     </property>

 </bean>
</beans>
```

The ACL Permission Evaluator

AclPermissionEvaluator is the default implementation for evaluating ACLs with expression-based access control but it's not enabled by default. It needs to be declared manually and it needs to be customized.

The <constructor-arg ref="aclService"/> is a reference to a custom ACL service. Basically this is the service that will access the ACL database.

```
<bean class="org.springframework.security.acls.AclPermissionEvaluator"
id="permissionEvaluator">

  <constructor-arg ref="aclService"/>

 </bean>
```

The ACL Service

The JdbcMutableAclService is a JDBC-based ACL service. It uses JdbcTemplate to simplify JDBC access.

The <constructor-arg ref="dataSource"/> is a reference to a datasource, in our case, a MySQL datasource.

The <constructor-arg ref="lookupStrategy"/> is a reference to a lookup strategy. Its purpose is to provide an optimized lookup when querying the database.

The <constructor-arg ref="aclCache"/> is a reference to an ACL cache. Its purpose is to lessen database access by checking first if the ACL entry is already available in the cache.

```
<bean
class="org.springframework.security.acls.jdbc.JdbcMutableAclService"
id="aclService">

      <constructor-arg ref="dataSource"/>

        <constructor-arg ref="lookupStrategy"/>

        <constructor-arg ref="aclCache"/>

 </bean>
```

The Datasource

This is a standard MySQL datasource that uses a C3P0 connection pool. The jdbcUrl property points to the acl database.

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
   destroy-method="close"
   p:driverClass="com.mysql.jdbc.Driver"
   p:jdbcUrl="jdbc:mysql://localhost/acl"
   p:user="root"
   p:password=""
   p:acquireIncrement="5"
   p:idleConnectionTestPeriod="60"
   p:maxPoolSize="100"
   p:maxStatements="50"
   p:minPoolSize="10" />
```

The Lookup Strategy

```
<bean id="lookupStrategy"
class="org.springframework.security.acls.jdbc.BasicLookupStrategy">

        <constructor-arg ref="dataSource"/>
        <constructor-arg ref="aclCache"/>
        <constructor-arg ref="aclAuthorizationStrategy"/>
        <constructor-arg ref="auditLogger"/>

    </bean>
```

Here we declare Spring Security's default implementation of a lookup strategy BasicLookupStrategy. As mentioned earlier, the purpose of a lookup strategy is to provide an optimized lookup when querying the database.

The <constructor-arg ref="dataSource"/> is a reference to the same MySQL datasource we described earlier.

The <constructor-arg ref="aclCache"/> is a reference to the same ACL cache we described earlier.

The <constructor-arg ref="aclAuthorizationStrategy"/> is a reference to an AclAuthorizationStrategy implementation.

What is AclAuthorizationStrategy?

Strategy used by AclImpl to determine whether a principal is permitted to call adminstrative methods on the AclImpl.

What is AuditLogger?

Used by *AclImpl* to log audit events.

The <constructor-arg ref="auditLogger"/> is a reference to an *AuditLogger* implementation.

```
<bean id="auditLogger"
class="org.springframework.security.acls.domain.ConsoleAuditLogger"/>
```

The ACL Cache

Here we declare Spring Security's default implementation of *AclCache* interface. It's purpose is to lessen database lookups

```
<bean id="aclCache"
class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
<constructor-arg>

  <bean class="org.springframework.cache.ehcache.EhCacheFactoryBean">

<property name="cacheManager">

<bean
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"/>

 </property>

 <property name="cacheName" value="aclCache"/>
  </bean>
 </constructor-arg>
</bean>
```

The ACL Authorization Strategy

```
<bean id="aclAuthorizationStrategy"
class="org.springframework.security.acls.domain.AclAuthorizationStrate
gyImpl">
   <constructor-arg>
```

```
            <list>
                <bean
class="org.springframework.security.core.authority.GrantedAuthorityImp
l">
                    <constructor-arg value="ROLE_ADMIN"/>
                </bean>
                <bean
class="org.springframework.security.core.authority.GrantedAuthorityImp
l">
                    <constructor-arg value="ROLE_ADMIN"/>
                </bean>
                <bean
class="org.springframework.security.core.authority.GrantedAuthorityImp
l">
                    <constructor-arg value="ROLE_ADMIN"/>
                </bean>
            </list>
        </constructor-arg>
    </bean>
```

AclAuthorizationStrategyImpl is the default implementation of AclAuthorizationStrategy. Notice the constructor accepts three arguments. Based on the Spring Security API, constructor signature is as follows:

public AclAuthorizationStrategyImpl(GrantedAuthority[] auths).

Part 3: Spring MVC Module

MainController

```
package org.krams.tutorial.controller;

import org.apache.log4j.Logger;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * Handles and retrieves the common or admin page depending on the URI
template.
 * A user must be log-in first he can access these pages.  Only the admin
can see
 * the adminpage, however.
```

```
 */
@Controller
@RequestMapping("/main")
public class MainController {

 protected static Logger logger = Logger.getLogger("controller");

 /**
  * Handles and retrieves the common JSP page that everyone can see
  *
  * @return the name of the JSP page
  */
    @RequestMapping(value = "/common", method = RequestMethod.GET)
    public String getCommonPage() {
     logger.debug("Received request to show common page");

     // Do your work here. Whatever you like
     // i.e call a custom service to do your business
     // Prepare a model to be used by the JSP page

     // This will resolve to /WEB-INF/jsp/commonpage.jsp
     return "commonpage";
 }

    /**
     * Handles and retrieves the admin JSP page that only admins can see
     *
     * @return the name of the JSP page
     */
    @RequestMapping(value = "/admin", method = RequestMethod.GET)
    public String getAdminPage() {
     logger.debug("Received request to show admin page");

     // Do your work here. Whatever you like
     // i.e call a custom service to do your business
     // Prepare a model to be used by the JSP page

     // This will resolve to /WEB-INF/jsp/adminpage.jsp
     return "adminpage";
 }
}
```

This controller declares two mappings:

```
/main/common - any registered user can access this page
/main/admin - only admins can access this page
```

Each mapping will resolve to a specific JSP page. The common JSP page is accessible by everyone, while the admin page is accessible only by admins. Right now, everyone has access to these pages because we haven't enabled Spring Security ye

We've finished setting-up the primary controller and the associated JSP views. Now, we add the required XML configurations to enable Spring MVC **and** Spring Security at the same time.

We start by adding the web.xml:

```
<filter>
        <filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</fi
lter-class>
</filter>

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

<context-param>
 <param-name>contextConfigLocation</param-name><param-value>
 /WEB-INF/spring-security.xml
 /WEB-INF/applicationContext.xml
 </param-value></context-param>

<servlet>
 <servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</serv
let-class>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>spring</servlet-name>
 <url-pattern>/krams/*</url-pattern>
</servlet-mapping>

<listener>
```

```
<listener-class>org.springframework.web.context.ContextLoaderListener<
/listener-class>
</listener>
```

Notice the url-patterns for the DelegatingFilterProxy and DispatcherServlet. Spring Security is placed at the root-path
`/*`
Whereas, Spring MVC is placed at a sub-path
`/krams/*`

We also referenced two important XML configuration files:

```
spring-security.xml
applicationContext.xml
```

spring-security.xml contains configuration related to Spring Security.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:security="http://www.springframework.org/schema/security"
 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-3.0.xsd
">

 <!-- This is where we configure Spring-Security  -->
 <security:http auto-config="true" use-expressions="true"
access-denied-page="/krams/auth/denied" >

  <security:intercept-url pattern="/krams/auth/login"
access="permitAll"/>
  <security:intercept-url pattern="/krams/main/admin"
access="hasRole('ROLE_ADMIN')"/>
  <security:intercept-url pattern="/krams/main/common"
access="hasRole('ROLE_USER')"/>

  <security:form-login
    login-page="/krams/auth/login"
    authentication-failure-url="/krams/auth/login?error=true"
    default-target-url="/krams/main/common"/>
```

```
  <security:logout
    invalidate-session="true"
    logout-success-url="/krams/auth/login"
    logout-url="/krams/auth/logout"/>

 </security:http>

 <!-- Declare an authentication-manager to use a custom userDetailsService
-->
 <security:authentication-manager>
         <security:authentication-provider
user-service-ref="userDetailsService">
           <security:password-encoder ref="passwordEncoder"/>
         </security:authentication-provider>
 </security:authentication-manager>

 <!-- Use a Md5 encoder since the user's passwords are stored as Md5 in
the database -->
 <bean
class="org.springframework.security.authentication.encoding.Md5Passwor
dEncoder" id="passwordEncoder"/>

  <!-- An in-memory list of users. No need to access an external database
layer.
      See Spring Security 3.1 Reference 5.2.1 In-Memory Authentication -->
  <!-- john's password is admin, while jane;s password is user  -->
  <security:user-service id="userDetailsService">
     <security:user name="john"
password="21232f297a57a5a743894a0e4a801fc3" authorities="ROLE_USER,
ROLE_ADMIN" />
     <security:user name="jane"
password="ee11cbb19052e40b07aac0ca060c23ee" authorities="ROLE_USER" />
   </security:user-service>

</beans>
```

Notice that the bulk of the security configuration is inside the **http** element. Here's what we observe:

1. We declared the denied page URL in the `access-denied-page="/krams/auth/denied"`

2. We provided three URLs with varying permissions. We use Spring Expression Language (SpEL) to specify the role access. For admin only access we specified **hasRole('ROLE_ADMIN'**) and for regular users we

use **hasRole('ROLE_USER')**. To enable SpEL, you need to set **use-expressions** to true

3. We declared the login URL
`login-page="/krams/auth/login"`

4. We declared the login failure URL
`authentication-failure-url="/krams/auth/login?error=true"`

5. We declared the URL where the user will be redirected if he logs out
`logout-success-url="/krams/auth/login"`

6. We declared the logout URL
`logout-url="/krams/auth/logout`