**Simplified Spring Security with Grails**

Engineering

Peter Ledbrook

August 11, 2010

[8 Comments](http://spring.io/blog/2010/08/11/simplified-spring-security-with-grails#disqus_thread)

Spring Security is a powerful library for securing your applications that comes with a bewildering number of options. Based on Spring, it can be readily integrated into a Grails application. But why not save the hassle and use the [new improved Grails plugin](http://grails.org/plugin/spring-security-core)?

The plugin has gone through several evolutionary stages that started with the Acegi plugin. Its most recent incarnation is a complete rewrite for Spring Security 3 and Spring 3. One of the results of this is that the plugin will only work with Grails 1.2.2 and above. Another significant change is that there is no longer just one Spring Security plugin: some features have been broken out into optional plugins. So now you only include the features you need in your application.

So what do the plugins give you? The core provides the basics necessary for access control in an easy-to-use package based on users and roles. In fact, many applications won’t need any other plugin than the core one. For those who do need something extra, here is a list of the other plugins in the family:

* [OpenID](http://grails.org/plugin/spring-security-openid) - authentication using [OpenID](http://openid.net/)
* [LDAP](http://grails.org/plugin/spring-security-ldap) - authentication against LDAP servers
* [CAS](http://grails.org/plugin/spring-security-cas) - single sign-on using [CAS](http://www.jasig.org/cas)
* [ACLs](http://grails.org/plugin/spring-security-acl) - access control via Spring Security's ACLs
* [UI](http://grails.org/plugin/spring-security-ui) - user interface for user and role managment, plus other features

In this article I’ll show you how to secure a Grails application from scratch using the new core plugin.

**Update** This article now has two companion screencasts:

|  |  |
| --- | --- |
| [caption id="attachment\_5509" align="center" width="250" caption="Spring Security Plugin Introduction"][http://blog.springsource.com/wp-content/uploads/2010/08/front-slides-for-spring-security-cast-1-thumbnail.png](http://www.youtube.com/watch?v=auwML_bsUEE)[/caption] | [caption id="attachment\_5510" align="center" width="250" caption="Spring Security Plugin - AJAX"][http://blog.springsource.com/wp-content/uploads/2010/08/front-slides-for-spring-security-cast-2-thumbnail.png](http://www.youtube.com/watch?v=H_mGgZddXx8)[/caption] |

**Setup**

As with most plugins, your first step will be to install the Spring Security plugin. Of course, you’ll need a project to install it into and for this article I [have provided](http://people.apache.org/%7Epledbrook/hubbub.zip) a simple Twitter-clone called Hubbub (based on the sample application from Grails in Action). You can also get the finished project from [here](http://people.apache.org/%7Epledbrook/hubbub-finished.zip).

So, from within your project, run:

grails install-plugin spring-security-core

If you look at the output generated by the plugin installation, you will see that it provides a couple of commands. The most important of these is s2-quickstart, which will help you get up and running with the minimum of fuss. It generates both the basic domain classes you need to store user information and the controllers that handle authentication.

Before you run the command, you may need to make a decision. If you already have a ‘user’ domain class, you will have to decide how to integrate it with the one generated by the plugin. One option is to replace the existing domain class and simply apply your customisations to the replacement. The other approach consists of making your own domain class extend the plugin’s one.

Which is better? I prefer the latter because it allows you to easily update the generated user domain class if its template ever changes. It also means you don’t overly pollute your domain model with Spring Security specifics. On the downside, you have to deal with domain class inheritance, although the cost is pretty minimal.

For Hubbub, we’ll make the user domain class extend the generated one, which means we should use domain class names that don’t conflict with the existing ones:

grails s2-quickstart org.example SecUser SecRole

This will create three domain classes for us:

* org.example.SecUser
* org.example.SecRole
* org.example.SecUserSecRole - links users to roles

and two controllers:

* LoginController
* LogoutController

plus their associated views. In just two commands, we have everything we need to start securing our application!

The sample application does need one more change: its URL mappings mean that the login and logout controllers can’t be reached. That’s simple enough to fix by adding the following two lines to UrlMappings.groovy:

"/login/$action?"(controller: "login")

"/logout/$action?"(controller: "logout")

If you don’t make the change, the login page will generate a 404 error! Now let’s get down to the business of protecting the application.

**Adding access control**

The whole point of this exercise is to limit access to certain parts of the application. For web applications this most commonly means protecting particular pages, or more specifically URLs. In the case of Hubbub, we have the following requirements:

* The home page is accessible to everyone - /
* Only known users can see the posts for a particular user - /person/<username>
* Only users with a 'user' role should be able to access their own timeline - /timeline
* Same goes for following another user - /post/followAjax
* Only fully authenticated users with the 'user' role should be able to post a new message - /post/addPostAjax

With the Spring Security plugin this is trivial to achieve, although you do have to make a decision on which of three mechanisms to use. You can take a controller-centric approach and annotate the actions; work with static URL rules in Config.groovy; or define runtime rules in the database using request maps.

**Annotations**

For a controller-centric approach, you can’t beat the @Secured annotation provided by the plugin. In it’s simplest incarnation, you pass it a list of basic rules that define who can access the corresponding action. Here, I apply Hubbub’s access control rules via annotations on the post controller:

package org.example

import grails.plugins.springsecurity.Secured

class PostController {

...

@Secured(['ROLE\_USER'])

def followAjax = { ... }

@Secured(['ROLE\_USER', 'IS\_AUTHENTICATED\_FULLY'])

def addPostAjax = { ... }

def global = { ... }

@Secured(['ROLE\_USER'])

def timeline = { ... }

@Secured(['IS\_AUTHENTICATED\_REMEMBERED'])

def personal = { ... }

}

The IS\_AUTHENTICATED\_\* rules are built into Spring Security, but ROLE\_USER is a role that must exist in the database - something we have yet to do. Also, if you specify more than one rule in the list, then the current user normally only has to satisfy one of them - as is explained in the user guide. IS\_AUTHENTICATED\_FULLY is a special case: if specified, it must be satisfied *in addition* to the other rules in the list.

The built-in rules are as follows:

* IS\_AUTHENTICATED\_ANONYMOUSLY - anyone has access; no need for the user to log in
* IS\_AUTHENTICATED\_REMEMBERED - only known users that have logged in or are remembered from a previous session are allowed access
* IS\_AUTHENTICATED\_FULLY - users must log in to gain access, even if they checked “remember me” last time

The first two of these distinguish between known and unknown users, where known users are ones that have an entry in the ‘user’ database table. The last is typically applied in cases where the user is accessing particularly sensitive information, such as bank account or credit card data. After all, someone else could be accessing your application using the “remember me” cookie from the previous user.

You can also apply the annotation to the controller class itself, which results in all actions inheriting the rules defined by it. If an action has its own annotation, that overrides the class-level one. The annotation isn’t just limited to a list of rules like this either: take a look at the [user guide](http://burtbeckwith.github.com/grails-spring-security-core/docs/manual/guide/5%20Configuring%20Request%20Mappings%20to%20Secure%20URLs.html#5.4%20Using%20Expressions%20to%20Create%20Descriptive,%20Fine-Grained%20Rules) to see how to use expressions to provide greater control over the rules.

**Static URL rules**

If annotations aren’t your thing, you can define access control rules via a static map in Config.groovy. If you like to keep your rules in one place, it’s ideal. Here is how you would define Hubbub’s rules using this mechanism:

import grails.plugins.springsecurity.SecurityConfigType

...

grails.plugins.springsecurity.securityConfigType = SecurityConfigType.InterceptUrlMap

grails.plugins.springsecurity.interceptUrlMap = [

'/timeline': ['ROLE\_USER'],

'/person/\*': ['IS\_AUTHENTICATED\_REMEMBERED'],

'/post/followAjax': ['ROLE\_USER'],

'/post/addPostAjax': ['ROLE\_USER', 'IS\_AUTHENTICATED\_FULLY'],

'/\*\*': ['IS\_AUTHENTICATED\_ANONYMOUSLY']

]

Notice how the most general rule comes last? That’s because order is important: Spring Security iterates through the rules and applies the *first one that matches the current URL*. So if the ‘/\*\*’ rule came first, your application would effectively be unprotected since all URLs would be matched to it. Also notice that you have to explicitly tell the plugin to use the map via the grails.plugins.springsecurity.securityConfigType settings.

**Dynamic request maps**

Do you want to update URL rules at runtime without restarting the application? If that’s the case, you’ll probably want to use request maps, which are basically URL rules stored in the database. To enable this mechanism, add the following to Config.groovy:

import grails.plugins.springsecurity.SecurityConfigType

...

grails.plugins.springsecurity.securityConfigType = SecurityConfigType.Requestmap

All you then have to do is create instances of the Requestmap domain class, for example in BootStrap.groovy:

new Requestmap(url: '/timeline', configAttribute: 'ROLE\_USER').save()

new Requestmap(url: '/person/\*', configAttribute: 'IS\_AUTHENTICATED\_REMEMBERED').save()

new Requestmap(url: '/post/followAjax', configAttribute: 'ROLE\_USER').save()

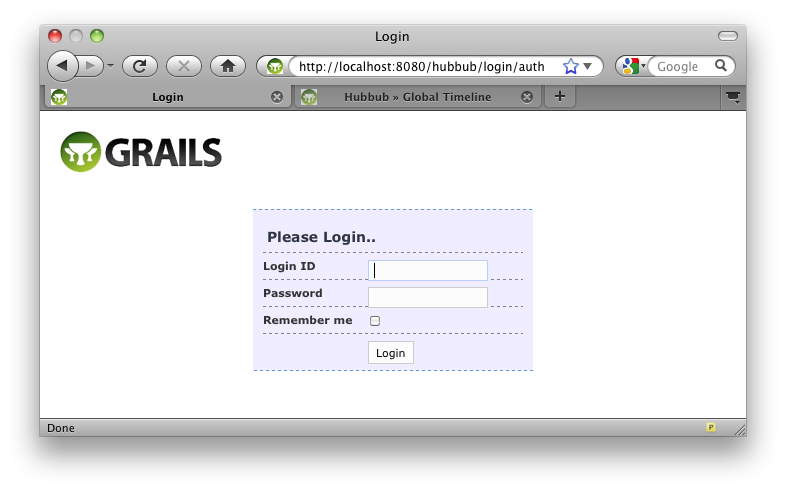
new Requestmap(url: '/post/addPostAjax', configAttribute: 'ROLE\_USER,IS\_AUTHENTICATED\_FULLY').save()

new Requestmap(url: '/\*\*', configAttribute: 'IS\_AUTHENTICATED\_ANONYMOUSLY').save()

Of course, there is a performance cost to this approach since it involves the database, but it is minimised through the use of caching. Take a look at the [user guide](http://burtbeckwith.github.com/grails-spring-security-core/docs/manual/guide/5%20Configuring%20Request%20Mappings%20to%20Secure%20URLs.html#5.3%20Requestmap%20Instances%20Stored%20in%20the%20Database) for more information on this. Also, you don’t have to worry about the order of the rules in this case because the plugin picks the most specific URL pattern that matches the current URL.

Which of these approaches should you use? It depends on how your application is set up and how you think about access control. Annotations make sense where rules apply on a per-controller basis and controllers have distinct URLs. If you tend to group controllers under a single URL, like /admin/ or you simply like to keep all your rules in one place, then you’re probably better off with the static rules defined in Config.groovy. The third mechanism, request maps, only make sense if you want to add, change, or remove rules at runtime. A classic example where you might want to do this is in a CMS application, where URLs themselves are defined dynamically.

Whichever approach you take, once the rules are implemented your application *is* protected. For example, if you try to access the /timeline page in Hubbub at this point, you will be redirected to the standard login page:

[](http://blog.springsource.com/wp-content/uploads/2010/08/spring-security-login-page.png)

Great! But who are you going to log in as? How are users going to log out? Protecting your pages is only the first step. You also need to make sure that you have the relevant security data (users and roles) and a user interface that’s security aware.

**Next steps**

With the access control in place, you need to look at the user experience. Do you really want users clicking on links that they don’t have access to? What about those roles you are using in the access control? When do they get created? Let’s answer those questions now.

**Security data**

Some applications only care whether a user is known or not and in such cases you don’t need to worry about roles because the IS\_AUTHENTICATED\_\* rules are sufficient. But if your application needs more control over who has access to what, you will need roles. These are typically defined early in the life of the application and correspond to unchanging reference data. That makes BootStrap the ideal place to create them. For Hubbub, we add ‘user’ and ‘admin’ roles like so:

import org.example.SecRole

class BootStrap {

def init = {

...

def userRole = SecRole.findByAuthority('ROLE\_USER') ?: new SecRole(authority: 'ROLE\_USER').save(failOnError: true)

def adminRole = SecRole.findByAuthority('ROLE\_ADMIN') ?: new SecRole(authority: 'ROLE\_ADMIN').save(failOnError: true)

...

}

}

Of course, if the data already exists we don’t want to recreate it, hence why we use findByAuthority().

Adding users is almost as straightforward, but there are a couple of requirements that you need to bear in mind. First, the generated ‘user’ domain class has an enabled property that is false by default. If you don’t explicitly initialise it to true the corresponding user won’t be able to log in. Second, passwords are rarely stored in the database as plain text, so you will need to encode them first using an appropriate digest algorithm.

Fortunately, the plugin provides a useful service to help here: SpringSecurityService. Let’s say we want to create an ‘admin’ user in Hubbub’s BootStrap. The code would look something like this:

import org.example.\*

class BootStrap {

def springSecurityService

def init = {

...

def adminUser = SecUser.findByUsername('admin') ?: new SecUser(

username: 'admin',

password: springSecurityService.encodePassword('admin'),

enabled: true).save(failOnError: true)

if (!adminUser.authorities.contains(adminRole)) {

SecUserSecRole.create adminUser, adminRole

}

...

}

}

We simply inject the security service into BootStrap and then use its encodePassword() method to convert the plain text password to its hash. This approach works particularly well when you decide to change the digest algorithm you use, because the service will encode passwords using the same algorithm as the one used when comparing them for authentication. In other words, the above code stays the same no matter what algorithm is used.

**Update** As of version 1.2 of the Spring Security Core plugin, the generated User class automatically encodes the password when an instance is saved. Hence you no longer need to explicitly use SpringSecurityService.encodePassword()

Once the user is created, we check whether it has the ‘admin’ role and if it doesn’t, we assign the role to the user. We do this by way of the generated SecUserSecRole class and its create() method.

With the security data in place, and the knowledge of how to create it on demand where necessary, it’s time to make the user interface aware of authentication, users, and roles.

**The user interface**

There are two aspects of the UI I want to look at here: displaying information specific to the user and making sure that the user can only see what he’s allowed to. The first of these boils down to one question: how do we get the ‘user’ domain instance for the currently logged in user? Consider Hubbub’s timeline page, which displays all the posts of the people that the current user is following:

class PostController {

def springSecurityService

...

@Secured(['ROLE\_USER'])

def timeline = {

def user = SecUser.get(springSecurityService.principal.id)

def posts = []

if (user.following) {

posts = Post.withCriteria {

'in'("user", user.following)

order("createdOn", "desc")

}

}

[ posts: posts, postCount: posts.size() ]

}

...

}

As you can see, all we need to do is inject the security service again and use it to get hold of the principal. Unless you have created a custom version of the UserDetailsService (don’t worry if you haven’t come across this before), the principal will be an instance of org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser whose id property contains the ID of the corresponding ‘user’ domain instance.

One thing you need to be aware of: if the current user is authenticated anonymously, i.e. he hasn’t logged in and isn’t remembered, the principal property will return a string instead. So if an action can be accessed by an unauthenticated user, make sure you check the type of the principal before using it!

What about ensuring users can only see what they are supposed to? For that, the plugin provides a rich set of GSP tags in the sec namespace. Let’s say we want to add a couple of navigation links to Hubbub, but we only want to display one of them when the user isn’t logged in and the other only if the user has the ROLE\_USER role:

<sec:ifNotLoggedIn>

<g:link controller="login" action="auth">Login</g:link>

</sec:ifNotLoggedIn>

<sec:ifAllGranted roles="ROLE\_USER">

<g:link class="create" controller="post" action="timeline">My Timeline</g:link>

</sec:ifAllGranted>

The markup inside the <sec:if\*> tags will only be rendered to the page if the condition is satisfied. The plugin provides several other similar tags that all behave in a consistent fashion. See the [user guide](http://burtbeckwith.github.com/grails-spring-security-core/docs/manual/guide/6%20Helper%20Classes.html#6.1%20SecurityTagLib) for more information.

The above example also shows you how to create a link to the login page. Allowing the user to log out is similarly straightforward. Hubbub provides a side panel that displays amongst other things the name of the logged in user and a link to sign out:

<sec:username /> (<g:link controller="logout">sign out</g:link>)

Easy! The combination of these tags and the security service should be more than sufficient to integrate your user interface with Spring Security. Just remember to keep your user interface elements in sync with your access control rules: you don’t want bits of UI visible that result in an “unauthorised user” error.

I’ve now covered all the basic elements of the Spring Security plugin, but there are still two features that will affect a large number of users: AJAX requests and custom login forms.

**The last pieces of the puzzle**

How many web applications *don’t* use AJAX to some degree now? And how many really want to use the stock login form for their application? It’s fine for internal use, but I wouldn’t recommend it for anything that’s customer facing. Let’s start with AJAX.

**Securing AJAX requests**

Dynamic user interfaces based on AJAX bring a new set of problems to access control. It’s very easy to deal with a standard request that requires authentication: simply redirect the user to the login page and then redirect them back to the target page if the authentication is successful. But such a redirect doesn’t work well with AJAX. So what do you do?

The plugin gives you a way to deal with AJAX request differently to normal ones. When an AJAX request requires authentication, Spring Security redirects to the authAjax action in LoginController rather than auth. But wait, that’s still a redirect right? Yes, but you can implement the authAjax to send an error status or render JSON - basically anything that the client Javascript code can handle.

Unfortunately, the LoginController provided by the plugin doesn’t implement authAjax at this time, so you will have to do add it yourself:

import javax.servlet.http.HttpServletResponse

class LoginController {

...

def authAjax = {

response.sendError HttpServletResponse.SC\_UNAUTHORIZED

}

...

}

This is a very simple implementation that returns a 401 HTTP status code. How do we deal with such a response? That depends on what you use to implement AJAX in the browser. The example Hubbub application uses adaptive AJAX tags, so I’ll use that to demonstrate the kind of thing you can do. This is part of the GSP template that is used for posting new messages:

<g:form action="ajaxAdd">

<g:textArea id='postContent' name="content" rows="3" cols="50" onkeydown="updateCounter()" /><br/>

<g:submitToRemote value="Post"

url="[controller: 'post', action: 'addPostAjax']"

update="[success: 'firstPost']"

onSuccess="clearPost(e)"

onLoading="showSpinner(true)"

onComplete="showSpinner(false)"

on401="showLogin();"/>

</g:form>

As you can see, it has an on401 attribute that specifies a bit of Javascript that should be executed when the AJAX submission returns a 401 status code. That bit of Javascript can, for example, display a dynamic, client-side login form for the user to authenticate with. Hubbub uses the client-side code provided in the plugin’s [user guide](http://burtbeckwith.github.com/grails-spring-security-core/docs/manual/guide/9%20Authentication.html#9.4%20Ajax%20Authentication) to do just that.

**Note** Version 1.1 of the plugin will come with a default implementation of the authAjax action.

You can also customise the ajaxSuccess and ajaxDenied actions to send back whatever response you want. As you can see, the server-side AJAX handling is simple and easy to customise. The real work has to be done in the client code.

**Custom login forms**

It’s no longer fashionable to dedicate an entire page to the login form. These days applications are more likely to have a content-rich home page with a discrete login form located somewhere on it, perhaps only made visible by some Javascript magic. It’s easy enough to provide your own dedicated login page (simply edit the auth action in LoginController and its associated GSP view to your heart’s content), but what about a login panel?

It’s not as hard as you might think. First of all, you need to decide where users should be redirected to when authentication is required. As you’ve probably gathered, this is /login/auth by default. Changing that default is as easy as adding a setting to Config.groovy:

grails.plugins.springsecurity.auth.loginFormUrl = '/'

This line tells the plugin to redirect to the home page whenever authentication is required. All you then need to do is add a login panel to the home page. Here’s an example GSP form that might go in such a panel:

<form method="POST" action="${resource(file: 'j\_spring\_security\_check')}">

<table>

<tr>

<td>Username:</td><td><g:textField name="j\_username"/></td>

</tr>

<tr>

<td>Password:</td><td><input name="j\_password" type="password"/></td>

</tr>

<tr>

<td colspan="2"><g:submitButton name="login" value="Login"/></td>

</tr>

<tr>

<td colspan="2">try "glen" or "peter" with "password"</td>

</tr>

</table>

</form>

The key points here are:

1. the form must use the POST method;
2. the form must be submitted to <context>/j\_spring\_security\_check;
3. the username field must have the name ‘j\_username’;
4. the password field must have the name ‘j\_password’; and
5. any “remember me” field must have the name ’\_spring\_security\_remember\_me’.

As long as these requirements are satisfied, the login form will work perfectly. Well, not quite perfectly. If an authentication attempt fails via your login form, you will find yourself redirected back to the old login page. Fortunately, this is quickly rectified by adding another configuration setting:

grails.plugins.springsecurity.failureHandler.defaultFailureUrl = '/'

And that’s all you need for a fully functioning login form! There are plenty of other options available to fine tune the behaviour, but you now have the basics on which to build.

This article has really only scratched the surface of the Spring Security plugin. I haven’t mentioned HTTP Basic and Digest Authentication, events, salted passwords and more. That doesn’t even include the other plugins that provide extra features such as alternative authentication mechanisms and access control lists (ACLs). But what you have read so far will enable you to get a fully working access control system up and running in no time. You will then be able to extend and customise as the need arises, knowing that Spring Security has more features than you will probably ever need.