Extracted from:

Grails 2: A Quick-Start Guide

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Grails 2

A Quick-Start Guide

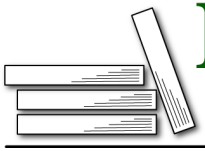
Dave Klein
Ben Klein

GRAILS

Grails 2: A Quick-Start Guide

Dave Klein
Ben Klein

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (project manager)
Potomac Indexing, LLC (indexer)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-77-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2013

CHAPTER 8

Knock, Knock: Who's There? Grails Security

Our customer keeps asking us when we are going to add security. We keep telling him, "As soon as we need it." But seriously, as we progress with the TekDays application, it's going to be very helpful to know who's using the application. Not only would that allow us to limit access to certain data or areas of the application, but it would also let us be more intelligent about what we display to our users.

Our goal this time around is to implement a simple security system and see how we can use it to provide a more customized user experience.

Grails Security Options

Grails provides several options when it comes to security, from rolling your own with *controller interceptors* and *filters* to using plugins for the more popular Java security frameworks out there. As of this writing, the main Grails plugin repository has forty-two security-related plugins.

There are plugins for Apache Shiro, CAS, Spring Security, Facebook Connect, and more. There is also the simple yet effective Authentication plugin, which doesn't rely on any external libraries. There are plugins for Captchas, OpenID plugins...you get the picture. For your *next* Grails application, it would be wise to spend some time looking at these plugins to see whether one or more of them might meet your needs.[1] For this project, however, we are going to implement our own solution using Grails filters.

Logging In

Before we get into creating filters and building our security system, let's talk about what we want the system to do. First, we want to know who is currently

1. http://grails.org/plugins/

using the system; that is, are they an anonymous user (which is fine), or are they represented by a TekUser instance? Next, we want to restrict access to certain areas of the application based on the current user. For example, only organizers should be able to edit a TekEvent instance, and only organizers *or* volunteers should be able to participate in the event's forum.

For the first step, we will need some sort of login process. We will create two new actions in the TekUserController: login and logout. We will also create a new login view.

Open TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy, and add the following action:

```
def login() {
}
```

Interestingly, we don't need anything in this action; simply having an action with this name will cause the GSP that we are about to create to be rendered. Let's create TekDays/grails-app/views/tekUser/login.gsp and give it the following code:

security.2/TekDays/grails-app/views/tekUser/login.gsp

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <meta name="layout" content="main" />
    <title>Login</title>
  </head>

  <body>
    <g:if test="${flash.message}">
      <div class="message">${flash.message}</div>
    </g:if>

    <g:form action="validate">
      <table>
        <tr class="prop">
          <td class="name">
            <label for="username">User Name:</label>
          </td>

          <td class="value">
            <input type="text" id="username" name="username" value="">
          </td>
        </tr>

        <tr class="prop">
          <td class="name">
            <label for="password">Password:</label>
          </td>
```

```
          <td class="value">
            <input type="password" id="password" name="password" value="">
          </td>
        </tr>

        <tr>
          <td>
          </td>
          <td>
            <input type="submit" value="login"/>
          </td>
        </tr>
      </table>

    </g:form>
  </body>

</html>
```

Simple enough. After a standard Grails message block (which we will need if there are problems during login), we have an HTML form with fields for username and password, followed by a submit button. This page will be merged with our standard header because of this line: <meta name="layout" content="main" />. The final result can be seen in the following figure.

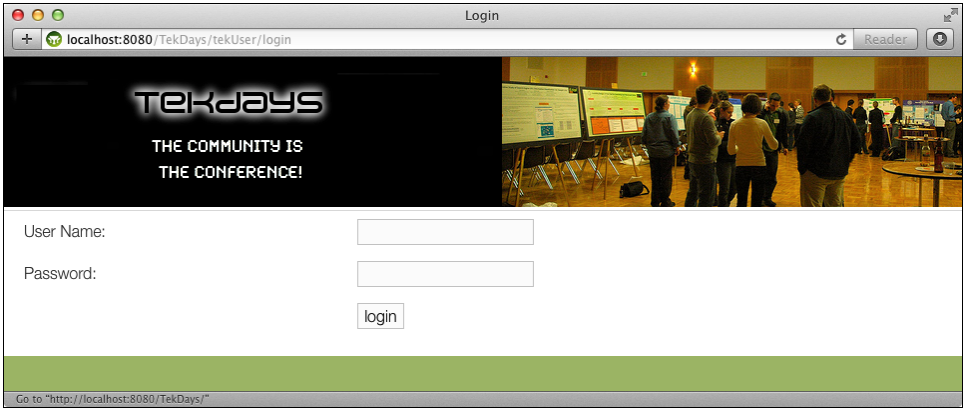


Figure 37—The login page

One important point about this code is the action that we've assigned to the <g:form>: validate. This action will be called when we submit the form. It will reside in the TekUserController and will use the form data to load an existing TekUser if found. We'll create this action now. Open TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy, and add the following action:

```
def validate() {
    def user = TekUser.findByUserName(params.username)
    if (user && user.password == params.password){
        session.user = user
        redirect controller:'tekEvent', action:'index'
    }

    else{
      flash.message = "Invalid username and password."
      render view:'login'
    }
}
```

Joe asks:
We Aren't Going to Use Plain-Text Passwords, Are We?

Don't panic, Joe. This is only a simple example. Before we would put this application in production, we would change this to use encrypted passwords, using something like the DigestUtils class of the Apache Commons project.[a] We might also move the authentication logic to a service class. There are many ways to enhance the security of our application, all of which would fit into the basic structure we are using here.

Our goal for this example is to show the use of Grails filters and to show the structure of a simple authentication system.

a. See http://commons.apache.org/proper/commons-codec/userguide.html.

This action, rather than the login action, does the real work of logging a user into the system. In the first line, we define the variable user, to which we assign the result of a call to TekUser.findByUserName(params.username). Next we check to see whether our user has a value and, if so, whether its password matches params.password. If both of those things are true, then we'll stuff this TekUser instance in the session for later use and call the redirect() method to send the user to the index action of the TekEvent controller. If either is false, we add a message to flash and call the render() method to redisplay the login view.

You'll notice that we used two methods that we didn't define anywhere. The redirect() and render() are added to all controller classes at runtime by Grails. The redirect() method will perform an HTTP redirect. That is, it will return a response to the client that will cause it to make a subsequent call to the URL that is created by the controller and action parameters.

The render() method is very versatile. We used it earlier to respond to an Ajax call. In that instance, we passed it a *template*; here we pass it a *view*. In both cases,

the end result was to send a chunk of text back to the client. This method can also be used to render XML, JSON, or any arbitrary text to the client.

In this action, we write to session, which is a Map stored in the *Session* scope. Anything we put there will be available as long as this user is interacting with our application. And since it is a Groovy Map, we can add new key/value pairs by assigning a value to a nonexistent key. There was no user key in session, but we added the key and assigned the value in the following line: session.user = user. We did the same with flash, which is a Map stored in a special scope that lasts for this request and the next, after which the values we put in will be cleared out.

Now that we have a login page and a process for logging a user in, let's see how we can use filters to prompt the user to log in at the appropriate times.

Filters

Filters allow us to hook into, or intercept, the processing of a request. There are interceptors for before, after, and afterView. There are many uses for filters, and you can have as many filters as you need in an application. In our case, we'll use a filter to determine whether a user is logged in when they try to access a "secure" page.

This must be sounding like a broken record (does anyone remember what that is?), but Grails makes implementing filters a snap. Create a Groovy class with a name ending in *Filters*, and place it in the grails-app/conf directory. In this class, define a code block called filters, and then include individual filters as if they were methods. Each filter (method) can take named parameters for controller and action. Calls to this controller and action pair will be intercepted by this filter. (An asterisk can be used as a wildcard to represent any controller or action.) But enough chatter—let's get to the code.

Create a new file called TekDays/grails-app/conf/SecurityFilters.groovy. Open this file, and add the following code:

security.2/TekDays/grails-app/conf/SecurityFilters.groovy

```
class SecurityFilters {
    def filters = {
        doLogin(controller:'*', action:'*'){
            before = {
                if (!controllerName)
                    return true
                def allowedActions = ['show', 'index', 'login',
                                      'validate']
                if (!session.user && !allowedActions.contains(actionName)){
                    redirect(controller:'tekUser', action:'login',
```

```
                        params:['cName': controllerName,
                                'aName':actionName])
                return false
            }
        }
    }
}
```

In our SecurityFilters class, we create a single filter called doLogin with a before interceptor. We use wildcards for both controller and action parameters, which means this filter will be called for all actions. We don't actually want to require the user to log in for every action, so we will fine-tune this filter further.

Every filter has certain properties injected into it by Grails; among these are controllerName and actionName. These represent the original controller and action that the user was trying to access before the filter so rudely interrupted. We will use the actionName to determine whether we really want to filter this call. We'll do this in two steps. In the first step, we'll check to see whether we have a controllerName. If we don't, then we can assume the user is going to the default home page (index.gsp), in which case we will return true. For the second step, we define a List variable with the names of actions that we want to *allow*. Along with the innocuous actions show and index, we included the login and validate actions to avoid unintended login loops. Then in our if comparison, we check to see whether this list contains the current actionName.

The other thing we check in the if comparison is whether we already have a user in the session. If we do not have a user and the current action is not in the allowedActions list, we redirect to the login action of the TekUserController and pass along the controllerName and actionName values in the params parameter. (We'll need them shortly.) In the final line, we return false, which will prevent any other filters (or the original action) from being called.

Now to make this all work nicely, we have to go back and make a few changes to our login view and the two controller actions we added to TekUserController. We want to take advantage of the controllerName and actionName values from the filter. When the filter redirects to the login action, it will pass these values in the params, so we need to do something with them to keep them available. Open TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy, and modify the empty login action like so:

security.2/TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy

```
def login() {
    if (params.cName)
        return [cName:params.cName, aName:params.aName]
}
```

This code checks to see whether those two values are available in params and, if so, passes them on to the view in the returned Map. Next, we'll modify the view to pass these values on to the validate action. Open TekDays/grails-app/views/tekUser/login.gsp, and add the following hidden input elements somewhere inside the <g:form>.

security.2/TekDays/grails-app/views/tekUser/login.gsp
```
<input type="hidden" name="cName" value="${cName}">
<input type="hidden" name="aName" value="${aName}">
```

Now when the form is submitted, the controllerName and actionName values from the filter will be passed on to the validate action. We will now use these values to redirect the user to their original destination on successful login.

Open TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy, and modify the validate action to look like this:

security.2/TekDays/grails-app/controllers/com/tekdays/TekUserController.groovy
```
def validate() {
    def user = TekUser.findByUserName(params.username)

    if (user && user.password == params.password){
        session.user = user

        if (params.cName)
            redirect controller:params.cName, action:params.aName
        else
            redirect controller:'tekEvent', action:'index'
    } else{
        flash.message = "Invalid username and password."
        render view:'login'
    }
}
```

What we're doing here is checking to see whether the controllerName and actionName (using shortened variables) are available. If they are, we use them to redirect the user; otherwise, we redirect them to the index action of the TekEventController as before. We can come back and change that to the *home* page later (after we add one).

This feature is a bit tricky to show in screenshots, but go ahead and try it. Run the application, and go to the default home page. Choose any of the controller links, and you should come to the list view. Click the "New" button, and you should see the login screen shown in the last figure. Log in using the credentials of one of the users we created earlier, and you should be redirected to the create view that you were aiming at. Good deal!