

Mastering Grails: Authentication and authorization

Securing your Grails application

Skill Level: Introductory

Scott Davis (scott@thirstyhead.com)
Founder, ThirstyHead.com
ThirstyHead.com

28 Apr 2009

Grails provides all the basic building blocks you need to put together a secure Web application, ranging from a simple login infrastructure to role-based authorization, and in this installment of *Mastering Grails*, Scott Davis gives you a hands-on lesson in securing your Grails application. You'll also learn about some plug-ins that can help you extend your applications' security capabilities in new directions.

In this article, I continue building a "tiny little blog" named *Blogito*. I stubbed out `Users` in the previous article ("[Rewiring Grails with custom URIs and codecs](#)") because the `name` field was an integral part of the URI. Now it's time to implement the `User` subsystem fully. You'll learn how to enable logins, limit activity based on whether or not the `User` is logged in, and even add in some authorization based on the `User`'s role.

To start, `Users` need a way to log in so that they can post new entries.

Authentication

Authentication is probably a good idea for a blog server that supports multiple users. You certainly don't want John Doe posting blog entries as Jane Smith, accidentally or otherwise. Setting up an authentication infrastructure answers the question, "Who are you?" In just a moment, you'll add in a bit of authorization as well. Authorization answers the question, "What are you allowed to do?"

Listing 1 shows the `grails-app/domain/User.groovy` file that you created [last time](#):

Listing 1. The User class

```
class User {
    static constraints = {
        login(unique:true)
        password(password:true)
        name()
    }

    static hasMany = [entries:Entry]

    String login
    String password
    String name

    String toString(){
        name
    }
}
```

The `login` and `password` fields are in place. All you need to do now is provide a controller and a form. Create `grails-app/controllers/UserController.groovy` and add the code, as shown in Listing 2:

Listing 2. Adding login, authenticate, and logout closures to UserController

```
class UserController {
    def scaffold = User

    def login = {}

    def authenticate = {
        def user = User.findByLoginAndPassword(params.login, params.password)
        if(user){
            session.user = user
            flash.message = "Hello ${user.name}!"
            redirect(controller:"entry", action:"list")
        }else{
            flash.message = "Sorry, ${params.login}. Please try again."
            redirect(action:"login")
        }
    }

    def logout = {
        flash.message = "Goodbye ${session.user.name}"
        session.user = null
        redirect(controller:"entry", action:"list")
    }
}
```

The empty `login` closure simply means that visiting <http://localhost:9090/blogito/user/login> in your browser will render the `grails-app/views/user/login.gsp` file. (You'll create that file in just a moment.)

The `authenticate` closure uses a convenient GORM method — `findByLoginAndPassword()` — to do just what it sounds like: find the `User` in the database whose `login` and `password` match the values typed in the form fields

and made available via the `params` hashmap. If the `User` exists, add it to the session. If not, redirect back to the login form to give the `User` another chance to provide the correct credentials. The `logout` closure bids adieu to the `User`, removes him or her from the session, and then redirects back to the `list` action in the `EntryController`.

Now it's time to create `login.gsp`. You can type the code shown in Listing 3 by hand, or you can:

1. Type `grails generate-views User` at the command line.
2. Copy `create.gsp` to `login.gsp`.
3. Whittle down the resulting code.

Listing 3. login.gsp

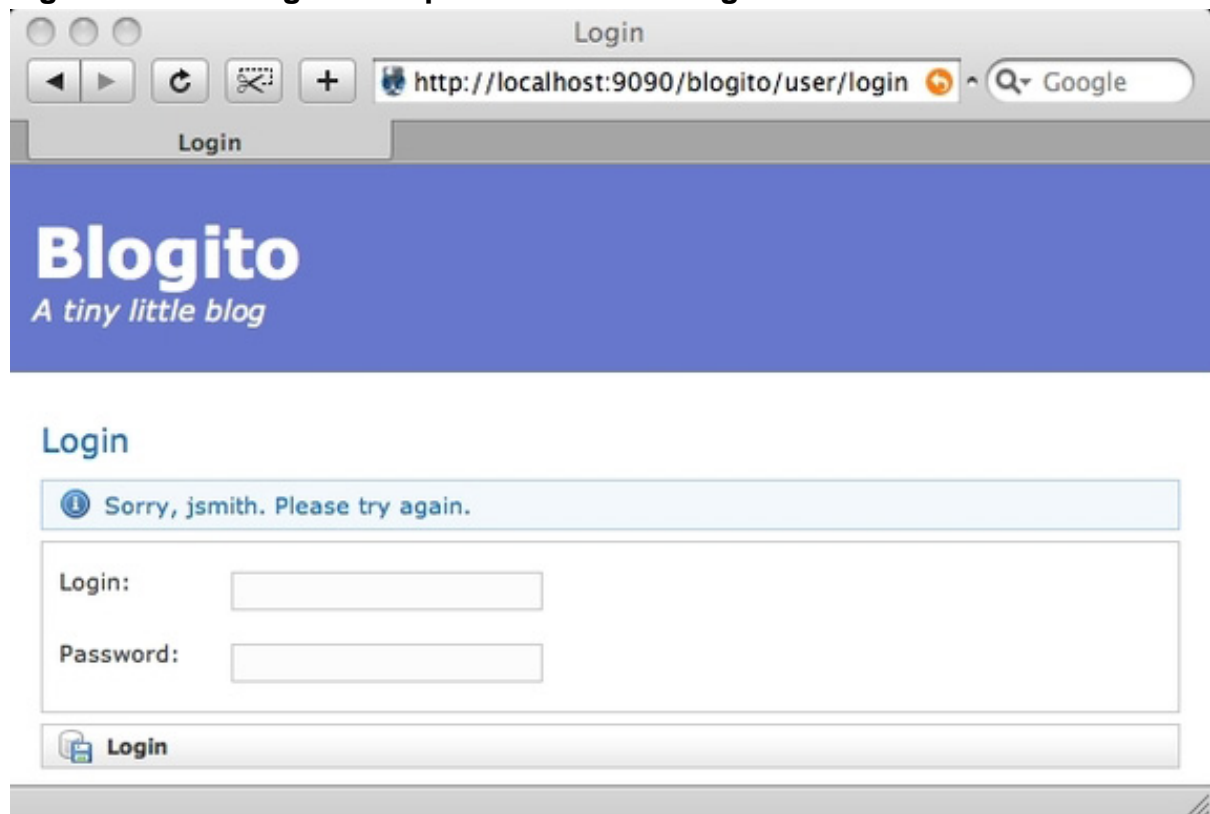
```
<html>
  <head>
    <meta name="layout" content="main" />
    <title>Login</title>
  </head>
  <body>
    <div class="body">
      <h1>Login</h1>
      <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
      </g:if>
      <g:form action="authenticate" method="post" >
        <div class="dialog">
          <table>
            <tbody>
              <tr class="prop">
                <td class="name">
                  <label for="login">Login:</label>
                </td>
                <td>
                  <input type="text" id="login" name="login"/>
                </td>
              </tr>
              <tr class="prop">
                <td class="name">
                  <label for="password">Password:</label>
                </td>
                <td>
                  <input type="password" id="password" name="password"/>
                </td>
              </tr>
            </tbody>
          </table>
        </div>
        <div class="buttons">
          <span class="button">
            <input class="save" type="submit" value="Login" />
          </span>
        </div>
      </g:form>
    </div>
```

```
</body>
</html>
```

Notice that the action of the form is `authenticate`, which matches the name of the closure in `UserController.groovy`. The names in the input elements — `login` and `password` — correspond to `params.login` and `params.password` in the `authenticate` closure.

Type `grails run-app` and take your authentication infrastructure out for a spin. Try logging in as `jsmith` with a password of `foo`. (Remember that in "[Rewiring Grails with custom URIs and codecs](#)" you seeded Blogito with a couple of users in `grails-app/conf/BootStrap.groovy`.) Your login should fail, as shown in Figure 1:

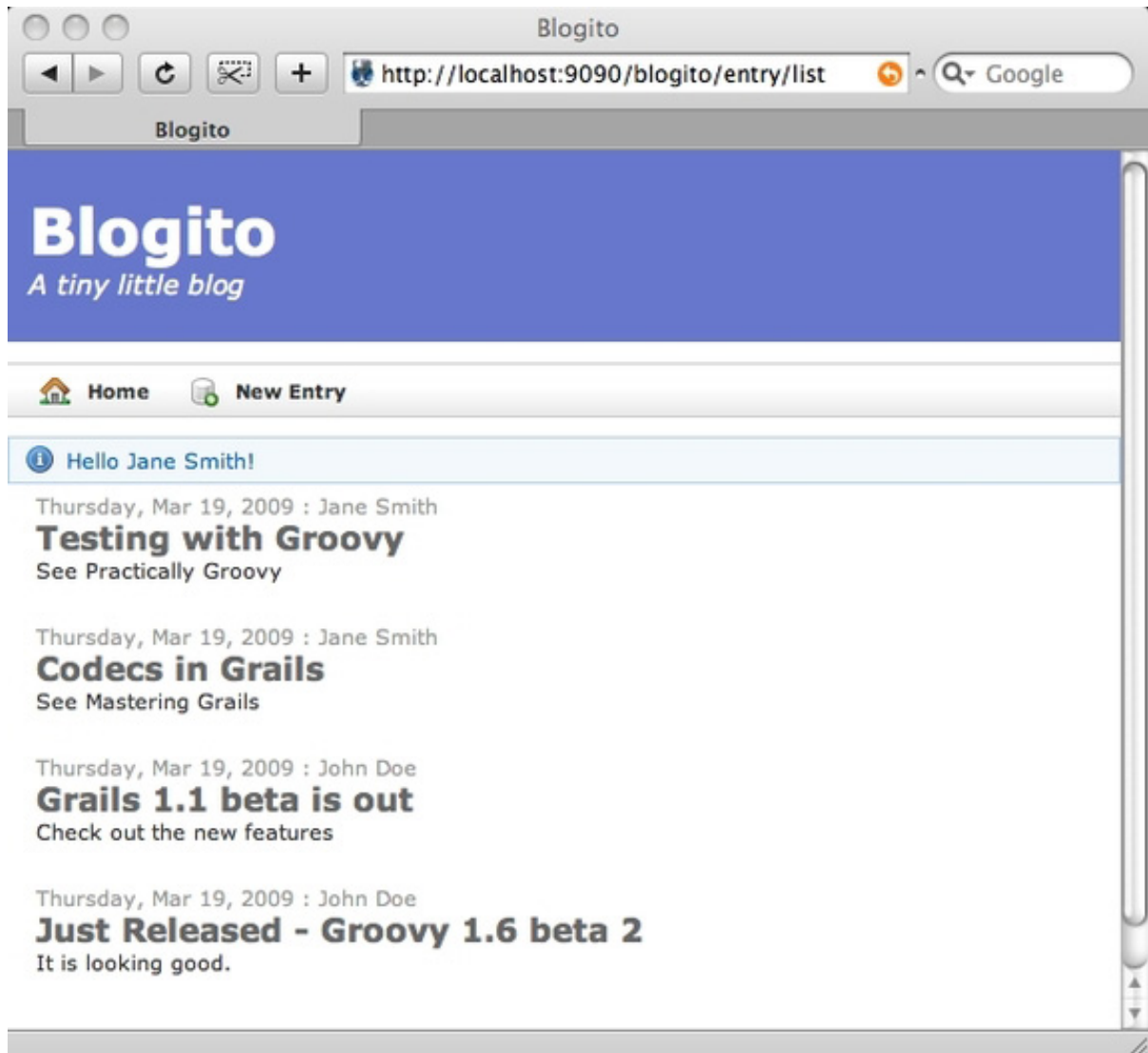
Figure 1. Failed login attempt with error message



Try again as `jsmith` with a password of `wordpass`. This time it should succeed.

If the welcome message doesn't appear in `grails-app/views/entry/list.gsp` — and it shouldn't — simply copy the `<g:if test='${flash.message}'>` block from `login.gsp` to the top of the `list.gsp` file. Log in again as `jsmith` to verify that the message now appears as shown in Figure 2:

Figure 2. Flash message confirming a successful login



Now that you are sure that authorization works, you should create a `TagLib` that makes it easy to log in and log out.

Creating an authorization `TagLib`

Web sites like Google and Amazon offer an unobtrusive text link in the header that allows you to log in and log out. You can do the same in Grails with just a few lines of code.

To begin, type `grails create-tag-lib Login` at the command prompt. Add the code in Listing 4 to the newly created `grails-app/taglib/LoginTagLib.groovy`:

Listing 4. `LoginTagLib.groovy`

```
class LoginTagLib {
  def loginControl = {
    if(session.user){
      out << "Hello ${session.user.name} "
      out << ""[${link(action:"logout", controller:"user"){Logout}}]""
    } else {
      out << ""[${link(action:"login", controller:"user"){Login}}]""
    }
  }
}
```

Now, add the new `<g:loginControl>` tag to `grails-app/views/layouts/_header.gsp`, as shown in Listing 5:

Listing 5. Adding the `<loginControl>` tag to the header

```
<div id="header">
  <p><g:link class="header-main" controller="entry">Blogito</g:link></p>
  <p class="header-sub">A tiny little blog</p>

  <div id="loginHeader">
    <g:loginControl />
  </div>
</div>
```

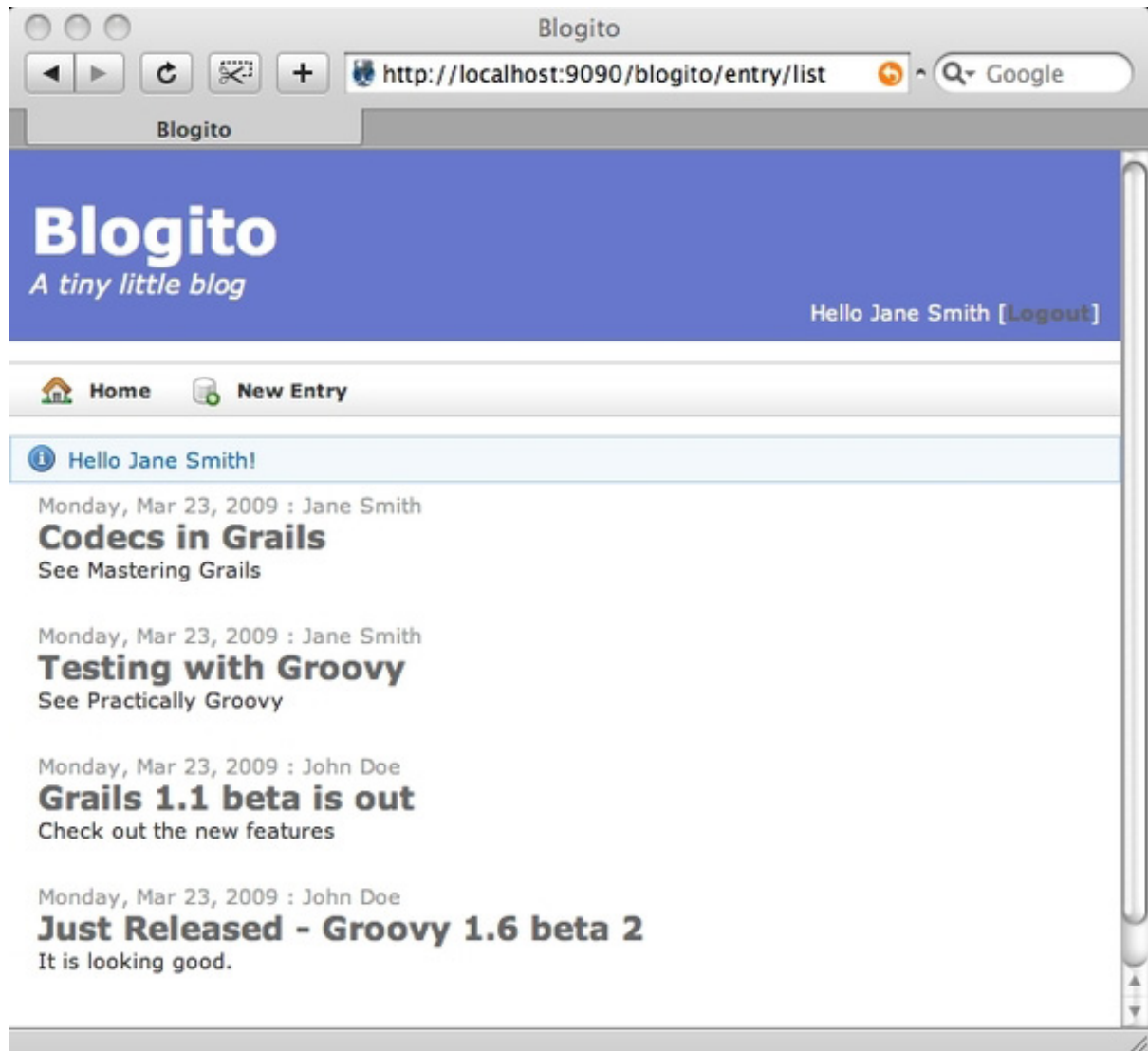
To wrap up, add a bit of CSS formatting for the `loginHeader <div>` to `web-app/css/main.css`, as shown in Listing 6:

Listing 6. CSS formatting for the `loginHeader <div>`

```
#loginHeader {
  float: right;
  color: #fff;
}
```

When you restart Grails and log in as `jsmith`, your screen should look like Figure 3:

Figure 3. Login TagLib in action



Basic authorization

Now that Blogito knows who you are, the next step is to limit what you can do. For example, anyone should be able to read an `Entry`, but only logged-in users should be allowed to create, update, and delete an `Entry`. To accomplish this, Grails offers a `beforeInterceptor` that, as the name implies, gives you a hook that is perfect for authorizing activities before the target closure is invoked.

Add the code in Listing 7 to `EntryController`:

Listing 7. Adding authorization to `EntryController`

```
class EntryController {
```

```
def beforeInterceptor = [action:this.&auth, except:["index", "list", "show"]]

def auth() {
  if(!session.user) {
    redirect(controller:"user", action:"login")
    return false
  }
}

def list = {
  //snip...
}
```

A subtle but important difference between `auth` and `list` is that `list` is a closure, whereas `auth` is a private method. (Closures use an equals sign in the definition; methods use parentheses.) Closures are exposed to the end user as a URI; methods are not accessible from the browser.

The `auth` method checks to see if a `User` is in the session. If not, it redirects to the login screen and returns `false`, blocking the original closure call.

The `auth` method gets invoked before each closure call by the `beforeInterceptor`. The action uses Groovy notation to point to `this` class's `auth` method using the ampersand (&) character. The `except` list contains the closures that should be exempted from the `auth` call. You can replace `except` with `only` if you'd like to intercept just a few closure calls. (For more information on the `beforeInterceptor`, see [Resources](#).)

Restart Grails and test the `beforeInterceptor`. Try to visit `http://localhost:9090/blogito/entry/create` without being logged in. You should be redirected to the login screen. Log in as `jsmith` and try it again. This time you should be able to create a new `Entry` successfully.

Fine-grained authorization

The coarse-grained authorization offered by the `beforeInterceptor` is a start, but you can add authorization hooks to individual closures as well. For example, as things stand now, any logged-in `User` — not just the original author — can edit any `Entry`. You can close that security hole by adding four well-placed lines of code to the `edit` closure in `EntryController.groovy`, as shown in Listing 8:

Listing 8. Adding authorization to the edit closure

```
def edit = {
  def entryInstance = Entry.get( params.id )

  //limit editing to the original author
  if( !(session.user.login == entryInstance.author.login) ){
    flash.message = "Sorry, you can only edit your own entries."
  }
}
```

```
        redirect(action:list)
    }

    if(!entryInstance) {
        flash.message = "Entry not found with id ${params.id}"
        redirect(action:list)
    }
    else {
        return [ entryInstance : entryInstance ]
    }
}
```

You can (and should) lock down the `delete` and `update` closures with the same four lines of code. If the prospect of copying and pasting duplicate code around is distasteful (and it should be), you can create a single private method and call it in all three closures. If you find that you are using the same `beforeInterceptor` and private methods across many controllers, you can factor out the common behavior into a single master controller and have the other controllers extend it as you would typically do with any Java class.

You can add one more thing to your authorization infrastructure to make it more robust: roles.

Adding roles

Assigning a role to `Users` is a convenient way to group them together. You can then assign permissions to the group instead of to individuals. For example, right now anyone can create a new `User`. Just checking to see if the person is logged in isn't secure enough. I'd like to limit the ability to manage `User` accounts to an administrator.

Listing 9 adds a role field to `User`, as well as a constraint that limits the values to either `author` or `admin`:

Listing 9. Adding a role field to the User

```
class User {
    static constraints = {
        login(unique:true)
        password(password:true)
        name()
        role(inList:["author", "admin"])
    }

    static hasMany = [entries:Entry]

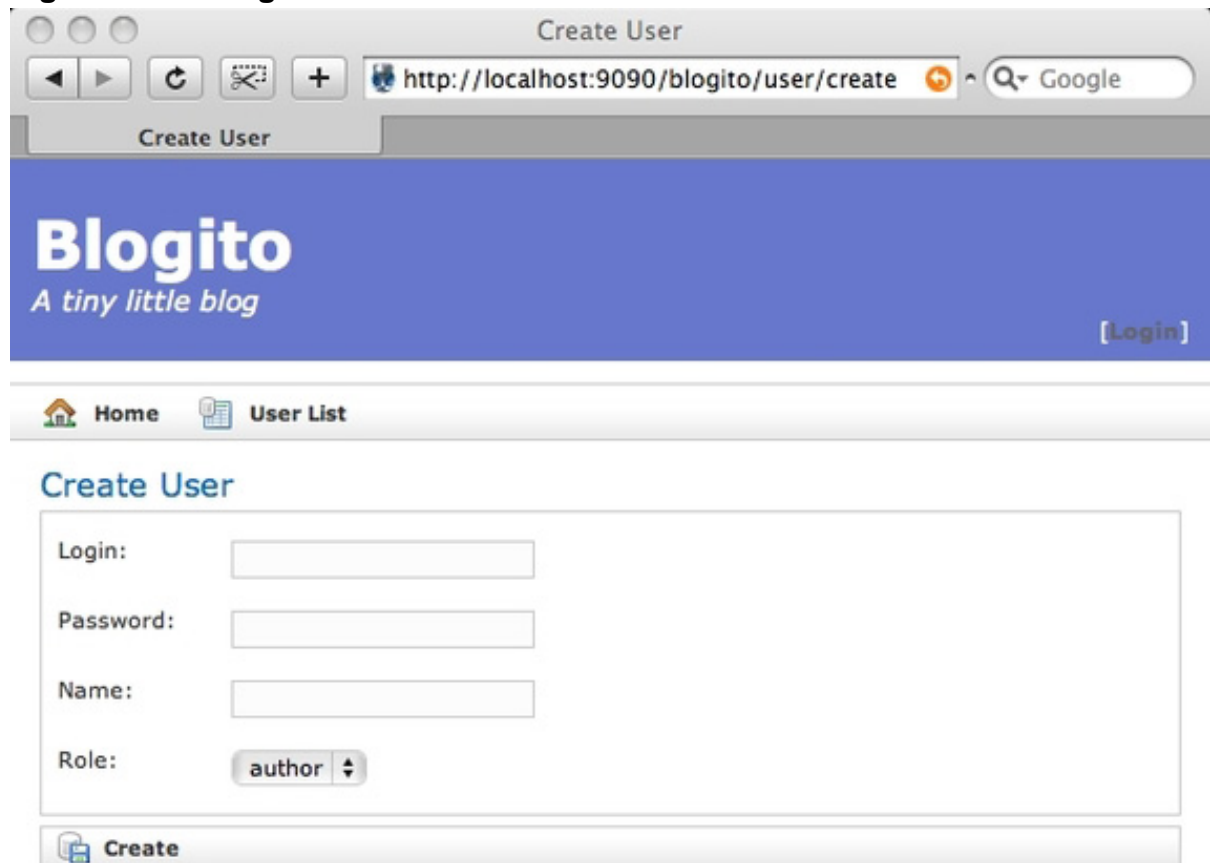
    String login
    String password
    String name
    String role = "author"

    String toString(){
        name
    }
}
```

```
}  
}
```

Notice that the `role` defaults to `author`. The `inList` constraint presents a combo box with the only two valid choices. Figure 4 shows this in action:

Figure 4. Limiting new user roles to either author or admin



The screenshot shows a web browser window titled "Create User" with the URL `http://localhost:9090/blogito/user/create`. The page header for "Blogito" includes the tagline "A tiny little blog" and a "[Login]" link. A navigation bar contains "Home" and "User List" links. The main form area is titled "Create User" and contains the following fields:

- Login:
- Password:
- Name:
- Role:

A "Create" button is located at the bottom of the form.

Create an `admin` `User` in `grails-app/conf/BootStrap.groovy`, as shown in Listing 10. Don't forget to add the `author` `role` to the two existing `Users`.

Listing 10. Adding an admin User

```
import grails.util.GrailsUtil  
class BootStrap {
```

```

def init = { servletContext ->
  switch(GrailsUtil.environment){
  case "development":
    def admin = new User(login:"admin",
                        password:"password",
                        name:"Administrator",
                        role:"admin")

    admin.save()

    def jdoe = new User(login:"jdoe",
                      password:"password",
                      name:"John Doe",
                      role:"author")

    //snip...

    def jsmith = new User(login:"jsmith",
                        password:"wordpass",
                        name:"Jane Smith",
                        role:"author")

    //snip...

    break

  case "production":
    break
  }
}
def destroy = {
}

```

And finally, add the code in Listing 11 to limit all `User` account activity to people who have the `admin` role:

Listing 11. Limiting User account management to the admin role

```

class UserController {

  def beforeInterceptor = [action:this.&auth,
                        except:["login", "authenticate", "logout"]]

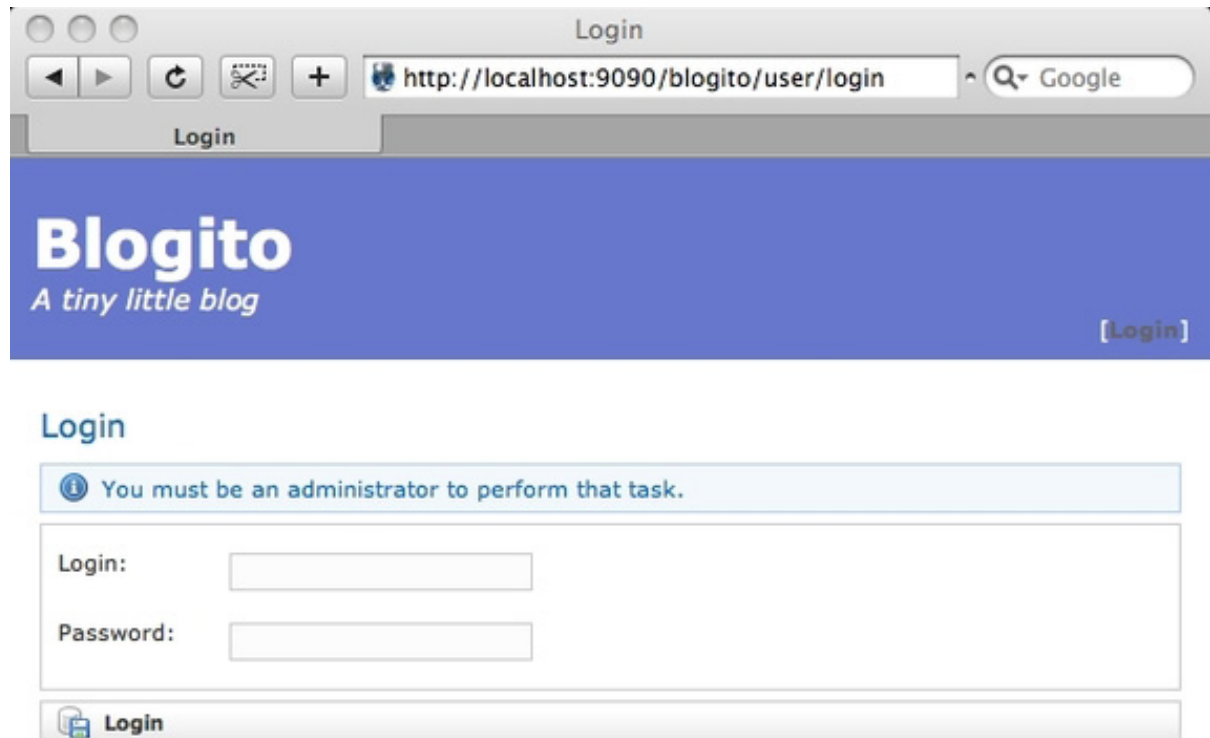
  def auth() {
    if( !(session?.user?.role == "admin") ){
      flash.message = "You must be an administrator to perform that task."
      redirect(action:"login")
      return false
    }
  }

  //snip...
}

```

To test out the role-based authorization, log in as `jsmith` and then try to visit `http://localhost:9090/blogito/user/create`. You should be redirected to the login screen, as shown in Figure 5:

Figure 5. Blocking access for nonadministrators



Now log in as the `admin` user. You should be able to access all of the closures.

Taking it to the next level with plug-ins

The "tiny little" authentication and authorization system for this "tiny little" blog application is now in place. You can extend it easily in new directions. Perhaps you would like `Users` to be able to manage their own accounts but not others. Maybe admins should have the capability to edit all `Entries`, not just their own. In each case, you are no more than a couple of strategically placed lines of code away from adding the new functionality.

It's common to mistake simplicity for lack of power. Blogito is still fewer than 200 lines of code — and that includes the unit and integration tests. Type `grails stats` at the command line to confirm this. The results are shown in Listing 12. But

Blogito's lack of complexity doesn't mean that it isn't ready for prime time.

Listing 12. The size of the "tiny little" application

```
$ grails stats
```

Name	Files	LOC
Controllers	2	95
Domain Classes	2	32
Tag Libraries	2	21
Unit Tests	5	20
Integration Tests	1	10
Totals	12	178

My goal since the first article in this series has been to show you the native power of core Grails and the concise expressiveness of the Groovy language. For example, once you understand codecs in Grails, it's easy to hash the passwords stored in the database instead of displaying them in the clear. (For more information on the HashCodec, see [Resources](#).) Create `grails-app/utils/HashCodec.groovy` and add the code in Listing 13:

Listing 13. Creating a simple HashCodec

```
import java.security.MessageDigest
import sun.misc.BASE64Encoder
import sun.misc.CharacterEncoder

class HashCodec {
    static encode = { str ->
        MessageDigest md = MessageDigest.getInstance('SHA')
        md.update(str.getBytes('UTF-8'))
        return (new BASE64Encoder()).encode(md.digest())
    }
}
```

With the HashCodec in place, simply change the references to `User.password` in the login, save, and update closures in `UserController` to `User.password.encodeAsHash()`. It's amazing that with a mere 10 lines of code, you have added a whole new level of sophistication to your application.

But a line of diminishing returns gets crossed at some point. In the case of Grails, the classic "build or buy" question gets translated to "build or download a plug-in." A number of plug-ins at <http://grails.org/plugin/list#security+tags> try to solve the authentication and authorization challenge in a way that is a quick `grails install-plugin` away.

For example, the Authentication plug-in offers some nice features, such as allowing Users to sign up for an account instead of forcing an admin to create one on their

behalf. You can then configure the plug-in to send a confirmation message to the `User` saying, "A new user account was created using this email address. Click on this link to verify your new account."

The OpenID plug-in takes a different approach. Rather than forcing your end users to create yet another username and password combination that they will surely forget, authentication is delegated to the OpenID provider of their choice. The Lightweight Directory Access Protocol (LDAP) plug-in takes a similar approach, allowing your Grails application to leverage your existing LDAP infrastructure.

The Authentication and OpenID plug-ins limit themselves to the authentication story. Others offer authorization solutions as well. The JSecurity plug-in brings a whole security framework into play, offering boilerplate domain classes for `Users`, `Roles`, and `Permissions`. The Spring Security plug-in leverages the Spring Security (formerly Acegi Security) libraries, allowing you to reuse your existing Spring Security knowledge and source code.

As you can see, multiple authentication and authorization strategies are available in Grails, because requirements vary widely among applications. With each step up in capabilities, your application takes an unavoidable corresponding step up in complexity. I've used many of the plug-ins I've listed here in production applications, but only after I was sure that the benefits outweighed the simple hand-rolled strategy I presented to you first.

Conclusion

You have now secured Blogito. `Users` have both a way to log in and log out, and a convenient set of links for doing so, thanks to the `LoginTagLib` you created. In some situations, simply logging into the application is security enough, as demonstrated by the `beforeInterceptor` in `EntryController` that verifies authentication. In other cases, roles bring another layer of sophistication in the form of authorization. Adding a simple role to the `User` allows you to limit user-management access to administrators.

Now that Blogito is secure, the next *Mastering Grails* article can focus on the main task at hand — providing a way for authenticated users to upload files and a way for end users to subscribe to an Atom feed. With that in place, Blogito will truly be a blog application. Until then, have fun mastering Grails.

Resources

Learn

- [Mastering Grails](#): Read more in this series to gain a further understanding of Grails and all you can do with it.
- [Grails](#): Visit the Grails Web site.
- [Grails Framework Reference Documentation](#): The Grails bible.
- [Action Interceptors](#): Read more about intercepting processing based on request, session, or application state.
- [Simple Dynamic Password Codec](#): You can create one-way hashed password strings with this Grails codec.
- [Groovy Recipes](#) (Scott Davis, Pragmatic Programmers, 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- [Practically Groovy](#): This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- [Groovy](#): Learn more about Groovy at the project Web site.
- [AboutGroovy.com](#): Keep up with the latest Groovy news and article links.
- [Technology bookstore](#): Browse for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Grails](#): Download the latest Grails release.
- Check out these security plug-ins for Grails:
 - [Authentication](#)
 - [OpenID](#)
 - [LDAP](#)
 - [JSecurity](#)
 - [Spring Security](#)
- [Blogito](#): You can download the completed Blogito application.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Scott Davis

Scott Davis is an internationally recognized author, speaker, and software developer. He is the founder of [ThirstyHead.com](#), a Groovy and Grails training company. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*. He writes two ongoing article series for IBM developerWorks: *Mastering Grails* and *Practically Groovy*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.