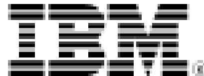


developerWorks.

Mastering Grails: File uploads and Atom syndication

Getting data in and out of Grails

Skill Level: Introductory

Scott Davis
Founder
ThirstyHead.com

09 Jun 2009

In this *Mastering Grails* installment, Scott Davis shows you how to upload files to your Grails application and set up an Atom syndication feed. With these last pieces in place, Blogito becomes a full-fledged blog server.

Over the past several *Mastering Grails* articles, you've been building a tiny little blog service (Blogito) piece by piece. In this article, Blogito will finally fulfill its destiny and become a functional blog application. You'll implement file-upload capabilities for the body of a blog entry and put together a hand-rolled Atom feed for syndication.

> **About this series**
> Grails is a modern Web development framework that mixes familiar Java technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails give you seamless integration with your legacy Java code while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

But before you get started, the authentication I added in the preceding article ("Authentication and authorization") introduced a subtle bug in the UI. Fixing it should take precedence over adding new features.

© Copyright IBM Corporation 2009. All rights reserved.

Fixing a hidden bug

As you start Grails, grails-app/conf/Bootstrap.groovy adds two users and four new blog entries. But what happens when you try to add a blog entry through the Web interface? Use the following steps to find out:

1. Log in as `jsmith`, password `wordpass`.
2. Click **New Entry**.
3. Add a title and a summary.
4. Click **Create**.

Oops! You're greeted with the following error: `Property [author] of class [class Entry] cannot be null`. How did this bug find its way into the application, especially when the bootstrap code continues to work correctly?

I had you generate the Groovy Server Pages (GSP) views in the first Blogito article ("Give your Grails applications a facelift") by typing `grails generate-views Entry`. In the subsequent articles, I made changes to the domain class and never had you go back and regenerate the views. The create.gsp view on disk got stale when I added the 1:M relationship between `Entry` and `User`, as shown in Listing 1. (Recall that `belongsTo` creates a field named `author` that is of `User` type.)

Listing 1. The 1:M relationship that broke the GSP

```
class Entry {
  static belongsTo = [author:User]

  String title
  String summary
  Date dateCreated
  Date lastUpdated
}
```

This is a painful reminder that dynamically scaffolding your views — especially in the early stages of development when the domain model is changing rapidly — is the safest way to keep everything in sync. You certainly can't rely solely on scaffolded views, but the minute you generate GSPs on disk, the responsibility for keeping them up to date shifts from Grails to you.

If you generated the views now for the `Entry` class, Grails would provide a combo box that displays a list of `Author`s, as shown in Listing 2. *Don't* do this yourself — this is just to illustrate a point. I'll offer you a couple of different options in just a moment.

© Copyright IBM Corporation 2009. All rights reserved.

Listing 2. Generated combo box for 1:M relationships

```
<g:form action="save" method="post" >
  <div class="dialog">
    <table>
      <tbody>
        <!-- SNIP -->
        <tr class="prop">
          <td valign="top" class="name">
            <label for="author">Author:</label>
          </td>
          <td valign="top"
              class="value ${hasErrors(bean:entryInstance,
                                       field:'author','errors')}">
            <g:select optionKey="id"
                      from="${User.list()}"
                      name="author.id"
                      value="${entryInstance?.author?.id}" ></g:select>
          </td>
        </tr>
        <!-- SNIP -->
      </tbody>
    </table>
  </div>
</g:form>
```

Notice the `<g:select>` element. The field's name is `author.id`. As you learned in "GORM: Funny name, serious technology," the text displayed in the list comes from the `User.toString()` method. That is also normally what would be sent back as the field value to the server when the form is submitted. In this case, the `optionKey` attribute overrides the field value, sending back the `id` of the `Author` instead. (For more information on the `<g:select>` tag, see Resources.)

The quickest way to give EntryController.groovy the `author.id` field it's looking for is to add a hidden field to the form, as shown in Listing 3. Because you must be logged in to get to the `create` action, and the logged in `User` is the `author` of the blog entry, you can safely use `session.user.id` for the value.

Listing 3. Passing the author.id field from the form

```
<g:form action="save" method="post" >
  <input type="hidden" name="author.id" value="${session.user.id}" />
  <!-- SNIP -->
</g:form>
```

For a simple application like Blogito, this is probably sufficient. It does, however, open up the possibility of a client-side hacker injecting a different value for `author.id`. To be completely safe, you can add `Entry.author` in the `save` closure instead, as shown in Listing 4:

Listing 4. Saving the author.id on the server

```
def save = {
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    def entryInstance = new Entry(params)
    entryInstance.author = User.get(session.user.id)
    if(!entryInstance.hasErrors() && entryInstance.save()) {
        flash.message = "Entry ${entryInstance.id} created"
        redirect(action:show,id:entryInstance.id)
    }
    else {
        render(view:'create',model:[entryInstance:entryInstance])
    }
}
```

This is the standard `save` closure you get when you generate the controller, plus one line of custom code. The `entryInstance.author` line gets the `User` from the database based on the `session.user.id` value and populates `Entry.author` field.

In the next section, you'll customize the `save` closure to handle file uploads, so you might as well err on the side of safety and add the code in Listing 4 to EntryController.groovy. Restart Grails and make sure that you can successfully add a new `Entry` through the HTML form.

File upload

Now that creating an `Entry` is working again, it's time to add another feature. I want users to be able to upload a file when they create a new `Entry`. The file could be HTML that contains the full body of the blog entry, or an image, or anything else. To accomplish this, you need to touch the `Entry` domain class, the `EntryController`, and the GSP views — and add a new TagLib to the mix.

To begin, take a look at grails-app/views/entry/create.gsp. Add a new field to upload the file, as shown in Listing 5:

Listing 5. Adding a file-upload field

```
<g:uploadForm action="save" method="post" >
  <!-- SNIP -->
  <tr class="prop">
    <td valign="top" class="name">
      <label for="payload">File:</label>
    </td>
    <td valign="top">
      <input type="file" id="payload" name="payload"/>
    </td>
  </tr>
</g:uploadForm>
```

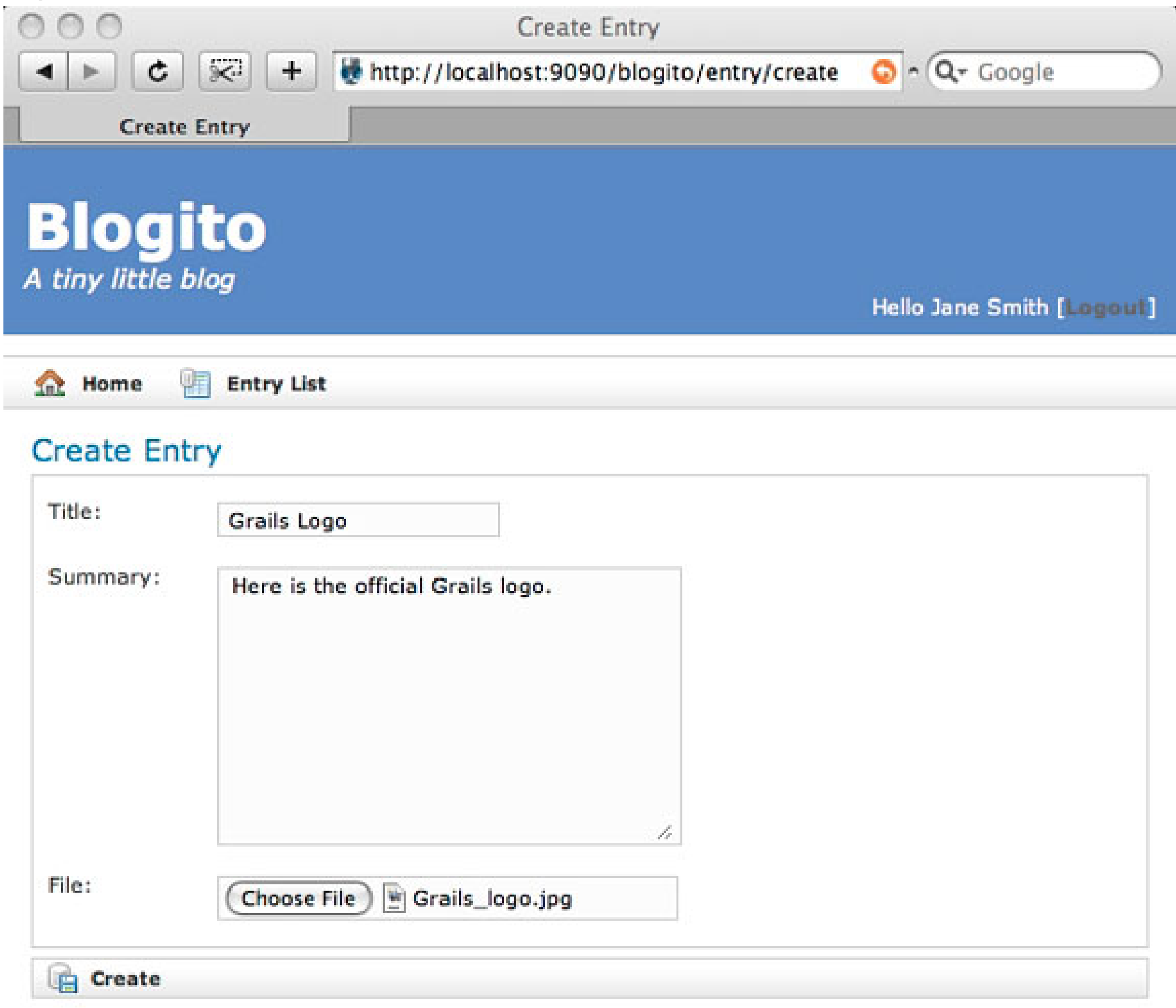
Notice that the `<g:form>` tags have been changed to `<g:uploadForm>`. This enables file uploads from the HTML form. You could also have left the `<g:form>` tags in place and simply added an `enctype="multipart/form-data"` attribute. (The default `enctype` for HTML forms is

© Copyright IBM Corporation 2009. All rights reserved.

application/x-www-form-urlencoded.)

Once the form's enctype is set correctly (or if you use <g:uploadForm>), you can add the <input type="file" /> field. This gives the user a button for browsing the local file system and selecting a file to be uploaded, as shown in Figure 1. My examples use the Grails logo; you can use any image file you'd like.

Figure 1. The Create Entry form with a file-upload field



Now that the client-side form is in place, it's time to adjust the server-side code to do something useful with the uploaded file. Open grails-app/controllers/EntryController.groovy in a text editor and add the code in Listing 6 to the save closure:

Listing 6. Showing information about the uploaded file

```
def save = {
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    def entryInstance = new Entry(params)
    entryInstance.author = User.get(session.user.id)

    //handle uploaded file
    def uploadedFile = request.getFile('payload')
    if(!uploadedFile.empty){
      println "Class: ${uploadedFile.class}"
      println "Name: ${uploadedFile.name}"
      println "OriginalFileName: ${uploadedFile.originalFilename}"
      println "Size: ${uploadedFile.size}"
      println "ContentType: ${uploadedFile.contentType}"
    }

    if(!entryInstance.hasErrors() && entryInstance.save()) {
        flash.message = "Entry ${entryInstance.id} created"
        redirect(action:show,id:entryInstance.id)
    }
    else {
        render(view:'create',model:[entryInstance:entryInstance])
    }
}
```

Notice that you use the `request.getFile()` method to get a reference to the uploaded file. Once you have that, you can do all sorts of introspection on it. Listing 7 shows the console output after the Grails logo is uploaded:

Listing 7. Console output from the uploaded file

```
Class: class org.springframework.web.multipart.commons.CommonsMultipartFile
Name: payload
OriginalFileName: Grails_logo.jpg
Size: 8065
ContentType: image/jpeg
```

Knowing that Grails uses the Spring MVC framework under the covers, it's not surprising that the uploaded file is made available to the controller as a `CommonsMultipartFile` object. In addition to exposing the name of the HTML form field, this class also gives you access to the original file name, the size in bytes, and the file's MIME type.

The next step is to save the uploaded file somewhere. Add a few more lines of code to the `save` closure, as shown in Listing 8:

Listing 8. Saving the uploaded file to disk

```
def save = {
    def entryInstance = new Entry(params)
    entryInstance.author = User.get(session.user.id)

    //handle uploaded file
    def uploadedFile = request.getFile('payload')
    if(!uploadedFile.empty){
      println "Class: ${uploadedFile.class}"
      println "Name: ${uploadedFile.name}"
      println "OriginalFileName: ${uploadedFile.originalFilename}"
      println "Size: ${uploadedFile.size}"
```


© Copyright IBM Corporation 2009. All rights reserved.

```
        println "ContentType: ${uploadedFile.contentType}"

        def webRootDir = servletContext.getRealPath("/")
        def userDir = new File(webRootDir, "/payload/${session.user.login}")
        userDir.mkdirs()
        uploadedFile.transferTo( new File( userDir, uploadedFile.originalFilename))
      }

      if(!entryInstance.hasErrors() && entryInstance.save()) {
          flash.message = "Entry ${entryInstance.id} created"
          redirect(action:show,id:entryInstance.id)
      }
      else {
          render(view:'create',model:[entryInstance:entryInstance])
      }
  }
```

Once you create the payload/jsmith directory under the Web root, you can use the `uploadedFile.transferTo()` method to save the file to disk. The `File.mkdirs()` method is nondestructive, so you can call it over and over again without worrying about losing existing files if the directory already exists.

Next, add a `String` field to the `Entry` class to store the `filename`, as shown in Listing 9. Be sure to add a constraint that allows the new field to be both `blank` (in the HTML form) and `nullable` (in the database).

Listing 9. Adding the filename field to Entry

```
class Entry {
  static constraints = {
    title()
    summary(maxSize:1000)
    filename(blank:true, nullable:true)
    dateCreated()
    lastUpdated()
  }

  static mapping = {
    sort "lastUpdated":"desc"
  }

  static belongsTo = [author:User]

  String title
  String summary
  String filename
  Date dateCreated
  Date lastUpdated
}
```

And finally, add the `filename` to the `Entry` object in the `save` closure. Listing 10 shows the complete `save` closure:

Listing 10. Storing the filename in the Entry

```
def save = {
    def entryInstance = new Entry(params)
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    entryInstance.author = User.get(session.user.id)

    //handle uploaded file
    def uploadedFile = request.getFile('payload')
    if(!uploadedFile.empty){
      println "Class: ${uploadedFile.class}"
      println "Name: ${uploadedFile.name}"
      println "OriginalFileName: ${uploadedFile.originalFilename}"
      println "Size: ${uploadedFile.size}"
      println "ContentType: ${uploadedFile.contentType}"

      def webRootDir = servletContext.getRealPath("/")
      def userDir = new File(webRootDir, "/payload/${session.user.login}")
      userDir.mkdirs()
      uploadedFile.transferTo( new File( userDir, uploadedFile.originalFilename))
      entryInstance.filename = uploadedFile.originalFilename
    }

    if(!entryInstance.hasErrors() && entryInstance.save()) {
        flash.message = "Entry ${entryInstance.id} created"
        redirect(action:show,id:entryInstance.id)
    }
    else {
        render(view:'create',model:[entryInstance:entryInstance])
    }
}
```

An alternate strategy to saving the uploaded files to the file system is to store them directly in the database. If you create a `byte[]` field named `payload` in `Entry`, you can completely bypass all of the custom code you added to the `save` closure. But if you do that, you'll miss out on all of the fun in the next section.

Displaying the uploaded file

What's the point of uploading a file if you don't display it somewhere? Open grails-app/views/entry/_entry.gsp and add the code in Listing 11:

Listing 11. GSP code to display the uploaded image

```
<div class="entry">
  <span class="entry-date">
      <g:longDate>${entryInstance.lastUpdated}</g:longDate> : ${entryInstance.author}
  </span>
  <h2><g:link action="show" id="${entryInstance.id}">${entryInstance.title}</g:link></h2>
  <p>${entryInstance.summary}</p>

  <g:if test="${entryInstance.filename}">
    <p>
      <img src="${createLinkTo(dir:'payload/'+entryInstance.author.login,
                               file:''+entryInstance.filename)}"
           alt="${entryInstance.filename}"
           title="${entryInstance.filename}" />
    </p>
  </g:if>
</div>
```

Because uploading the file is optional, I wrap the output in a `<g:if>` block. If the

© Copyright IBM Corporation 2009. All rights reserved.

`entryInstance.filename` field is populated, I display the results in an `<img>` tag.

Figure 2 shows the new list, proudly displaying the uploaded Grails logo:

Figure 2. Displaying the uploaded image

But what if the user uploads something other than an image? Rather than putting more logic in the GSP, this feels like the perfect place for a custom TagLib.

Creating a TagLib

Blogito already has two TagLibs in grails-app/taglib: DateTagLib.groovy and LoginTagLib.groovy. You can define as many custom tags in a single TagLib as you'd like, but this time I'd recommend creating a new one just to keep the tags semantically grouped together. Type `grails create-tag-lib Entry` at the

© Copyright IBM Corporation 2009. All rights reserved.

command prompt and add the code in Listing 12:

Listing 12. Creating the displayFile tag

```
class EntryTagLib {

  def displayFile = {attrs, body->
    def user = attrs["user"]
    def filename = attrs["filename"]

    if(filename){
      def extension = filename.split("\\.")[-1]
      def userDir = "payload/${user}"

      switch(extension.toUpperCase()){
        case ["JPG", "PNG", "GIF"]:
             def html = """
             <p>
               <img src="${createLinkTo(dir:''+userDir,
                                         file:''+filename)}"
                    alt="${filename}"
                    title="${filename}" />
             </p>
             """

             out << html
             break

        case "HTML":
             out << "p>html</p>"
             break
        default:
             out << "<p>file</p>"
             break
      }
    }else{
      out << "<!-- no file -->"
    }
  }

}
```

As you'll see in a moment, this code creates a `<g:displayFile>` tag that expects two attributes: `user` and `filename`. If the `filename` attribute is populated, the file extension is pulled off and converted to uppercase.

`Switch` statements in Groovy offer significantly more flexibility than their Java counterpart. For starters, you can switch on `String`s. (The Java language can only switch on an `int`.) Even more impressively, your `case` can specify a `List` of conditions as well as a single condition.

With this TagLib in place, the _entry.gsp partial template can be dramatically simplified, as shown in Listing 13:

Listing 13. The simplified partial template

```
<div class="entry">
```


© Copyright IBM Corporation 2009. All rights reserved.

```
  <span class="entry-date">
      <g:longDate>${entryInstance.lastUpdated}</g:longDate> : ${entryInstance.author}
  </span>
  <h2><g:link action="show" id="${entryInstance.id}">${entryInstance.title}</g:link></h2>
  <p>${entryInstance.summary}</p>

  <g:displayFile filename="${entryInstance.filename}"
                 user="${entryInstance.author.login}" />

</div>
```

Restart Grails and upload the Grails logo once again. Before you add support for other file types, you should make sure that your TagLib refactoring didn't break existing functionality.

Now that you're sure that uploading an image works, adding support for other file types is a matter of implementing the appropriate `case`s in the `switch` block. Listing 14 demonstrates how to handle an uploaded HTML file, as well as simply creating a link to download the file for the default case:

Listing 14. The full switch/case block

```
class EntryTagLib {

  def displayFile = {attrs, body->
    def user = attrs["user"]
    def filename = attrs["filename"]

    if(filename){
      def extension = filename.split("\\.")[-1]
      def userDir = "payload/${user}"

      switch(extension.toUpperCase()){
        case ["JPG", "PNG", "GIF"]:
             //SNIP
             break

        case "HTML":
             def webRootDir = servletContext.getRealPath("/")
             out << new File(webRootDir+"/"+userDir, filename).text
             break
        default:
             def html = """
             <p>
               <a href="${createLinkTo(dir:''+userDir,
                                       file:''+filename)}">${filename}</a>
             </p>
             """
             out << html
             break
      }
    }else{
      out << "<!-- no file -->"
    }
  }

}
```

Create two new text files to exercise this new behavior: one named test.html and a

© Copyright IBM Corporation 2009. All rights reserved.

second named noextension. Add the content in Listing 15 to the appropriate file, upload it, and verify that the TagLib displays each as expected:

Listing 15. Two sample files to upload

```
//test.html
<p>
This is some <b>test</b> HTML.
</p>

<p>
Here is a link to the <a href="http://grails.org">Grails</a> homepage.
</p>

<p>
And here is a link to the
<img src="http://grails.org/images/grails-logo.png">Grails Logo</img>.
</p>



//noextension
This file doesn't have an extension.
```

Your Web browser should look like Figure 3:

Figure 3. Showing all three types of uploaded files

© Copyright IBM Corporation 2009. All rights reserved.

Adding an Atom feed

By this point, you should see a distinct pattern forming. For each new feature you add to your Grails application, you'll most likely touch the model, the view, and the controller. You might also add a partial template or a TagLib along the way for good measure.

Adding an Atom feed to Blogito follows this pattern. It doesn't require you to change the model, but you'll end up doing everything else. You'll:

© Copyright IBM Corporation 2009. All rights reserved.

1. Add a closure to the `Entry` controller to handle the Atom request.
2. Create a new GSP page to render the results as a well-formed Atom document.
3. Create a new partial template and a new custom tag to speed things along.

You could install a nice Feeds plug-in that adds RSS and Atom capabilities to your Grails application (see Resources), but I think that you'll find that the Atom format is simple enough to tackle on your own. To prove this, you can view the source of an existing Atom feed, or check out the example at the end of the Wikipedia page on Atom (see Resources). You could even read RFC 4287, the IETF specification for the Atom format (see Resources). Or, you can simply keeping reading to see a Grails-specific solution.

To begin, add an `atom` closure to EntryController.groovy, as shown in Listing 16:

Listing 16. Adding an atom closure to EntryController.groovy

```
def atom = {
  if(!params.max) params.max = 10
  def list = Entry.list( params )
  def lastUpdated = list[0].lastUpdated
  [ entryInstanceList:list, lastUpdated:lastUpdated ]
}
```

The only difference between this and the standard `list` closure is the addition of the `lastUpdated` field. Because the list is already sorted by `lastUpdated` (thanks to the `sort "lastUpdated":"desc"` setting in the `static mapping` block in the `Entry` domain class), grabbing this field from the first `Entry` in the list effectively gives you the most recent date.

Next, create grails-app/views/entry/atom.gsp. Add the code in Listing 17:

Listing 17. atom.gsp

```
<% response.setContentType("application/atom+xml")
%><feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">News from Blogito.org</title>
  <link rel="alternate" type="text/html" href="http://blogito.org/"/>
  <link rel="self" type="application/atom+xml" href="http://blogito.org/entry/atom" />
  <updated><g:atomDate>${lastUpdated}</g:atomDate></updated>
  <author><name>Blogito.org</name></author>
  <id>tag:blogito.org,2009-01-01:entry/atom</id>
  <generator uri="http://blogito.org" version="0.1">Hand-rolled Grails code</generator>

  <g:each in="${entryInstanceList}" status="i" var="entryInstance">
<g:render template="atomEntry" bean="${entryInstance}" var="entryInstance" />
  </g:each>
```


© Copyright IBM Corporation 2009. All rights reserved.

```
</feed>
```

As you can see, the first thing you do is set the MIME type to application/atom+xml. After that, you supply some basic metadata about the feed: updated, author, generator, and so on.

If you want to avoid the hardcoded blogito.org throughout the feed, you can have the atom closure grab request.serverName, assign it to a variable, and return it in the response hashmap along with entryInstanceList and lastUpdated. To be fully dynamic, you can use request.scheme to return http, and request.serverPort to return 80, as well. (The only place you want to avoid using the request.serverName variable is in the id, as I'll discuss in just a moment.)

It's common for an Atom feed to supply links in several different formats. As you can tell by the type attribute, this feed offers two: one HTML link, and one back to itself in the Atom format. The self link is especially useful; if you ever end up with an Atom document that you didn't download yourself, this is your trail of breadcrumbs back to the canonical source.

The id field is the unique identifier for your Atom feed, distinct from the URI or current location where it can be downloaded. (As you just learned, the <link> element supplies the feed's current source.) In this example, I use the technique outlined by Mark Pilgrim to generate a unique, permanent ID string: combine your domain name, the first date that your feed went into service, and the rest of the URI. (See Resources for more information.)

The individual pieces of the id aren't nearly as important as the uniqueness of the whole. Be sure that this id doesn't change over time by inadvertently passing in variables from the controller — in the case of the feed id, you want it to be both unique and invariant. Even if the *address* of your server changes, the feed id should remain unchanged if the content remains unchanged.

The updated field needs to be in a specific format — 2003-12-13T18:30:02Z, or RFC 3339 to be exact. (See Resources for details.) Add an atomDate closure to the existing grails-app/taglib/DateTagLib.groovy file, as shown in Listing 18:

Listing 18. Adding the atomDate tag

```
import java.text.SimpleDateFormat

class DateTagLib {
  public static final String INCOMING_DATE_FORMAT = "yyyy-MM-dd hh:mm:ss"
  public static final String ATOM_DATE_FORMAT = "yyyy-MM-dd'T'HH:mm:ss'-07:00'"

  def atomDate = {attrs, body ->
    def b = attrs.body ?: body()
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    def d = new SimpleDateFormat(INCOMING_DATE_FORMAT).parse(b)
    out << new SimpleDateFormat(ATOM_DATE_FORMAT).format(d)
  }

  //SNIP
}
```

To complete the Atom feed, create grails-app/views/entry/_atomEntry.gsp and add the code in Listing 19:

Listing 19. The _atomEntry.gsp partial template

```
<entry xmlns='http://www.w3.org/2005/Atom'>
  <author>
    <name>${entryInstance.author.name}</name>
  </author>
  <published><g:atomDate>${entryInstance.dateCreated}</g:atomDate></published>
  <updated><g:atomDate>${entryInstance.lastUpdated}</g:atomDate></updated>
  <link href="http://blogito.org/blog/${entryInstance.author.login}/
    ${entryInstance.title.encodeAsUnderscore()}" rel="alternate"
    title="${entryInstance.title}" type="text/html" />
  <id>tag:blogito.org,2009:/blog/${entryInstance.author.login}/
    ${entryInstance.title.encodeAsUnderscore()}</id>
  <title type="text">${entryInstance.title}</title>
  <content type="xhtml">
    <div xmlns="http://www.w3.org/1999/xhtml">
      ${entryInstance.summary}
    </div>
  </content>
</entry>
```

The last thing you need to do is open up the Atom feed to unauthenticated users. Tweak the `beforeInterceptor` in EntryController.groovy, as shown in Listing 20:

Listing 20. Opening the Atom feed to unauthenticated users

```
class EntryController {

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

  //SNIP
}
```

Restart Grails, and a visit to http://localhost:9090/blogito/entry/atom should yield a well-formed Atom feed, as shown in Listing 21:

Listing 21. The well-formed Atom feed

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">News from Blogito.org</title>
  <link rel="alternate" type="text/html" href="http://blogito.org/"/>
  <link rel="self" type="application/atom+xml" href="http://blogito.org/entry/atom" />
  <updated>2009-04-20T00:03:34-07:00</updated>
  <author><name>Blogito.org</name></author>
  <id>tag:blogito.org,2009-01-01:entry/atom</id>
```


© Copyright IBM Corporation 2009. All rights reserved.

```
  <generator uri="http://blogito.org" version="0.1">Hand-rolled Grails code</generator>

<entry xmlns='http://www.w3.org/2005/Atom'>
  <author>
    <name>Jane Smith</name>
  </author>
  <published>2009-04-20T00:03:34-07:00</published>
  <updated>2009-04-20T00:03:34-07:00</updated>
  <link href="http://blogito.org/blog/jsmith/Testing_with_Groovy" rel="alternate"
    title="Testing with Groovy" type="text/html" />
  <id>tag:blogito.org,2009:/blog/jsmith/Testing_with_Groovy</id>
  <title type="text">Testing with Groovy</title>
  <content type="xhtml">
    <div xmlns="http://www.w3.org/1999/xhtml">
      See Practically Groovy
    </div>
  </content>

<!-- SNIP -->

</entry>
</feed>
```

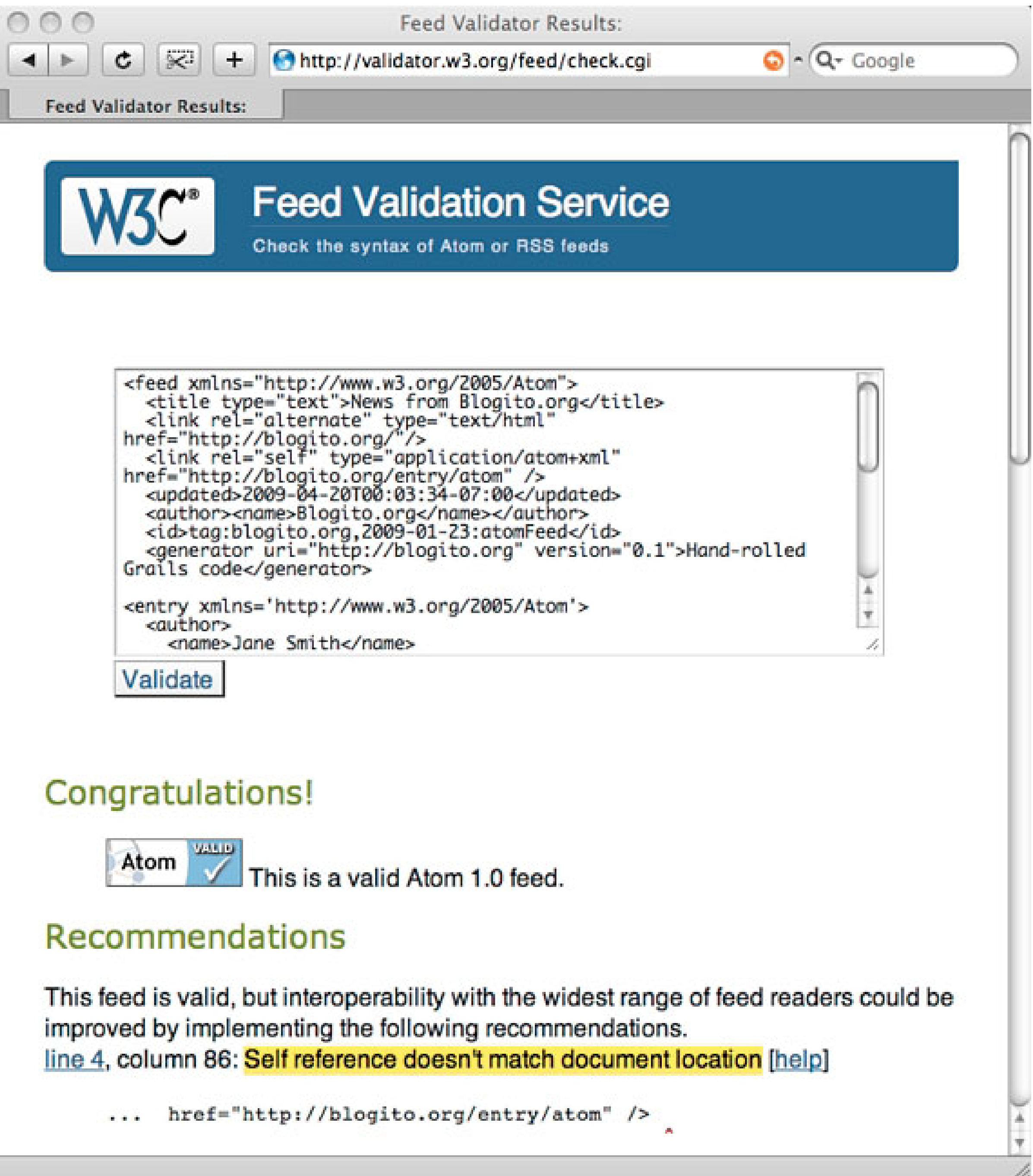
Although the semantics of Atom might be new to you, the mechanics of producing an Atom feed using Grails should be pretty straightforward.

Validating the Atom feed

To verify that the feed is, indeed, well-formed Atom, visit the W3C's online Feed Validator (see Resources). If your feed is at a publicly accessible URI, you can paste it on the home page and click **Check**. Your Atom feed is running at localhost, so click on **Validate by Direct Input** and paste in the output from your feed. The results are shown in Figure 4:

Figure 4. The W3C validator

© Copyright IBM Corporation 2009. All rights reserved.



Other than a warning that the self link isn't available at the URI provided — which is obviously the case — your Atom feed should be deemed valid and ready for production.

Adding the feed icon

The icing on the cake is adding a link for the feed to the header. You can download

© Copyright IBM Corporation 2009. All rights reserved.

the ubiquitous feed icon from many places on the Web; it is released under the open source Mozilla license (see Resources).

Copy the file to web-app/images and then tweak grails-app/views/layouts/_header.gsp as shown in Listing 22:

Listing 22. Adding the feed icon to the header

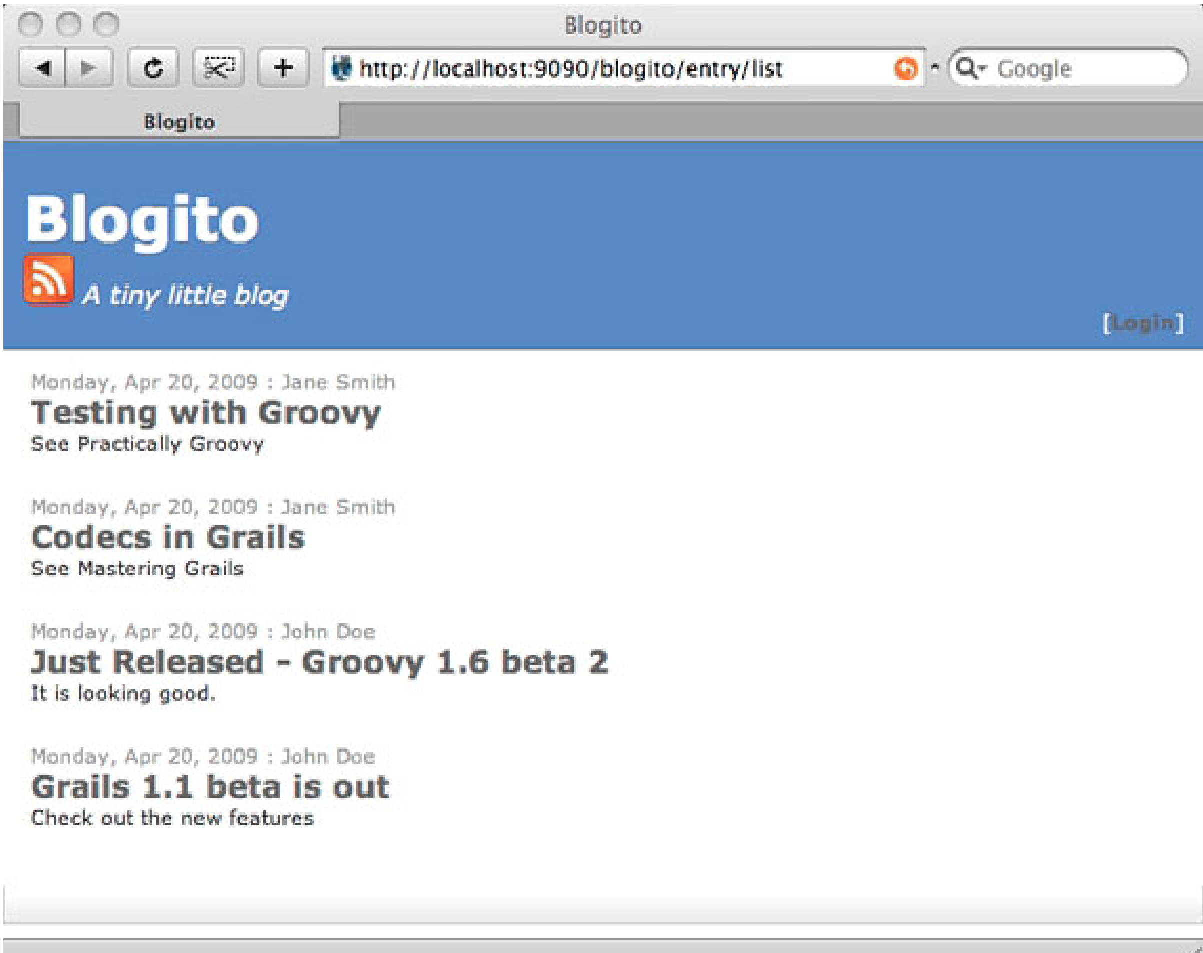
```
<div id="header">
  <p><g:link class="header-main" controller="entry">Blogito</g:link></p>
  <p class="header-sub">
    <g:link controller="entry" action="atom">
    <img src="${createLinkTo(
        dir:'images',file:'feed-icon-28x28.png')}" alt="Subscribe" title="Subscribe"/>
    </g:link>
    A tiny little blog
  </p>

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

The results should be a home page that looks like Figure 5:

Figure 5. Blogito home page with the feed icon

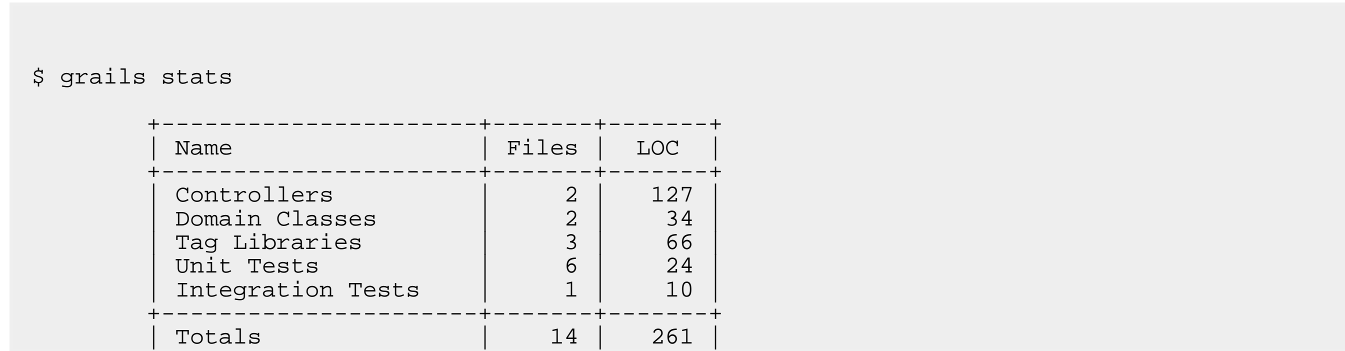
© Copyright IBM Corporation 2009. All rights reserved.



Conclusion

In this article, you added file-upload capabilities as well as an Atom syndication feed. And there you have it: Blogito is now a *working* tiny little blog server. Just how tiny is it? Two domain classes, two controllers, and just over 250 lines of code. Type `grails stats` to verify this. Listing 23 shows the results:

Listing 23. The size of Blogito



```
$ grails stats

        +----------------------+-------+-------+
        | Name                 | Files |  LOC  |
        +----------------------+-------+-------+
        | Controllers          |     2 |   127 |
        | Domain Classes       |     2 |    34 |
        | Tag Libraries        |     3 |    66 |
        | Unit Tests           |     6 |    24 |
        | Integration Tests    |     1 |    10 |
        +----------------------+-------+-------+
        | Totals               |    14 |   261 |
```


© Copyright IBM Corporation 2009. All rights reserved.

```
          +----------------------+-------+-------+
```

Although this exercise stretched out over four articles, the reality is that it represents about a single day's worth of development effort once you have a solid working knowledge of Grails.

I hope that you've enjoyed piecing Blogito together. Next time, you'll add support for comments, tags, and more through the corresponding plug-ins. And in subsequent installments, I'll explore the Grails plug-in ecosystem further with you. Until then, have fun mastering Grails.

© Copyright IBM Corporation 2009. All rights reserved.

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, where you'll find documentation on the `<select>` and `<uploadForm>` tags.
- The Atom Syndication Format: IETF RFC 4287 is the Atom specification.
- Atom (standard): You'll find an example Atom feed at the end of this Wikipedia article.
- Atom Syndication Format: Check out these developerWorks resources for lots more information on Atom.
- "How to make a good ID in Atom" (Mark Pilgrim, dive into mark, May 2004): Learn more about Pilgrim's technique for generating a unique, permanent Atom ID.
- Date and Time on the Internet: Timestamps: RFC 3339 defines the date-and-time format that this article's Atom `id`-generation technique uses.
- W3C Feed Validation Service: Use this service to verify that an Atom feed is well-formed.
- *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.
- Feeds Plugin: A Grails plug-in that renders RSS/Atom feeds and iTunes-compatible podcasts.

© Copyright IBM Corporation 2009. All rights reserved.

- Feed icon: You can pick up the standard feed icon at the Feed Icon site or from Wikimedia Commons.
- Blogito: You can download the completed Blogito application.

Discuss

- Get involved in the My 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.

© Copyright IBM Corporation 2009. All rights reserved.