Extracted from:

Grails

A Quick-Start Guide

This PDF file contains pages extracted from Grails, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Programmers

Grails

A Quick-Start Guide

Dave Klein

Edited by Colleen Toporek

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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://www.pragprog.com

Copyright © 2009 Dave Klein.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-46-8

ISBN-13: 978-1-934356-46-3

Printed on acid-free paper.

B6.0 printing, July 28, 2009

Version: 2009-8-7

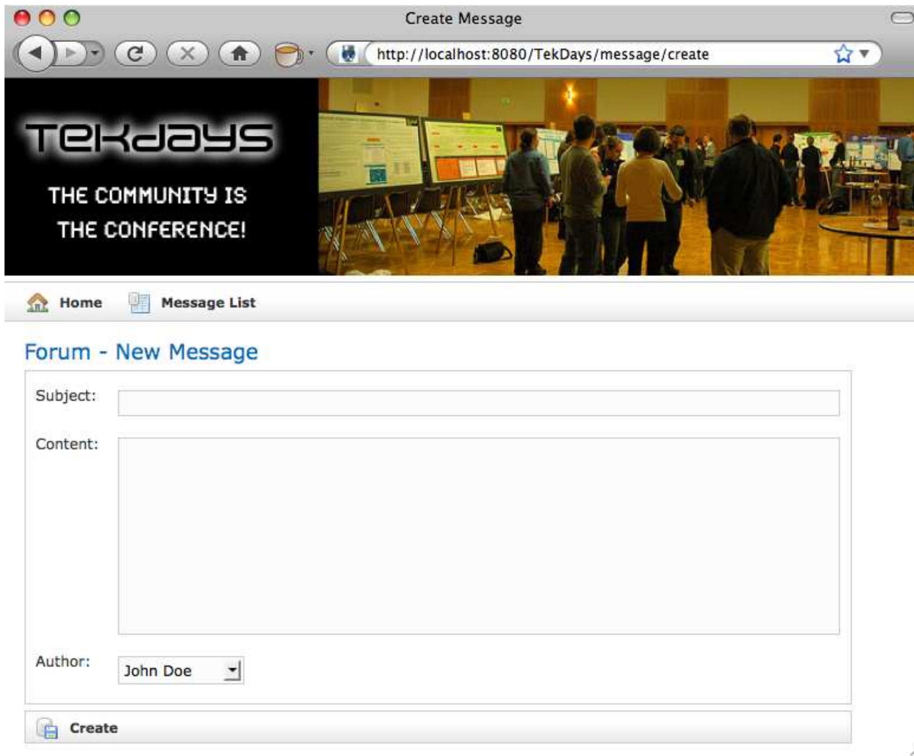

Figure 7.2: Create View 2.0 (so to speak)

```
        ${fieldValue(bean:messageInstance, field:'content')}
      </textarea>
    </td>
</tr>
```

Figure 7.2, shows our new create view. Much better. Next up: cleaning up the list and show views.

7.2 Of Templates and Ajax

On second thought, instead of cleaning up the list and show views, let's just set them aside and create a new view that will replace them both. To do that, we'll take advantage of Grails' GSP templates.

GSP templates are simply chunks of GSP code in a file that begins with an underscore (_likethis.gsp). They provide an easy way to share common

code across multiple pages. You can include a GSP template in a GSP page with the <*g:render*> tag, like this:

```
&lt;g:render template="someTemplate" /&gt;
```

This line would render a template called _someTemplate.gsp in the same directory as the page that it is being called from. To render templates from a different directory, add the path before the name of the template. We never include the "_" at the beginning of the template name in the <*g:render*> tag.

Another popular use for GSP templates is rendering the response to Ajax calls; that's what we're after here. Before we get too much further, let me lay out the plan. What we want is a single page with a list of messages in the upper section, and fields for viewing a single message in the lower section. When a user selects a message in the list, that message's values will display in the fields below, without reloading the rest of the page. Pretty cool, huh? Our customer sure thought so (if I do say so myself). Now let's see how easy this can be with Grails.

To get started, let's create TekDays/grails-app/views/message/ajaxList.gsp. As a shortcut, just copy TekDays/grails-app/views/message/list.gsp, and remove most of it. Keep the <*html*> and <*head*> (with contents), and in the <*body*>, keep the first <*div*>. You should end up with something that looks like this:

Download forum/TekDays/grails-app/views/message/ajaxList.gsp

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <meta name="layout" content="main" />
    <title>Messages</title>
  </head>
  <body>
    <div class="nav">
      <span class="menuButton">
        <a class="home" href="${resource(dir:'')}">Home</a>
      </span>
      <span class="menuButton">
        <g:link class="create" action="create"
            params='[eventId:"${event?.id}"]'>New Message
        </g:link>
      </span>
    </div>


  </body>
</html>
```

We kept the <*head*> section from the list because it contains a couple of <*meta*> tags that we need. Since this new view is going to replace the list view, it makes sense to keep the same button bar. Other than that we only kept the basic page structure tags.

To flesh out the body of our new view, add the following code right after that </*div*> tag.

Download forum/TekDays/grails-app/views/message/ajaxList.gsp

```
<div class="body">
  <h1>${event?.name} - Forum Messages</h1>
  <div id="messageList">
    <g:each in="${messageInstanceList}" var="messageInstance">
    </g:each>
  </div>
  <h3>Message Details</h3>
  <div id="details">
  </div>
</div>
```

First we added a <*div*> with class="body" to be consistent with the other pages in our application, and then an <*h1*> tag similar to the one on the create view, using the TekEvent instance that will be passed in from the controller. Then we added a <*div*>, with an id of messageList, to hold the list of messages. We have a style rule in main.css for this id that will provide scrolling if our list gets that long. (See Appendix A, on page 203.) Inside this <*div*> we have a <*g:each*> tag, which will iterate over the messageInstanceList. Whatever we put in the body of that tag will be displayed once for each element in the list. We'll talk about what to put there shortly.

Below the list<*div*>, we added an <*h3*> tag to serve as a heading to the message detail portion of the page. Finally, we added a <*div*> with an id of details. This is where the message detail template that we are about to create will be rendered.

Creating the Template

Now we need to create the template that will display an individual message. This time, just create a blank file called _details.gsp in the TekDays/grails-app/views/message directory. We'll borrow the <*div*>, <*table*>, and three <*tr*> tags from TekDays/grails-app/views/messages/show.gsp. (The three <*tr*> tags are for the subject, content and author properties.) Since this file's code will be inserted into another page, it doesn't need its own <*html*> or <*head*> tags.

CLICK HERE to purchase this book now.

Download forum/TekDays/grails-app/views/message/_details.gsp

```
<div class="dialog">
  <table>
    <tr class="prop">
      <td valign="top" class="name">Subject:</td>
      <td valign="top" class="value">
        ${fieldValue(bean:messageInstance, field:'subject')}
      </td>
    </tr>
    <tr class="prop">
      <td valign="top" class="name">Content:</td>
      <td valign="top" class="value">
        ${fieldValue(bean:messageInstance, field:'content')}
      </td>
    </tr>
    <tr class="prop">
      <td valign="top" class="name">Author:</td>
      <td valign="top" class="value">
        <g:link controller="tekUser" action="show"
            id="${messageInstance?.author?.id}">
          ${messageInstance?.author?.encodeAsHTML()}
        </g:link>
      </td>
    </tr>
  </table>
  <div class="buttons">
    <span class="menuButton">
      <g:link class="create" action="reply" id="${messageInstance?.id}">
        Reply
      </g:link>
    </span>
  </div>
</div>
```

You may have noticed that we also added a Reply "button" at the bottom of the template. This is actually a *<g:link>* that will be styled to look like a button. The *<g:link>* will call the reply action—which we still need to create. Don't let me forget to come back to that.

Looking at the code for our template, we can see that the only data element that it will need is a Message instance called (believe it or not) messageInstance. This is important to note, because when a template is rendered, the data it requires needs to be passed to it. A template cannot automatically see the data elements of the page that renders it. We'll look at how to provide the data to the template in the next section as we see how to render our template in response to an Ajax call.

Click Here to purchase this book now.

Ajax in Grails

Grails includes several Ajax tags, which we can use to call a controller action and update a page element with the results. That's exactly what we need to do, but before we do it, let's discuss a bit about the way that Grails Ajax tags work.

Grails supports a variety of popular Javascript libraries with regard to its Ajax tags.[1] In order to use these tags, we need to tell Grails which library we are using. We do this with the <*g:javascript*> tag and its libary attribute. This tag is placed in the <*head*> section of a page. Let's go back to TekDays/grails-app/views/message/ajaxList.gsp and add the following line to the <*head*>:

Download forum/TekDays/grails-app/views/message/ajaxList.gsp

```
<g:javascript library="prototype" />
```

Now we can use one of Grails' Ajax tags and it will adapt to use the Prototype library.[2] The tag we're going to use is <*g:remoteLink*>.

Let's see how this looks in our code, and then we'll discuss what it's doing. In TekDays/grails-app/views/message/ajaxList.gsp, add the following code to the <*g:each*> body in our list<*div*>.

Download forum/TekDays/grails-app/views/message/ajaxList.gsp

```
        <g:each in="${messageInstanceList}" var="messageInstance">
▶         <g:remoteLink action="showDetail" id="${messageInstance?.id}"
▶             update="details">
▶           ${messageInstance.author.fullName} - ${messageInstance.subject}
▶         </g:remoteLink>
        </g:each>
```

The <*g:remoteLink*> tag can take controller, action, and id attributes. If the controller attribute is not provided, then the controller that rendered the current page will be used by default. Since the ajaxList view will be rendered by the MessageController, we don't need to specify it here. We did give it an action attribute, which points to an action (which we will create next in the MessageController). Then for the id, we use the messageInstance variable from the <*g:each*>. The final attribute that we set on the <*g:remoteLink*> tag is update. This attribute contains the id

1. See the Grails website for a list of supported libraries: http://www.grails.org/Ajax.
2. Grails handles any differences that might exist in the way different Javascript libraries handle the tasks involved in the Ajax tags; the behavior of these tags is the same regardless of which of the supported libraries we use.

CLICK HERE to purchase this book now.

of the HTML element on this page that will be updated with the result of the action—in this case, details.

For the body of the <*g:remoteLink*> we used the messageInstance variable to build a string containing the name of the message's author and the subject of the message. We'll see how this looks shortly, but first we have to create the showDetail action. Open TekDays/grails-app/controllers/MessageController.groovy and add the following action:

Download forum/TekDays/grails-app/controllers/MessageController.groovy

```
def showDetail = {
    def messageInstance = Message.get(params.id)
    if (messageInstance) {
        render(template:"details", model:[messageInstance:messageInstance])
    }
    else {
        render "No message found with id: ${params.id}"
    }
}
```

This action expects the params to contain an id value. The first thing we do is define a messageInstance variable, and retrieve a Message using the id value in the params. If we have a valid instance, we call the render() method and pass it the name of a template ("details") and a model, which is a Map. The model parameter is used to provide the data that the template will need. In this case, we only have one object in the model, but we can include as many objects as our template needs. The render() method will merge our template with the data in the messageInstance bean and return the results as HTML. This HTML will then replace the contents of the <*div*> on our page.

Now there's just one thing left to do before we can marvel at our handiwork: We need to provide a way to reach our new view. If we added an action to the MessageController called ajaxList, it would automatically render our new view, but it would just be a copy of the list action, and that wouldn't be very *DRY*. So we'll use a different approach. The same render() method that we just used for our details template can be used to render an entire view. Let's go back to the list action in TekDays/grails-app/controllers/MessageController.groovy and modify the last line (the line that returns the Map).

Download forum.1/TekDays/grails-app/controllers/MessageController.groovy

```
def list = {
    params.max = Math.min( params.max ? params.max.toInteger() : 10,  100)
    def list
    def count
```

```
    def event = TekEvent.get(params.id)
    if (event){
        list = Message.findAllByEvent(event, params)
        count = Message.countByEvent(event)
    }
    else{
        list = Message.list(params)
        count = Message.count()
    }
    render(view:'ajaxList',
           model:[messageInstanceList: list, messageInstanceTotal: count,
                  event: event])
}
```

This time, we pass a view parameter instead of a template. We set that parameter to our new page, and then we pass the existing Map as the model. Now when the list action is called (for example, when we navigate to http://localhost:8080/TekDays/message/list), our new view will be rendered.

Wait a minute. We still need to add a reply action to the MessageController. OK. The reply action will be very similar to the create action, except that it will set the parent of the *new*Message to the *current* one. Hopefully, you still have MessageController.groovy open, so you can slip in the following code:

Download forum.1/TekDays/grails-app/controllers/MessageController.groovy

```
def reply = {
    def parent = Message.get(params.id)
    def messageInstance = new Message(parent:parent, event:parent.event,
                                      subject:"RE: $parent.subject")
    render(view:'create', model:['messageInstance':messageInstance])
}
```

In this action, we take the id parameter that is passed in on the link from the ajaxList view and use it to retrieve a Message instance. Then we create a new Message, setting its parent and subject properties based on the retrieved instance. Finally, we use the render() method to render the create view with the messageInstance in the model. This will bring up the create view, which we will now modify to handle this new responsibility.

When the create view is rendered from the reply action, the message will have a parent assigned. We'll change our view slightly and check for the existence of this property. Open TekDays/grails-app/views/message/create.gsp and add the following code right after the <*tbody*> tag:

CLICK HERE to purchase this book now.

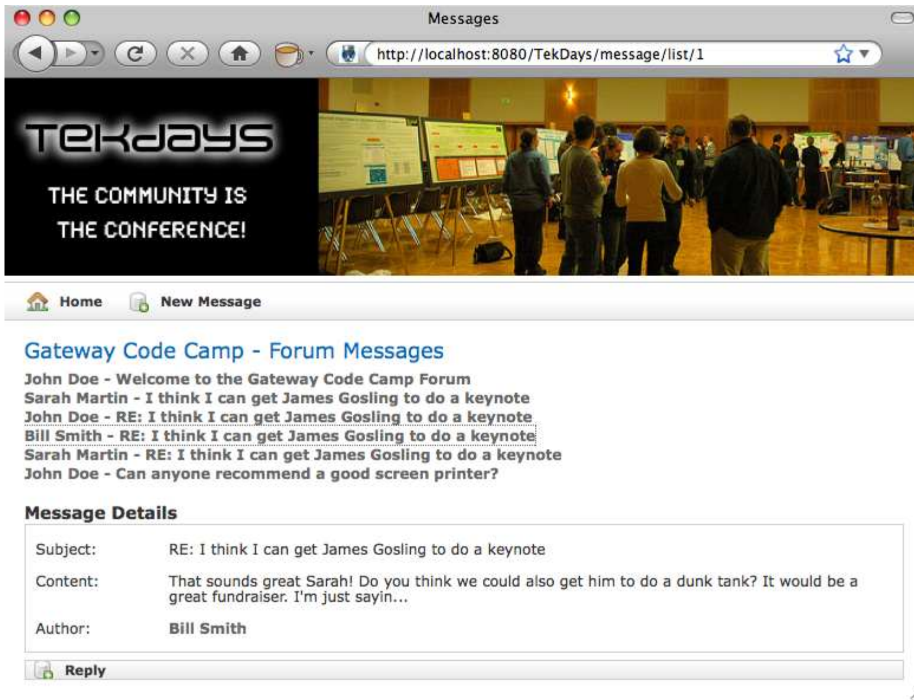

Figure 7.3: Ajax Enabled Message List

Download forum.1/TekDays/grails-app/views/message/create.gsp

```
<g:if test="${messageInstance.parent}">
  <input type="hidden" name="parent.id" value="${messageInstance.parent.id}" />
  <tr class="prop">
      <td valign="top" class="name">
          <label>In Reply to:</label>
      </td>
      <td valign="top" class="value">
          ${messageInstance.parent.author}
      </td>
  </tr>
</g:if>
```

Inside a <*g:if*> block, we added an "In Reply to:" label, and filled in the subject appropriately. We also added a hidden field to store the message.parent value so that it can be passed on to the save action, to complete the link between a reply and its parent.

It's difficult to do justice to this functionality in print, but we'll try.

Click Here to purchase this book now.

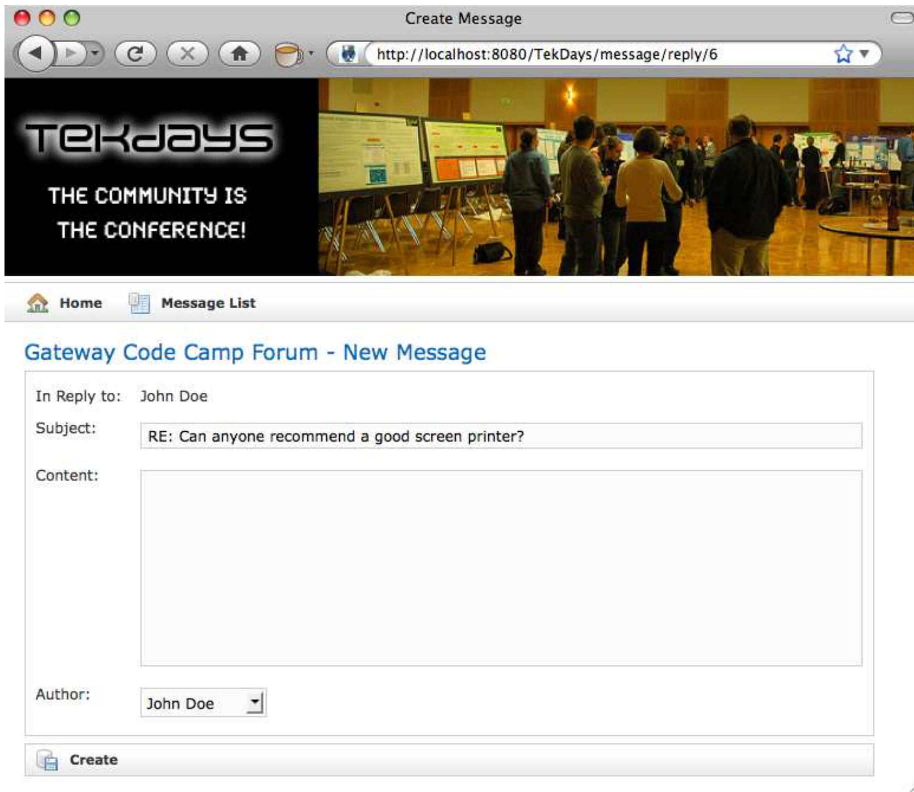

Figure 7.4: Message Create View - Reply

Figure 7.3, on the previous page, shows our new ajaxList view with a message selected, and Figure 7.4, shows the result of clicking on the Reply button for that message. If you've done anything like this before in another Java web framework, you're probably as impressed as I am by how easy it was to do this. I've heard that that sense of awe and amazement wears off after a while. But I'm still waiting.

7.3 Display Message Threads with a Custom Tag

Now we need to add nesting to our message list, in order to visualize the various threads in our forum. We'll do this with a custom GSP tag. If you've ever written custom JSP tags or JSF components, come out from under the table. It's not like that at all. But just to reassure you,

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Grails Quick Start's Home Page
http://pragprog.com/dkgrails
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/dkgrails.

Contact Us

Online Orders: www.pragprog.com/catalog
Customer Service: support@pragprog.com
Non-English Versions: translations@pragprog.com
Pragmatic Teaching: academic@pragprog.com
Author Proposals: proposals@pragprog.com
Contact us: 1-800-699-PROG (+1 919 847 3884)