

From Technologies to Solutions

Apache MyFaces 1.2 Web Application Development

Building next-generation web applications with JSF and Facelets

Bart Kummel

PACKT PUBLISHING

www.it-ebooks.info

Apache MyFaces 1.2

Web Application Development

Building next-generation web applications with JSF and Facelets

Bart Kummel



BIRMINGHAM - MUMBAI

www.it-ebooks.info

Apache MyFaces 1.2
Web Application Development

Copyright © 2010 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2010

Production Reference: 1240210

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847193-25-4

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

www.it-ebooks.info

Credits

Author
Bart Kummel

Reviewers
Cagatay Civici
Hazem Saleh
Matthias Weßendorf

Acquisition Editor
Rashmi Phadnis

Development Editor
Darshana D. Shinde

Technical Editors
Aliasgar Kheriwala
Conrad Sardinha

Copy Editor
Sneha Kulkarni

Indexer
Hemangini Bari

Editorial Team Leader
Mithun Sehgal

Project Team Leader
Lata Basantani

Project Coordinator
Joel Goveya

Proofreader
Dirk Manuel

Production Coordinator
Adline Swetha Jesuthas

Cover Work
Adline Swetha Jesuthas

www.it-ebooks.info

About the Author

Bart Kummel is an experienced Java EE developer and architect living in The Netherlands. He studied Electrical Engineering at the Delft University of Technology and graduated with honor from the Hogeschool van Amsterdam, with a specialization in Technical Computer Sciences. After his study, he started as a developer of embedded software for a security systems manufacturer in Amsterdam. After four years of developing embedded software, Bart switched to enterprise software and started at Transfer Solutions B.V., based in Leerdam. Transfer Solutions is a consulting company that specializes in Oracle and Java technology.

As a consultant for Transfer Solutions, Bart gained a lot of experience with Java EE. For different clients, he has fulfilled the roles of developer or architect in small as well as large projects. In those projects, he has worked with various frameworks and standards, including Oracle's Application Development Framework (ADF), Apache MyFaces, EclipseLink, JavaServer Faces (JSF), Java Persistence API (JPA), and Java Messaging Service (JMS). Bart also teaches courses in Object Orientation, UML, Java, Java EE, and ADF, at Transfer Solutions' education department.

Bart published an article on EclipseLink in the Dutch Java Magazine, and presented the use of AJAX capabilities in Oracle's ADF Faces at the ODTUG Kaleidoscope conference in 2007.

www.it-ebooks.info

Acknowledgement

Writing a book is an awful lot of work. People warned me about that, but I wanted to try it anyway. And I learned that it also gives a lot of satisfaction. I wouldn't have missed the experience! Writing this book wasn't possible without the help of a lot of people and I want to thank these people.

First of all, I would like to thank my wife for supporting me while I was writing this book. It really means a lot to me how she gave me space to work on this project and helped me relax when I needed to. I would also like to thank my friends and family for their understanding when I had to cancel a party or leave early. And I would like to thank my employer, Transfer Solutions, for giving me the opportunity to write this book partly during work hours.

Of course, I also have a big 'thank you' for all the Packt Publishing staff that helped in the process. Special thanks to Joel Goveya, who helped me in keeping an eye on the schedule. I also want to mention Gerhard Petracek. He not only did the MyFaces project a big favor by contributing the Extensions Validator project, but he was also of great help when I wrote the chapter on that new subproject. I also owe lots of thanks to the many reviewers who helped me in getting the quality of my writings to a higher level: Cagatay Civici, Anton Gerdessen, Albert Leenders, Kim Mooiweer, Hazem Saleh, Herman Scheltinga, Reginald Sprinkhuizen, Pieter Stek, Peter Vermaat, Matthias Weßendorf, and René van Wijk.

www.it-ebooks.info

About the Reviewers

Cagatay Civici is the PMC member of the open source JSF implementation of Apache MyFaces, and the project leader of the popular PrimeFaces framework. In addition to being a recognized speaker at international conferences such as JSFSummit, JSFDays, and local events, he's an author and technical reviewer of books regarding web application development with Java and JSF. Cagatay is currently working as a consultant and instructor in the UK.

Hazem Saleh has five years of experience in Java EE and open source technologies. He is committed to Apache MyFaces and is the initiator of many components in the MyFaces projects, such as Tomahawk CAPTCHA, Commons ExportActionListener, Media, PasswordStrength, and others. He is the founder of GMaps4JSF (an integration project that integrates Google Maps with Java ServerFaces), and is the co-author of The Definitive Guide to Apache MyFaces and Facelets by Apress. He is now working for IBM Egypt as a staff software engineer, where he is recognized as a subject matter expert in Web 2.0 technologies.

I dedicate my review efforts for the prophet Muhammad from whom I learnt all the good things in my life.

Matthias Weßendorf is a principal software developer at Oracle. He currently works on server-side-push support for ADF Faces and Trinidad 2.0. Matthias also contributes to the open source community, mainly to Apache MyFaces and Apache MyFaces Trinidad. You can follow Matthias on Twitter (@mwessendorf).

www.it-ebooks.info

Table of Contents

Preface 1

Chapter 1: Introduction 9

Introducing Apache MyFaces 9

License, community, and support 10

MyFaces and Sun JSF RI 10

Subprojects of Apache MyFaces 11

Core 11

Tomahawk 12

Sandbox 13

Trinidad 13

Tobago 14

Orchestra 14

Portlet Bridge 14

Extensions Validator 14

Summary 15

Chapter 2: Getting Started 17

Configuring the development environment 17

Configuring Eclipse 18

Installing extra plugins 18

Installing the libraries 20

Preparing a new project 21

Configuring JDeveloper 25

Installing the libraries 25

Preparing a new project 28

Creating a new project using Maven 33

Application server and configuration files 34

The web.xml configuration file 35

The faces-config.xml configuration file 37

Settings for specific application servers 40

www.it-ebooks.info

Settings for MyFaces Core on GlassFish 40
Other application servers 41
Introduction to the example case 41
Summary 42
Chapter 3: Facelets 43
Why Facelets? 43
Content inverweaving 44
Templating 44
Don't Repeat Yourself (DRY) 45
Expanding the Expression Language 45
Summarizing the benefits of Facelets 45
Setting up a Facelets project 46
Preparing web.xml 46
Preparing faces-config.xml 47
Creating a test page 47
Debugging easily with Facelets 49
Templating with Facelets 51
Creating a template 51
Using the template 52
Using comments in Facelets page definitions 55
Are Facelets files XHTML? 57
Creating and using composition components 58
Creating a tag library 58
Creating the composition component itself 60
Identifying redundancies 60
Creating a skeleton for the composition component 61
Defining the actual composition component 62
Adding validators without violating the DRY principle 63
Putting it all together 64
Using the composition component 66
Using static functions 67
Using inline texts 69
Facelets tags overview 70
<ui:component> tag 70
<ui:composition> tag 71
<ui:debug> tag 71
<ui:decorate> tag 72
<ui:define> tag 72
<ui:fragment> tag 73
<ui:include> tag 74
<ui:insert> tag 74
<ui:param> tag 74

www.it-ebooks.info

<ui:remove> tag 75
<ui:repeat> tag 75
Summary 75
Chapter 4: Tomahawk 77
Setting up Tomahawk 78
Downloading Tomahawk 78
Configuring web.xml 79
Resolving dependencies 80
Using extended versions of standard components 81
Extended components 83
<t:aliasBean> and <t:aliasBeanScope> components 83
<t:buffer> 83
<t:captcha> component 83
Creating basic data tables 86
Setting up a data table 87
Adding columns to the table 88
Using pagination 89
Changing the looks of the data table 91
Styling the data table itself 92
Styling the data scroller 93
Looking at the result 94
Using advanced data table features 94
Sorting 94
Improving the sort arrows 97
Showing details inline 98
Linking to an edit form 100
Grouping rows 103
Newspaper columns 104
Uploading files 105
Working with dates and calendars 108
Using a pop-up calendar 109
Localizing the pop-up calendar 111
Using an inline calendar 112
Using the calendar in a form 113
Extra validators 115
Validating equality 115
Validating e-mail addresses 115
Validating credit card numbers 116
Validating against a pattern 116
Summary 117

www.it-ebooks.info

Chapter 5: Trinidad—the Basics 119
Setting up Trinidad 120
Configuring the web.xml file 120
Configuring the faces-config.xml file 121
Configuring the trinidad-config.xml file 122
Adapting our template 122
Creating data tables 123
Adding columns 124
Using pagination 126
Displaying inline details 127
Configuring banding and grid lines 128
Using row selection 129
Creating input and edit forms 132
Exploring the common features of input components 132
Using automatic label rendering 132
Using error message support and the required indicator 133
Using auto submit 133
Creating plain text input fields 134
Using the <tr:inputText> component in a composition component 135
Creating date input fields 136
Converting dates 137
Validating dates 138
Creating the ultimate date input composition component 140
Creating selection lists 141
Adding list contents 142
Optional empty selection for single selection lists 142
Options for all selection components 143
Checkboxes and radio buttons 143
Listboxes 144
Choice list 144
Shuttle 144
Ordering shuttle 145
Creating a universal composition component for selections 146
Creating fields for numerical input 149
Adding conversion to a field 149
Adding validation to a field 150
Adding a spin box to an input field 150
File uploading 151
Meeting the prerequisites 151
Using the file upload component 152
Creating and using a file upload composition component 152
Saving the file in the backing bean 153
Configuring file upload limits 155
Setting upload limits in web.xml 156
Setting upload limits in trinidad-config.xml 156

www.it-ebooks.info

Using Trinidad's hierarchical navigation features **158**
Configuring the hierarchy 158
Creating navigation panes 160
Creating breadcrumbs 161
Creating a hierarchical menu 162
Creating layouts for our pages **163**
Using a border layout 163
Layout methods 164
Using group layout 165
Using a horizontal layout 166
Creating layouts for input forms 167
Grouping components 167
Label and message 168
Footer facet 168
Creating an accordion 169
Creating a tabbed panel 170
Creating a choice panel 171
Creating a radio panel 172
Displaying boxes 172
Displaying tips 174
Using a header panel 174
Using pop ups 175
Creating button bars 176
Using caption groups 177
Creating bulleted lists 179
Lay out a page by using the panel page component 180
Using the page header panel 182
Summary **183**
Chapter 6: Advanced Trinidad **185**
Data visualization **185**
Creating the data model 186
Understanding the terminology 186
Implementing a minimal data model 186
Calculating the values 188
Initializing the data model 189
Adding a graph to a page 190
Changing data display 191
Changing the looks 191
Chart types 192
Some final thoughts on data visualization 197
Passing on data with page flows **198**
Using AJAX and Partial Page Rendering **202**

www.it-ebooks.info

Comparing full submit and partial submit 202
Using the autoSubmit and partialTriggers attributes 203
Working with partialTriggers and naming containers 207
Creating a status indicator 209
Using the addPartialTarget() method 209
Dynamically hiding or showing components 211
Polling 212
Exploring the possibilities of PPR 213
Creating dialogs 214
Building a dialog 215
Creating the backing bean for the dialog 216
Using an alternative way of returning values 218
Calling the dialog 218
Receiving the dialog's output 219
Using inputListOfValues as an easier alternative 220
Using lightweight dialogs 221
Client-side validation and conversion 221
Defining the data structure 222
Creating the converter 224
Enabling client-side capabilities 225
Implementing the client-side code 227
Creating the validator 228
Enabling client-side capabilities 229
Implementing the client-side code 230
Wiring everything together 231
Declaring the converter and validator in faces-config.xml 231
Creating custom tags 232
Using the converter and validator in a page 232
Internationalization of messages 233
Changing getClientValidator() 233
Changing the JavaScript constructor 234
Formatting the error message 234
Using Trinidad's JavaScript API 235
Writing, testing, and debugging JavaScript 235
Writing JavaScript code 235
Debugging 236
Logging 236
Summary 239
Chapter 7: Trinidad Skinning and Tuning 241
Skinning 241
Understanding the terminology 242
Setting up skinning 242

www.it-ebooks.info

Letting the user choose the skin 244
Creating a Trinidad skin 246
Skinning components 246
Using component state selectors 247
Using component piece selectors 248
Setting global styles using alias selectors 248
Skinning icons 249
Skinning text 251
Extending skins 253
Tuning Trinidad 253
trinidad-config.xml file 254
web.xml file 254
Accessibility 255
Accessibility mode (T) 255
Accessibility profile (T) 255
Lightweight dialogs (W) 256
Performance 256
Page flow scope lifetime (T) 256
Uploaded file processor (T) 256
State saving (W) 257
Application view caching (W) 258
Debugging 258
Enabling debug output (T) 259
Turning off compression and obfuscation (W) 259
Changing deployed files (W) 260
Appearance 260
Client validation (T) 260
Output mode (T) 260
Skin family (T) 261
Localization 261
Time zone (T) 261
Two-digit year start (T) 261
Reading direction (T) 261
Number notation (T) 262
Summary 262
Chapter 8: Integrating with the Backend 263
The Model-View-Controller architecture 263
Setting up the Java EE application structure 264
Creating a skeleton EJB JAR 264
Creating an EAR to wrap them all 265
Preparing a database environment 267
Creating a database 267
Connecting to the database 268
Managing the database 269
Creating a table for employees 271

www.it-ebooks.info

Populating the table with data 272
Implementing the Model 272
Creating an entity 272
Creating a service facade 275
Creating named queries 278
Defining persistence units 279
Defining a data source 280
Using the service facade in the View layer 281
Updating the pages 282
Limitations and problems 283
Transactions 283
Validation of data 284
Summary 284
Chapter 9: MyFaces Orchestra 285
Setting up Orchestra 286
Adapting the application structure 286
Downloading the Spring framework 287
Configuring Spring 288
Letting Spring manage the beans 288
Configuring the faces-config.xml file for Spring 293
Configuring the web.xml file for Spring 293
Configuring Spring and persistence 294
Accessing the services 295
Downloading and installing Orchestra 296
Configuring Orchestra 297
Using the Orchestra ViewController 299
Using event methods 299
Setting up Orchestra conversations 300
Creating a conversation 301
Extending the conversation 305
Ending the conversation 307
Generating forms with DynaForm 310
Installing DynaForm 310
Using DynaForm 311
Summary 313
Chapter 10: Extensions Validator 315
Setting up ExtVal 316
Basic usage 321
Complementing JPA annotations 322
Using ExtVal annotations for standard JSF validators 322
Defining length validation 323
Defining double range validation 323

www.it-ebooks.info

Defining long range validation 323
Defining required fields 323
Using ExtVal's additional annotations 324
Defining pattern-based validation 324
Using custom validators 325
Reusing validation 326
Applying cross validation 328
Using cross validation for date values 329
Using cross validation based on equality 330
Making a value required conditionally 331
Creating custom error messages 331
Overriding standard JSF error messages 332
Overriding ExtVal default error messages 332
Creating our own validation strategy 334
Implementing a custom validation strategy 335
Configuring ExtVal to use a custom validation strategy 337
Using alternative configuration add-ons 338
Testing the custom validation strategy 339
Extending ExtVal in many other ways 340
Extending ExtVal with add-ons 341
Getting add-ons for ExtVal 341
Installing ExtVal add-ons 344
Using Bean Validation 344
Setting up Bean Validation and ExtVal 345
Using Bean Validation annotations 346
Reusing validation 349
Inheriting validation 349
Using recursive validation 350
Composing custom constraints 350
Using payloads to set severity levels 353
Setting up the Continue with warnings add-on 353
Setting the severity level of a constraint 354
Setting the severity level on ExtVal Property Validation constraints 355
Setting the severity level on any constraint 356
Summary 357

Chapter 11: Best Practices 359

Preventing direct access to page definitions 359
Using container-managed security with JSF 362
Enabling container-managed security 362
Navigating to the login page 364
Creating the login page 364
Alternatives 366
Logout link 366

www.it-ebooks.info

Component bindings **368**
Keeping the state of a component **369**
Summary **371**
Appendices
`http://www.packtpub.com/files/3254-Appendices.pdf`
Index **373**



The Appnedices are available for free at `http://www.packtpub.com/files/3254-Appendices.pdf`.

www.it-ebooks.info

Preface

Hypes and trends (such as Web 2.0) cause a change in the requirements for user interfaces every now and then. Although a lot of frameworks are capable of meeting these changing requirements, they often mean that you, as a developer, need in-depth knowledge of web standards, such as XHTML and JavaScript. Apache MyFaces hides all of the details of how the page is rendered at the client, and at the same time offers a rich set of tools and building blocks. This can save you a lot of time not only when you're building a brand-new application, but also when you're adapting an existing application to meet new user interface requirements.

This book will teach you everything that you need to know in order to build appealing web interfaces with Apache MyFaces, and to maintain your code in a pragmatic way. It describes all of the steps that are involved in building a user interface with Apache MyFaces. This includes building templates and composition components with Facelets, and using all sorts of specialized components from the Tomahawk and Trinidad component sets. Adding validation with MyFaces Extensions Validator as well as using MyFaces Orchestra to manage transactions in a page flow, are also covered.

Unlike comparable books, this book not only introduces Facelets as an alternative to JSP, but actually uses Facelets in all the examples throughout this book. This makes the book a valuable resource for Facelets examples. The book also shows how various components of the MyFaces project can be used together, in order to deliver the functionality of the new JSF 2.0 standard, in current projects, without the need to upgrade your project to JSF 2.0.

This book uses a step-by-step approach, and contains a lot of tips based on experience of the MyFaces libraries in real-world projects. Throughout the book, an example scenario is used to work towards a fully-functional application by the end of this book.

This step-by-step guide will help you to build a fully-functional and powerful application.

www.it-ebooks.info

What this book covers

Chapter 1, *Introduction*, introduces the Apache MyFaces project and all of its subprojects. Forward references to other chapters are given wherever applicable.

Chapter 2, *Getting Started*, discusses downloading and installing the MyFaces libraries. The set-up of two specific IDEs is discussed, as well as the set-up of an application server for testing. This chapter also covers the use of Maven and the Maven artifacts that are provided by the MyFaces project.

Chapter 3, *Facelets*, covers the installation of Facelets into our project. It discusses the benefits of Facelets over JavaServer Pages as a view technology for JavaServer Faces. This chapter also introduces the most important features of Facelets. By the end of the chapter, we have created a layout template that we can use throughout the book, when developing our application. We will also have learned the basic Facelets techniques that we will use in all examples throughout the book.

Chapter 4, *Tomahawk*, looks at the Tomahawk component set that is a part of MyFaces. Some of the most important components from the set are covered, and we will learn how we can use these in an optimal way, in combination with Facelets. This chapter gives us enough information to build fully-functional JSF pages by using Tomahawk components.

Chapter 5, *Trinidad – the Basics*, is the first of three chapters covering MyFaces Trinidad. This chapter introduces a lot of Trinidad components, including the data input and output components. Special attention is given to the many layout components that are available in the Trinidad library. As with Tomahawk, we will see how we can get the most out of the combination of Trinidad and Facelets.

Chapter 6, *Advanced Trinidad*, introduces some more advanced features of the Trinidad library. This includes the charting component that can be used to easily create nice looking charts. Also, Trinidad's page flow scope feature, which enables us to create page flows more easily, is introduced. This chapter also discusses the AJAX or Partial Page Rendering capabilities of Trinidad, including client-side validation and conversion. The Trinidad dialog framework is also covered.

Chapter 7, *Trinidad Skinning and Tuning*, is an introduction to the advanced skinning framework that is a part of Trinidad. This chapter also discusses the most important tuning parameters of Trinidad.

www.it-ebooks.info

Chapter 8, *Integrating with the Backend*, discusses how we can integrate the frontend that we created with some backend system, in a standard way. This chapter gives us some basic knowledge about the Model-View-Controller architecture, and about important standards such as Enterprise Java Beans (EJB) and the Java Persistence API (JPA). We will use the knowledge and examples from this chapter as a starting point for the more advanced integration topics discussed in the subsequent chapters.

Chapter 9, *MyFaces Orchestra*, introduces the MyFaces Orchestra library. This chapter starts with a very brief introduction to the Spring framework, as Orchestra is based on parts of that framework. We see how we can create a Spring application context and then how we should use such a context in combination with Orchestra. Some key concepts of Orchestra are introduced, such as the Orchestra ViewController concept and the concept of conversations. This chapter concludes with a quick view of Orchestra's DynaForm component.

Chapter 10, *Extensions Validator*, is about one of the latest additions to the MyFaces project: the Extensions Validator, or ExtVal for short. This chapter starts by teaching us how to configure our project to use ExtVal. We see how JPA annotations can be used to automatically generate JSF validations. This chapter also shows us the extra annotations that ExtVal offers to complement the JPA annotations. This chapter also shows how we can use Bean Validation (JSR 303) annotations as an alternative to JPA and ExtVal annotations. As a whole, this chapter is a good introduction to this very flexible and versatile member of the MyFaces family.

Chapter 11, *Best Practices*, is the last chapter of this book. It discusses some best practices with JSF in general and MyFaces in particular. This chapter describes a way to prevent direct access to page definitions, as well as a way to enable container-based security in our JSF application. This chapter also shows how to create a login page by using JSF components, and discusses how to use component bindings wisely. This chapter concludes by discussing how to save the state of request-scoped components in an elegant way.

Appendix A, *XHTML Entities*, lists all of the numeric entities that can be used in XML documents. This list may be needed because Facelets files must be valid, plain XML files, and can't contain named entities that can be used in normal XHTML files.

Appendix B, *Trinidad Tags*, gives a list of all of the tags from the Trinidad library. This can be referred to if you forget the exact name of one of the many tags. It can also be used to determine if a certain Trinidad tag is a naming container.

Appendix C, *Trinidad Text Keys*, lists the keys that Trinidad uses to lookup the default texts that are displayed on components. These keys can be used to customize or translate the default texts.

www.it-ebooks.info



www.it-ebooks.info

Code words in text are shown as follows: "There are two important configuration files for a JSF application—`web.xml` and `faces-config.xml`."

A block of code is set as follows:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<sun-web-app>
  <security-role-mapping>
    <role-name>user</role-name>
    <group-name>miasusers</group-name>
  </security-role-mapping>
  <class-loader delegate="false"/>
  <property name="useMyFaces" value="true"/>
</sun-web-app>
```

Any command-line input or output is written as follows:

```
connect 'jdbc:derby://localhost:1527/test;create=true';
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Under the **Web Tier** node, we select the **JSF** node and then on the right-hand side, we select **JSF Page** and click on **OK**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

www.it-ebooks.info

Reader feedback

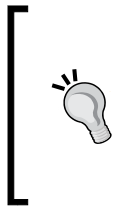
Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support



www.it-ebooks.info

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

www.it-ebooks.info

www.it-ebooks.info

1
Introduction

This chapter introduces Apache MyFaces and its subprojects. At the end of the chapter, you will have an idea about what can be done with MyFaces, and about the role of the several subprojects.

We cover the following topics in this chapter:

- A brief introduction to the Apache MyFaces project
- Some insights in the history of Apache MyFaces
- An overview of all of the subprojects of Apache MyFaces

Introducing Apache MyFaces

Apache MyFaces started out back in 2002 as the first open source implementation of the **JavaServer™ Faces (JSF)** standard. In July 2004, the project became part of Apache as an **Apache Incubator** project. In February 2005, MyFaces was promoted to a top-level Apache project. By the time MyFaces was submitted as an Apache project, it was already more than just an implementation of the standard. As the set of components defined by the JSF standard is rather limited, MyFaces went beyond the standard by adding more components and extended versions of the standard components. In April 2006, the extended components were separated from the core JSF implementation. From that moment, the components are in a subproject called **Tomahawk**, and the core JSF implementation is in the **Core** project.

Over time, Apache MyFaces was further expanded by other subprojects. Some of them added even more extended components, while others focused on other extended functionalities such as persistence scopes and annotation-based validation. Support for the use of **Facelets,** instead of JSP, was also added. Chapter 3 focuses on how we can get the most out of MyFaces by using Facelets. In the remaining chapters, we will use Facelets as the view technology.

www.it-ebooks.info

License, community, and support

Because it is a part of the Apache project, Apache MyFaces and all of its subprojects are available under the terms of the liberal **Apache License, Version 2.0**. Judging by the list of companies using MyFaces at the MyFaces wiki (`http://wiki.apache.org/myfaces/Companies_Using_MyFaces`), Apache MyFaces is widely used, worldwide.

The MyFaces project as a whole has an active community of developers all over the world. Many individuals, as well as a number of companies, contribute to the projects. As one of the co-founders of the MyFaces project, the Austrian consulting company, Irian Solutions GmbH, has a large number of employees contributing to the project. The multinational software giant, Oracle Corporation, also has a number of employees contributing mainly to the Trinidad subproject, which was started with a large donation from Oracle.

Good community support for MyFaces and all subprojects is available through the **MyFaces user mailing list**. Detailed information on this mailing list, as well as a subscription link, can be found at `http://myfaces.apache.org/mail-lists.html`. It is good manners to start the subject line of any message to the list with the name of the subproject in square brackets such as `[Core]` or `[Tomahawk]`. As with most mailing lists, it is worth searching the history to check if your question has been asked already. A good tool to search the mailing list archives is provided by MarkMail, a link to which is provided at the page mentioned earlier.

MyFaces and Sun JSF RI

A lot of application servers are using Sun Microsystems' **JSF Reference Implementation** (Sun JSF RI) of the JSF standard, which is also known by its code name, **Mojarra**. In most cases, it is possible to replace the application server's default JSF implementation with MyFaces; however, this is almost never necessary. As both the Sun JSF RI and MyFaces core implement the same standard, there should not be much difference between using one or the other. All subprojects of Apache MyFaces should work on both MyFaces Core and Mojarra/Sun JSF RI without any problems.

On the other hand, if your application server uses Sun JSF RI by default, switching to MyFaces might be worth considering, especially during the development stage, as MyFaces Core gives a lot more diagnostic and debug information in the server logs than Mojarra does.

www.it-ebooks.info

Whatever you choose, it is always useful to know which JSF implementation you are running your application on. When you encounter a strange problem, which could be a bug, one of the first questions on the mailing list will probably be what implementation you are running. An easy way to determine which implementation you are using is to consult the server logs of your application server. During start up, the name and version of the JSF implementation will be written into the log file.

Subprojects of Apache MyFaces

Apache MyFaces consists of many subprojects. Each subproject has its own release cycle. Most projects are available for JSF 1.1 as well as JSF 1.2. We will look at each of the subprojects briefly in this section to get an idea of what can be done with MyFaces. Where applicable, a forward reference is provided to a chapter where the project is discussed in detail.

Note that there is much overlap in functionality between some of the subprojects, and in particular in the component sets. This is because some component sets were donated to the MyFaces project by different companies at different times. Each of these component sets has its weaknesses and strengths. That being said, choosing a component set to be used for your project may not be an easy task. The information presented in this chapter is not sufficient to make such a choice. So, it is best to read the chapters on each component set before choosing any of them.

Core

The Apache MyFaces Core project is where it all started. It does not do much more than implement the JSF standard. There are currently three relevant versions of this standard:

- **JSF 1.1**: This version of the standard fixes some bugs in the original 1.0 version. Both the 1.0 and the 1.1 versions were specified by the **Java Specification Request (JSR)** number **JSR 127**. The versions of MyFaces Core implementing this version of JSF are numbered 1.1.x.
- **JSF 1.2**: This version of the JSF standard was specified by a separate JSR, with the number **JSR 252**. JSF 1.2 adds some important improvements over JSF 1.1. The MyFaces Core versions implementing JSF 1.2 are numbered 1.2.x.
- **JSF 2.0**: This brand new version of JSF will be part of Java EE 6.0. JSF 2.0 is specified in **JSR 314**, which had its final release on July 1, 2009. The versions of MyFaces Core implementing JSF 2.0 are numbered 2.0.x. At the moment of writing this book, only an alpha release is available.

www.it-ebooks.info

The Core project currently has branches for each of these JSF versions. The 1.1 and 1.2 versions are stable and are updated regularly. Work is in progress on the 2.0 version, but a stable release is not yet available, as of the time of writing this book. In this book, we will focus on JSF 1.2. We'll see that many features defined in the JSF 2.0 standard are already available in JSF 1.2 through one of the many subprojects of MyFaces.

You may have noticed that there is no chapter on the MyFaces Core project in this book. This has to do with the fact that the MyFaces Core project implements only the JSF standard. That means that discussing the Core project in detail would be more or less the same as discussing the JSF standard in detail, and that is beyond the scope of this book. However, some good books on JSF are available, and you can refer to them. Of course, we will discuss specific issues that we will come across when using MyFaces Core as a JSF implementation on our application server. These issues will be discussed in Chapter 2, *Getting Started*.

Tomahawk

Tomahawk has been part of the Apache MyFaces project from the very beginning. Its main goal is to implement a larger set of components than the rather minimal set that is defined in the JSF specification. Tomahawk contains both extended versions of standard JSF components and components that are not defined at all in the standard. Some examples of components that are added by Tomahawk are:

- A CAPTCHA component
- An extensive data table component that includes pagination functionality for larger data sets
- A file upload component
- Various date selection components

Tomahawk also features some custom validators that make it easy to validate, for instance, credit card numbers and email addresses.

There are two variants of Tomahawk, as follows:

- Tomahawk Core is compatible with both JSF 1.1 and JSF 1.2
- Tomahawk Core 1.2 takes advantage of some features of JSF 1.2, making it incompatible with JSF 1.1

We will take a detailed look at Tomahawk and all of its components in Chapter 4.

www.it-ebooks.info

Sandbox

The MyFaces project also has a subproject called **Sandbox**. The Sandbox project is a testing ground for new JSF components that might be added to Tomahawk in the future. Sandbox components are generally "work in progress" and are generally not "feature complete". Sandbox components may or may not be promoted to the main Tomahawk project.

Due to the dynamic character of the project and the uncertain status of its contents, it doesn't make sense to cover it in a book. However, if you're looking for a special or advanced component that cannot be found in one of the other component sets, it may be worth looking in the Sandbox. You should realize that it might be harder to get support for Sandbox components, and that there is some uncertainty about their future. On the other hand, you might be able to help to get the component to production quality so that it could be promoted to Tomahawk.

Trinidad

Trinidad is a very extensive set of JSF components. It was developed by Oracle under the name **ADF Faces**. When Oracle donated this component set to the ADF Faces project, it was renamed to Trinidad. As with the Core project, there are two versions of Trinidad for each version of the JSF standard. Trinidad releases numbered 1.0.x are compatible with JSF 1.1, and releases numbered 1.2.x are compatible with JSF 1.2.

Some of the most important characteristics of Trinidad are:

- It has a large number of components. There is a component for nearly everything, in Trinidad.
- It has many "all-in-one" components. Some of the Trinidad components render multiple elements. With some other component sets, we need several components to get the same effect.
- It has advanced skinning possibilities.
- It has a lot of advanced options for displaying tabular and hierarchical data.
- It has a chart component, making it fairly easy to display numeric data in a visual way.
- It has a dialog framework, making it easy to create pop-up dialogs.
- "Partial page rendering", client-side validation and conversion, and other AJAX functionality are embedded in most components and is easy to add where needed.

MyFaces Trinidad will be covered in detail in Chapters 5, 6, and 7.

www.it-ebooks.info

Tobago

Tobago is the third set of components that is a part of MyFaces. Tobago was contributed to the MyFaces project by a German company, Atanion GmbH. The emphasis is on separating structure from design. Tobago offers the same concept of extended components as Trinidad does, with one component rendering several elements. It also uses the concept of a layout manager, a little bit like "good old" Swing does. Tobago comes with four different themes that you can choose from. Unfortunately, there is not enough space in this book to cover this third component library.

Orchestra

MyFaces **Orchestra** is aimed at making it easier to use transactions to persist data in a database. To achieve this, extended scopes are provided, in addition to JSF's standard **application**, **session**, and **request** scopes. Orchestra is mainly useful in applications where a lot of data is entered into a database with a JSF frontend. Whereas the standard JSF scopes are based on how a web server works, Orchestra focuses on what a lot of applications need, and adds a **Conversation Scope**, making it easier to keep certain actions within the same Java Persistence API (JPA) transaction, whether they are on the same page or not. Orchestra is based on parts of the Spring 2.0 framework and works with JPA-based persistence layers. Orchestra is covered in Chapter 9 of this book.

Portlet Bridge

Portlet Bridge is one of the newer subprojects of the MyFaces project. It is still in the alpha stage at the time of writing this book. It will be the reference implementation of the **JSR 301** standard. The JSR 301 standard is an effort to standardize the way that JSF artifacts (pages or parts of pages) can be used as **portlets** within a portal environment. Considering the stage of both the JSR 301 standard and the Portlet Bridge subproject, it will not be covered in this book.

Extensions Validator

The **Validator** project was recently added under the **MyFaces Extensions** umbrella project. This project was created as a separate project by its lead developer under a different name, but joined the MyFaces project in December 2008. Although the full name of the project is **MyFaces Extensions Validator**, it is mostly referred to as **ExtVal**.

www.it-ebooks.info

The goal of ExtVal is to eliminate the need to repeat validation code in the View layer of a Java EE application. This is often necessary in order to give user-friendly error messages. However, according to the **Don't Repeat Yourself** (**DRY**) principle, repeating code is not desirable. ExtVal uses standard JPA annotations to dynamically create JSF validations. It also adds some extra annotations for cases where a validation cannot be expressed in a JPA annotation. Thus, ExtVal eliminates the need to add validators to JSF pages as well as the need to repeat validation code.

It should be noted that there is a JSR with number 303 that aims to achieve a comparable goal. **JSR 303** is called **bean validation**, and will be part of Java EE 6.0. ExtVal will be compatible with JSR 303 and already goes beyond what is possible with a bare JSR 303 implementation. ExtVal will be discussed in detail in Chapter 10.

Summary

In this chapter, we learned a little about the history of Apache MyFaces and took a quick look at several sub-projects of Apache MyFaces. We saw that, apart from the Core implementation of the JSF standard, Apache MyFaces offers a lot of additional libraries, making the life of a JSF developer easier and more fun.

Most subprojects of MyFaces introduced in this chapter will be discussed in much more detail in seperate chapters of this book. We will also look into the use of these libraries in real-life projects. A sample case will be used throughout the book, and will be introduced in the next chapter. The next chapter will also discuss how to prepare our development environment and our application server for use with MyFaces and its subprojects.

www.it-ebooks.info

www.it-ebooks.info

2
Getting Started

Before we can start building a JSF application with MyFaces, we have to prepare our development environment and our application server. This chapter focuses on both of these things.

In this chapter we will cover the following topics:

- Installing the Apache MyFaces libraries in both Eclipse and JDeveloper
- Creating an empty project in which a JSF application with MyFaces can be created
- The most important configuration files for the application server on which the application is to be deployed
- Introduction to the example case that will be used throughout this book

Configuring the development environment

In this section, we will discuss two Integrated Development Environments (IDEs) that can be used for developing JSF applications with the Apache MyFaces libraries:

- **Eclipse** is covered because it is very widely used, freely available, and open source
- **JDeveloper** is covered because it has features that are extra powerful when used in combination with MyFaces Trinidad, and is also free to use

www.it-ebooks.info

Of course, you are free to choose your own favorite IDE, but we cannot cover all available IDEs here. The rest of the book is written to be independent of the tools that you use. All of the examples in this book can be executed whether you use one of the two IDEs covered here, another IDE, or no IDE at all. The downloadable source code is created with Eclipse, but it should be possible to use it in other environments with little effort.

In the next subsections, we describe the steps that need to be taken in order to install the MyFaces libraries in selected IDEs for optimal integration with the IDE. Also, the steps required to create a new project that uses those libraries are described. In a third subsection, an alternative approach to creating a new project by using Maven is described.

Configuring Eclipse

For configuring Eclipse, we use the latest stable version, that is, Eclipse 3.5—"**Galileo**". Be sure to download the "**Eclipse IDE for Java EE Developers**" edition. This edition already contains most of the required plugins. If you already have Eclipse installed, be sure that you have installed at least the "**Java EE Developer tools**" and the "**Web developer tools**" packages.

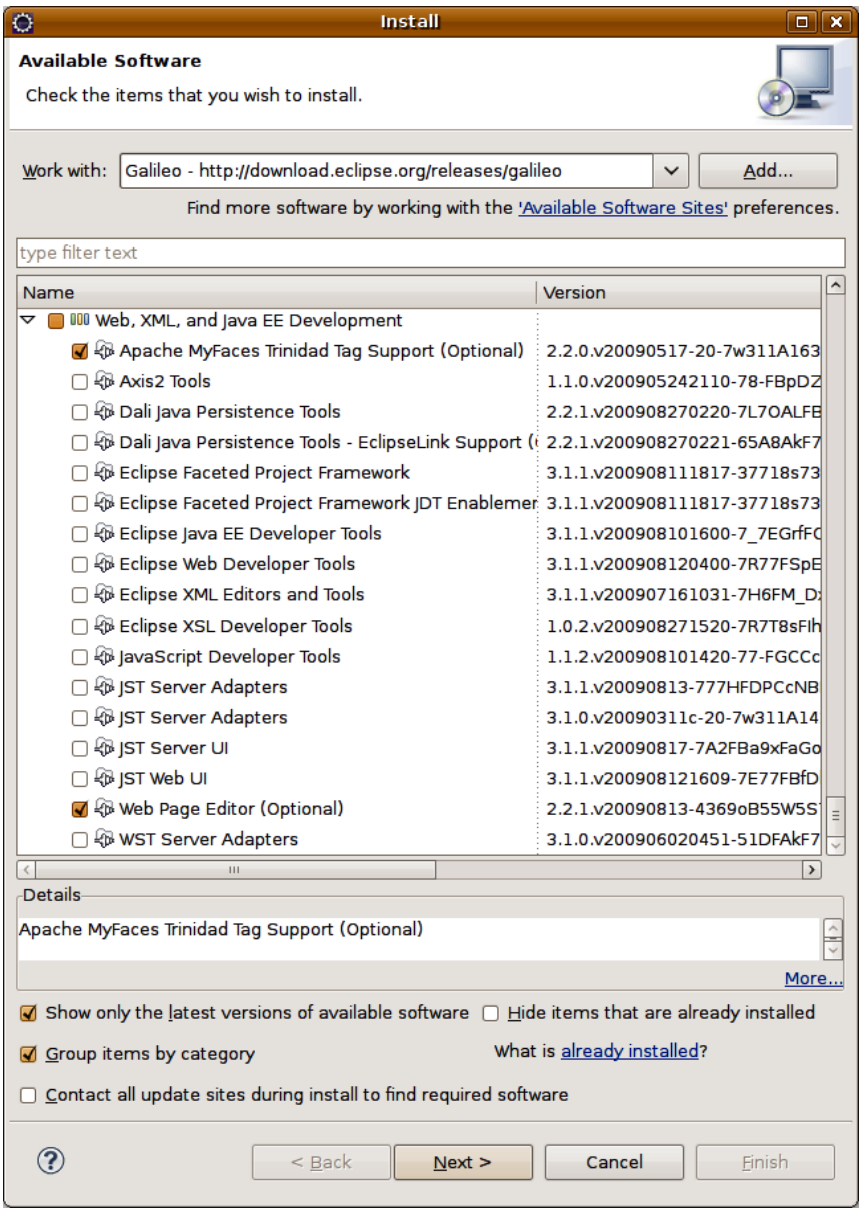
Installing extra plugins

To have better support for the various MyFaces libraries in editors, there are some extra plugins that you may want to install:

- **Web Page Editor**: This is an advanced editor for JSPs and similar technologies. The web page editor features a palette with JSF components that you can drag-and-drop onto your pages and a "live" preview mode, although the preview is still limited and certainly not "What You See Is What You Get".
- **Apache MyFaces Trinidad Tag Support**: This adds support for the components of the Trinidad library in the Web Page Editor and property panes. Unfortunately, the other tag libraries from MyFaces (Tomahawk and Tobago) are not supported by this or any other plugin.

www.it-ebooks.info

Both plugins can be installed through the Eclipse update manager (**Help | Install New Software...**), as shown in the following figure:



www.it-ebooks.info

Installing the libraries

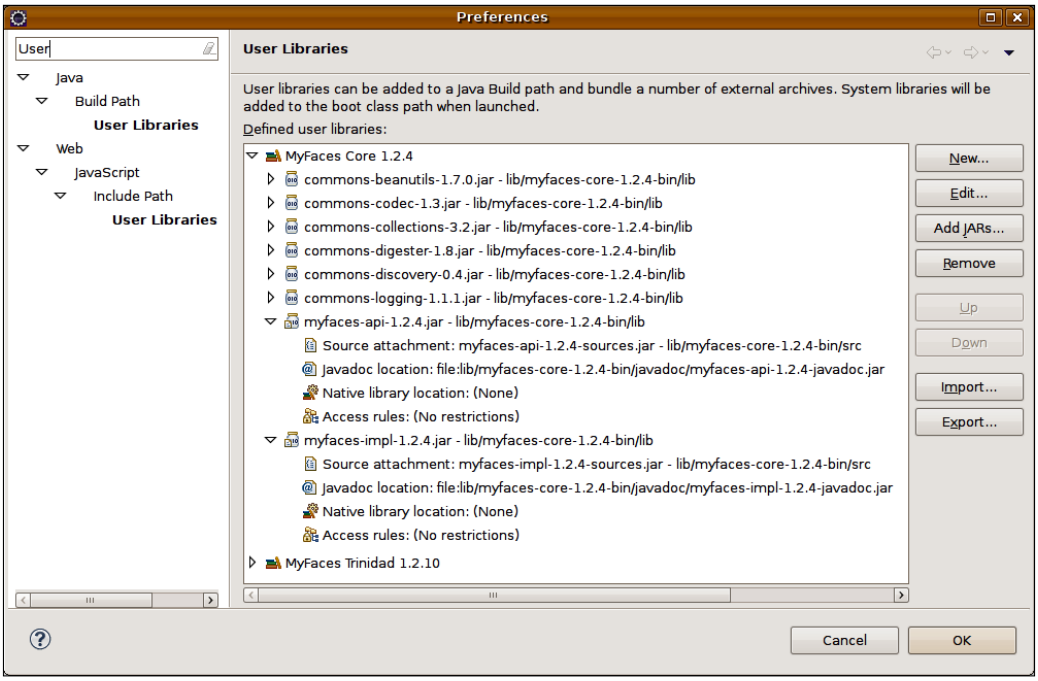
Now that we have Eclipse itself configured to have all of the editors and property panes in place, we have to install the libraries. Of course, we have to download and unpack the libraries from the Apache MyFaces website. Each subproject of MyFaces has its own download page. On the main download page of MyFaces, the latest version of MyFaces core can be downloaded, and links are provided to the download pages of the sub-projects. The main download page can be found at `http://myfaces.apache.org/download.html`.

Once the libraries are downloaded and unpacked, we can tell Eclipse where they can be found. In order to do so, we start by opening the **Preferences** window by clicking on **Window** | **Preferences**. In the **Preferences** window, navigate to **Java** | **Build Path** | **Libraries**. For each of the libraries, we perform the following steps to create a user library definition:

1. We click on the **New...** button. The **New User Library** window appears.
2. We have to enter a name for the library. It is advisable to include the version number in the name, in case a newer version is added later on. For example, we could specify "MyFaces Core 1.2.4".
3. Now the created user library entry appears in the list. Select it and click on the **Add JARs...** button. Browse to the directory where you unpacked the downloaded library. Within the directory where the library is unpacked, there should be a `lib` directory containing all the JARs that are needed. If you downloaded MyFaces Core 1.2.4, it is probably unpacked in a directory named `myfaces-core-1.2.4-bin`; and in that directory is a subdirectory named `lib`. Select all the `.jar` files in that directory, and then click on **OK**.

For easy debugging and online Javadoc reading, we can now add references to the sources and Javadocs that are bundled with the downloaded libraries. This can be done by selecting one of the JARs and expanding its node. Each JAR node will have a number of child nodes, among which are at least **Source attachment: (None)** and **Javadoc location: (None)**. By selecting one of these child nodes and clicking on the **Edit...** button, the location of the provided sources and Javadocs can be set. For the MyFaces Core, the window should now look as displayed in the following image:

www.it-ebooks.info



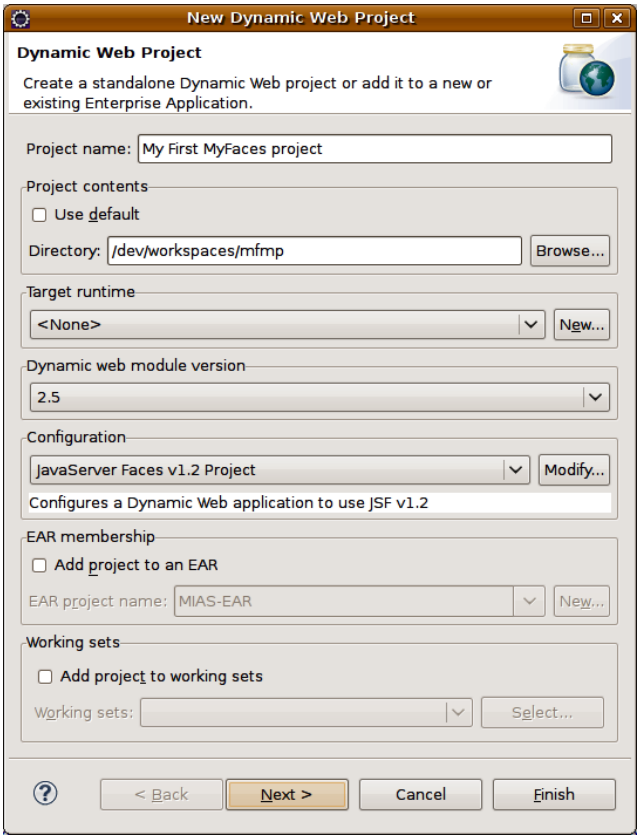
Preparing a new project

We are now nearly finished preparing Eclipse. To make sure everything is in place, we start a new project in Eclipse. To do so, we follow these steps:

1. Start the wizard by choosing **File** | **New** | **Dynamic Web Project**.
2. Choose a name for our project.
3. Under **Project contents**, we can just leave the **Use default** checkbox selected, or choose a custom location, if we like.
4. If there is a **Target runtime** definition available, we could choose one. If not, we will just leave it at **<None>** and choose which server to deploy to, later.
5. Of course, we choose the latest **Dynamic web module version**, that is, **2.5**.

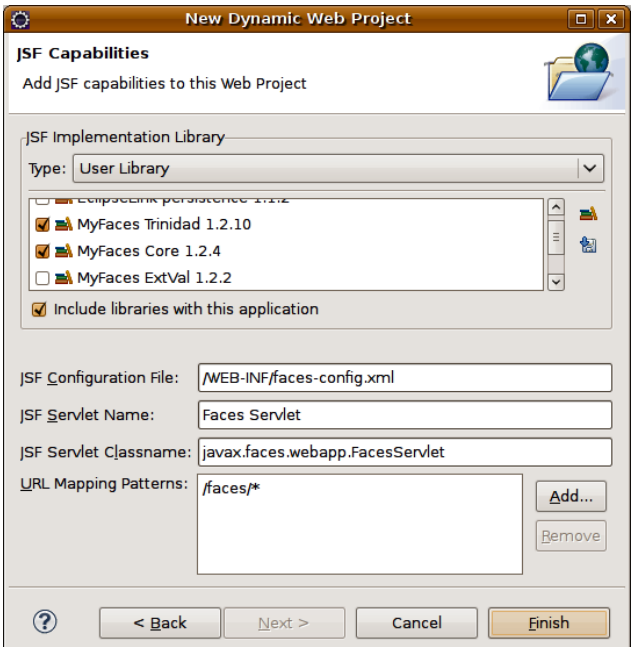
www.it-ebooks.info

6. Under **Configuration**, we choose **JavaServer Faces v1.2 Project**.
7. For now, we do not choose to add our project to an EAR project. This can always be done later on. We click on **Next >** to go to the next step in the wizard.



8. In the second and third steps, we can just leave all of the defaults, and click on **Next >**.
9. The fourth and last step of the wizard is again important. First, we have to choose our JSF libraries. All of the user libraries are listed; these should include the MyFaces libraries that we added previously. We can select the libraries that we want to use in our project by selecting the checkboxes in the list. The **Include libraries with this application** checkbox tells Eclipse to add our JSF libraries to the WAR file that will be deployed to the server.
10. The other options in this screen can be left at the default values for now. The next screenshot shows how the window should look now. If everything is alright, let's click on the **Finish** button.

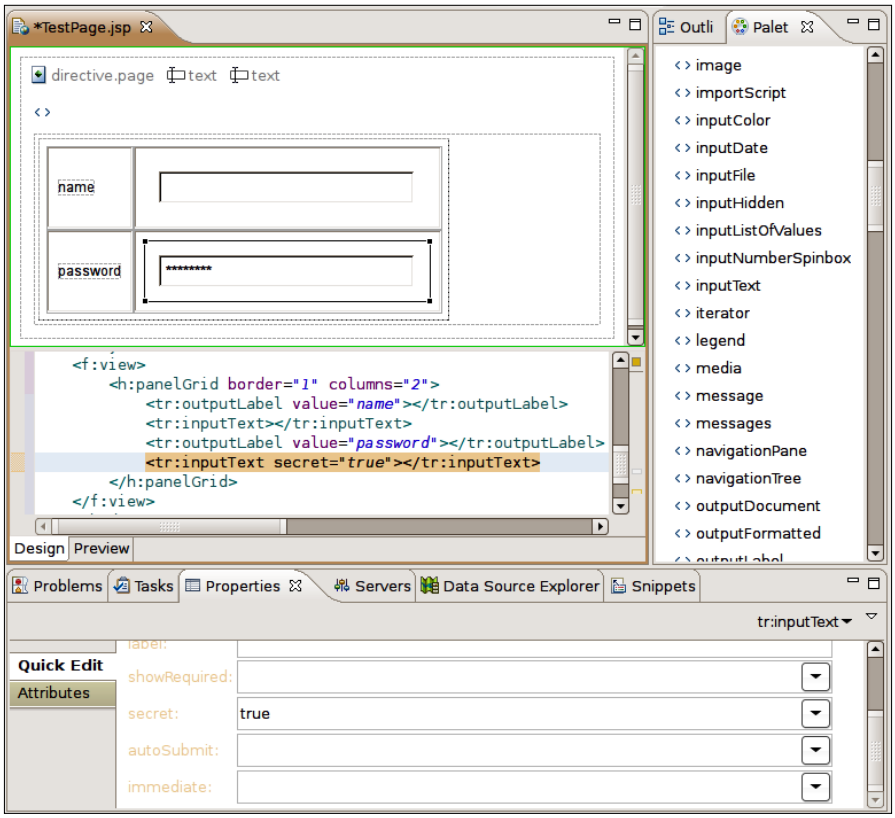
www.it-ebooks.info



www.it-ebooks.info

On the right-hand side is a palette that contains the components that are ready to be placed on a page. Sometimes, this palette is not initially shown. If that is the case, we can make it visible through **Window | Show View | Other...**. The **Show View** window opens; under the **General** node, click on **Palette**. Now we can just select some components and place them on the page. Note that it's not exactly drag-and-drop. A component is selected by clicking on it once in the palette, and then placed on the page by clicking on the required position in either the preview or the source pane. The cursor changes to indicate that we are going to drop a component onto the page. If a component is dropped on the page, we can edit its attributes by selecting the component and then opening the **Properties** tab.

The following figure shows a simple login page created by using one JSF core component (`<h:panelGrid>`) and two Trinidad components (`<tr:outputLabel>` and `<tr:inputText>`). Note that the preview is far from what it will look like in a browser, but it gives some idea about the layout of the page. Unfortunately, the preview mode doesn't work at all with the Tobago or Tomahawk components. For these components, only a placeholder is shown. There aren't any property panels defined for either of these components.



www.it-ebooks.info

As stated before, the visual editor just gives a hint of how the page should look in a browser. That said, the use of this visual editor is very limited. When we start using Facelets instead of JSPs in the next chapter, we'll see that the visual editor is even less useful. So, the recommended way to edit JSF pages is still to use the code editor. However, the property panes can be handy if you find it difficult to remember all possible attributes of the different JSF components.

If you have an application server installed and configured, it should be possible to build the project and deploy it to the application server to test. Note that the page we created was just to test if everything is configured correctly. It can be thrown away, as we will use Facelets instead of JSP from next chapter onwards. However, the created project might be a good starting point for the next chapters.

Configuring JDeveloper

The **Trinidad** components were donated to Apache MyFaces by Oracle, and are the former **ADF Faces 10g** component set. (After the donation, Oracle developed its **ADF Faces 11g** component set that is not (yet) donated to Trinidad.) For this reason, the Trinidad components are fairly well integrated with the visual editors of Oracle's **JDeveloper** development environment—a good reason to take a look at JDeveloper and how to configure it for use with MyFaces. There are several versions of JDeveloper available for download from the Oracle website. All versions are free to use and you are free to deploy any application created with JDeveloper, as long as you don't use Oracle's **Application Development Framework** (**ADF**). Let's take a look at the latest production version of JDeveloper. At the time of writing of this book, this happens to be JDeveloper 11.1.1. Be sure to download the "**Studio Edition**" that comes with all the bells and whistles. No extra plugins have to be installed; JDeveloper is an "all inclusive" package.

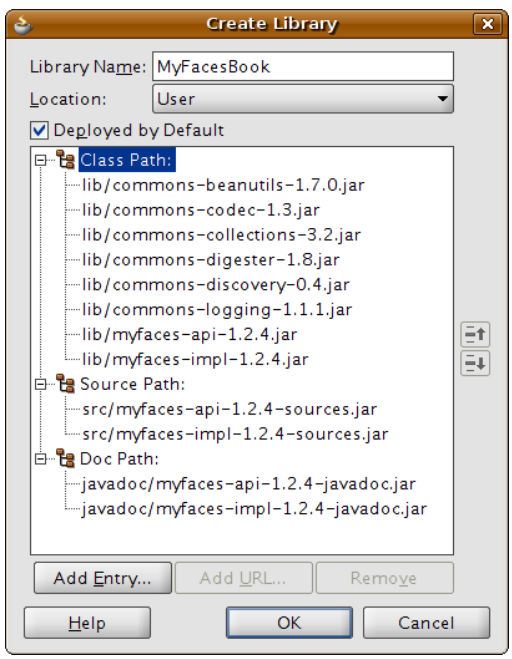
Installing the libraries

We need the same libraries that we downloaded for preparing Eclipse. If you skipped that, download them now from `http://myfaces.apache.org/download.html`. The Trinidad library does not have to be installed as it is a standard part of the JDeveloper 11g package. (In the earlier versions of JDeveloper, Trinidad was not included.) However, if we want to use the MyFaces core instead of the Sun JSF RI that is delivered with JDeveloper, and/or we want to use Tobago or Tomahawk components, we have to install these libraries.

www.it-ebooks.info

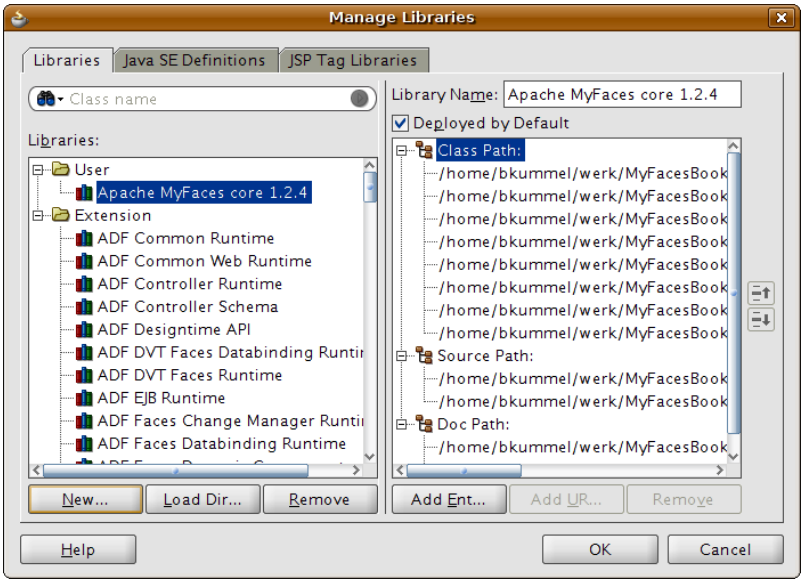
To install the MyFaces Core, we follow these steps in JDeveloper:

1. Go to **Tools | Manage Libraries...**.
2. In the **Manage Libraries** window, we select the first tab, **Libraries**.
3. Click on **New...**.
4. In the **Create Library** window, we enter a name for our library; for instance, `Apache MyFaces core 1.2.4`.
5. We select the **Deployed by Default** checkbox to make sure that our library will be deployed to the application server.
6. In the list box, we select the **Class Path:** node, and then click on **Add Entry...**.
7. In the **Select Path Entry** window, we navigate to the `lib` directory in the directory where we unpacked the MyFaces library. Select all of the JAR files in this directory, and then click on the **Select** button.
8. If desired, we could add a JAR with the sources in the same way, but with the **Source Path:** node selected. The same goes for the documentation.
9. The **Create Library** window should now look more or less as shown in the next screenshot. Click on **OK** to finish.



www.it-ebooks.info

10. The created library should now show up under the **User** node in the **Manage Libraries** window, as shown in the next screenshot:

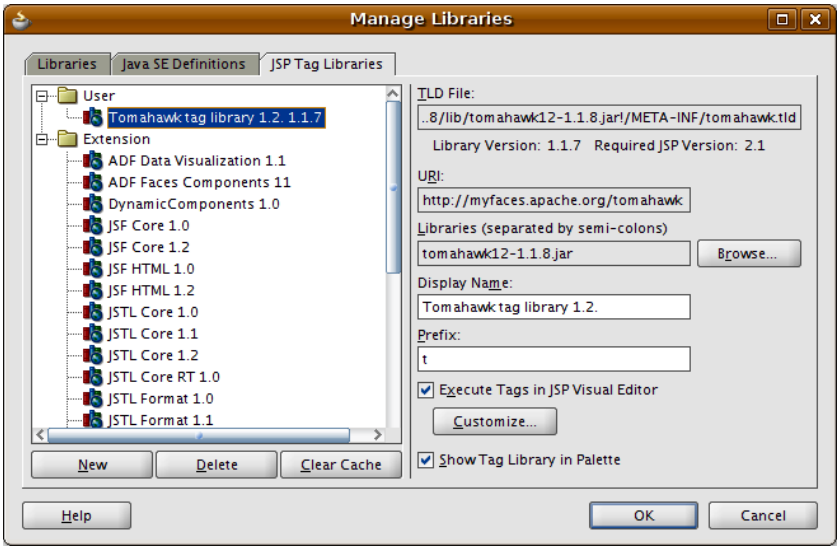


The process of adding a component library is slightly different. Let's take Tomahawk as an example this time:

1. Go to **Tools | Manage Libraries...**.
2. In the **Manage Libraries** window, select the **JSP Tag Libraries** tab.
3. Make sure that the **User** node in the tree is selected.
4. Click on the **New** button.
5. Browse to the `lib` directory in the directory where the library (in this example, Tomahawk) was unpacked. Only JARs that contain tag definitions are shown. In the case of Tomahawk, this would be `tomahawk12-1.1.x.jar`.
6. Select **Open** to add the library. JDeveloper will read the tag definition information and use it to complete all the fields, so we don't have to enter a name ourselves.

www.it-ebooks.info

7. The library is now added, as shown in the following screenshot:



8. Once the library has been added, we can select the **Execute Tags in JSP Visual Editor** checkbox to enable the components to be rendered in the design view of JDeveloper. Unfortunately, this option doesn't seem to work for Tobago or Tomahawk.
9. The **Show Tag Library in Palette** checkbox should be checked, unless you don't want to use the components that you just added.

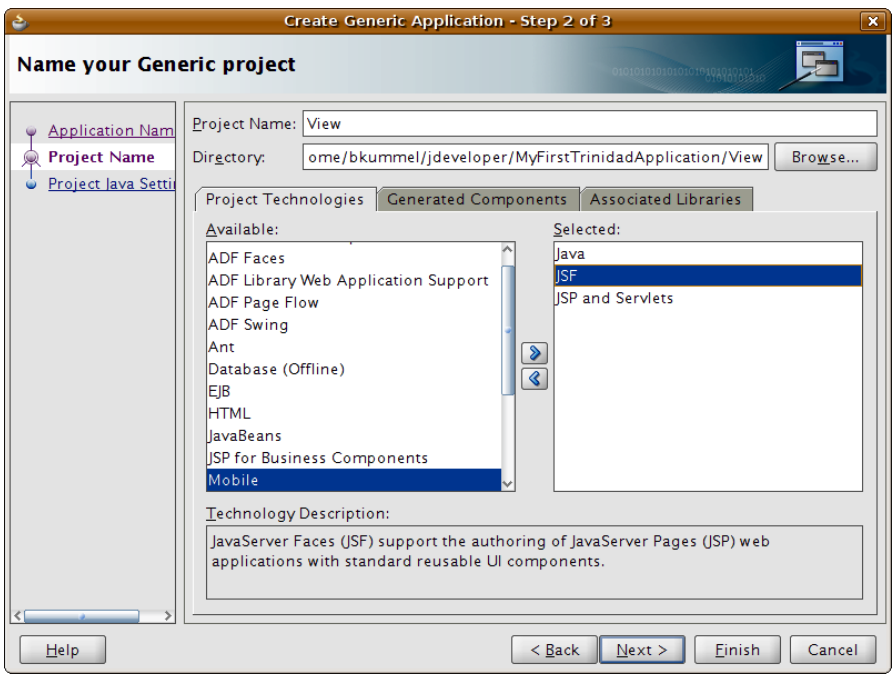
Preparing a new project

Now, to check if adding the libraries was successful, we can create a small test project. In JDeveloper, a project can exist only within an application. Therefore, we have to define an application first by following these steps:

1. Open the **New Gallery** via **File | New...**.
2. Select the **General** node in the tree on the left-hand side.
3. Select **Generic Application** from the list on the right-hand side.
4. Click on **OK**.
5. Now, the **Create Generic Application** wizard starts, and we have to come up with a name for the application. We can also change the location on disk where the application's files will be stored, and configure a global package name for the entire application, such as `com.ourcompany`.

www.it-ebooks.info

6. By clicking on **Next >**, we arrive at the second step of this wizard. This is where we define the project. We have to name our project, too. This is because Oracle wants us to separate our model from our view in separate projects, according to the Model-View-Controller pattern. So in this case, a good name for our project would be `View`.
7. On the **Project Technologies** tab, we can choose which technologies are going to be used. We are going to create a JSF project, so we just have to select **JSF** and move it to the right by clicking on the appropriate arrow button. Note that some technologies on which JSF depends are automatically selected too. The window should now look as displayed in the following screenshot:



8. After clicking on **Next >**, we can accept all defaults in the last step of the wizard, and then click on **Finish**.

www.it-ebooks.info

Before we can use the libraries that we added to JDeveloper earlier, we have to add them to our project as well. To do so, we right-click on our project and choose **Project Properties...**. In the tree on the left-hand side, we select the node **JSP Tag Libraries**. We click on the **Add** button, and then select the desired libraries. The libraries that we added ourselves are under the **User** node, but Trinidad is a part of the JDeveloper package and so is under the **Extension** node. After selecting the libraries, we click on **OK** to confirm. Close the project properties by clicking on **OK** again.

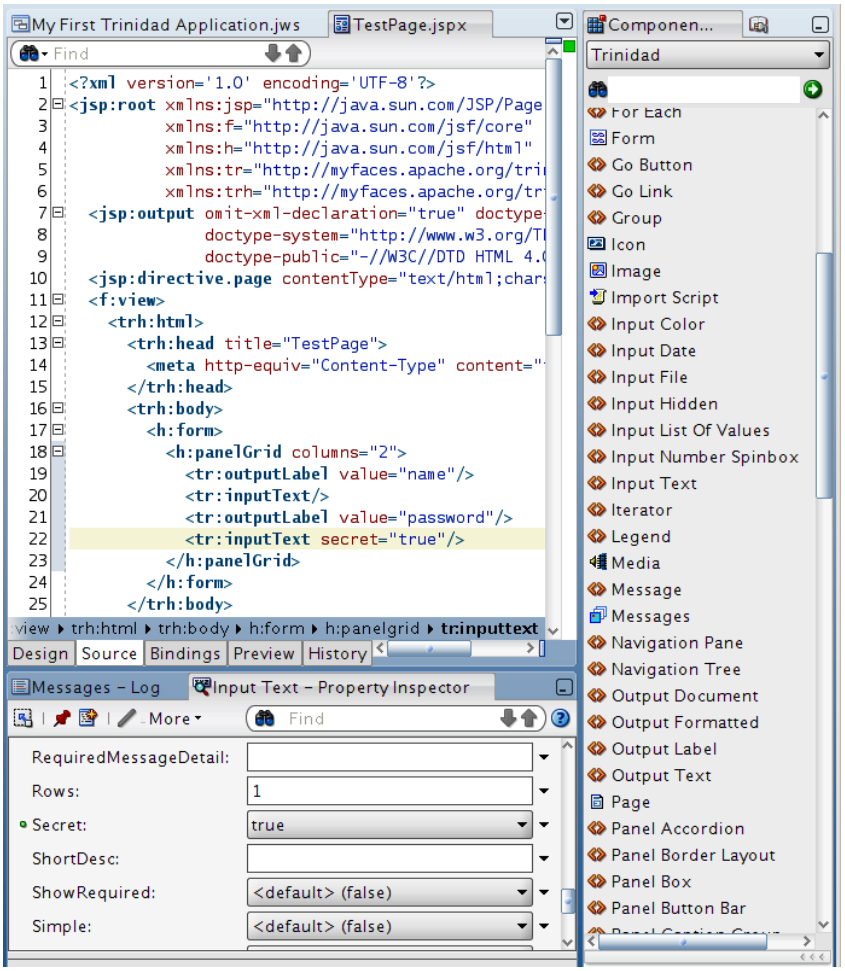
Now, let's create a simple test page. Right-click on our project and select **New...** from the context menu to open the **New Gallery**. Under the **Web Tier** node, we select the **JSF** node, and then on the right-hand side, we select **JSF Page** and click on **OK**. In the **Create JSF Page** window, we just give our page a filename, and accept all the defaults. Our page opens in JDeveloper's visual editor. At the bottom of the editor are five tabs that can be used to switch the view. The available views are:

- **Design**: This is a sort of What You See Is What You Get editor for JSF pages. Although the rendering looks slightly better than the live preview of Eclipse's Web Page Editor, it is still not perfect. And as with Eclipse, not all tag libraries are able to be rendered in this view. Trinidad renders fine, but unfortunately Tomahawk and Tobago do not render here.
- **Source**: This view shows the source code of the JSPX file.
- **Bindings**: This is a graphical editor to edit the data bindings of this page. This is intended for use with Oracle's ADF Bindings framework. We won't use this framework in this book.
- **Preview**: This view shows the page as it will be displayed in a browser. It does not support editing, though.
- **History**: Every editor in JDeveloper has this tab. It shows the version history of the file.

The JSF components are listed in the **Component palette** that is positioned in the upper-right corner of the screen by default. Components can be dragged-and-dropped from the palette. They can be dropped either on the **Design** view, the **Source** view, or in the **Structure** pane. At the top of the palette is a drop-down box that can be used to select the library.

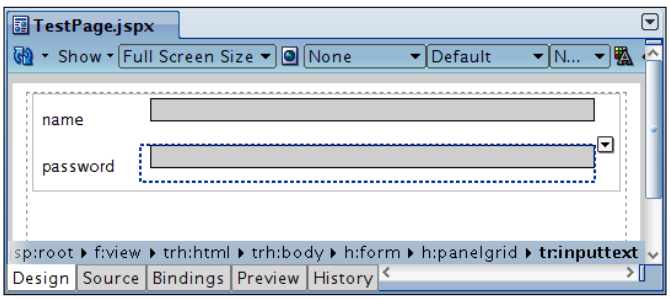
www.it-ebooks.info

The next screenshot shows a simple login page opened in the **Source** view of JDeveloper:

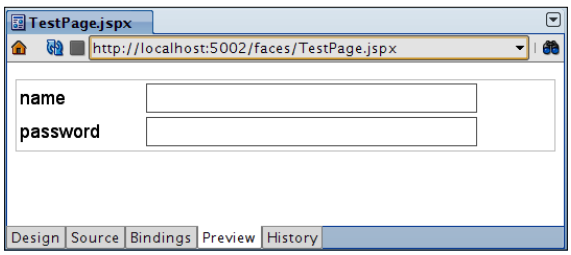


www.it-ebooks.info

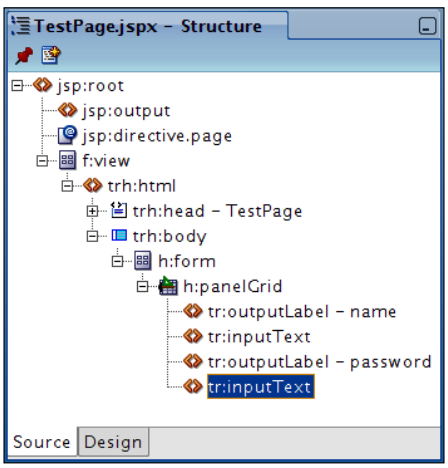
The following screenshot illustrates how the same page shown in the previous screenshot looks in the **Design** view:



The **Preview** view is shown in the following image. As you can see, the page looks more like the way it would be rendered in a browser.



There is also a hierarchical view of the page that shows up in the **Structure** pane, which is located in the lower-left corner of the screen by default. If it doesn't show up, it can be activated through **View | Structure**. The **Structure** pane is shown in the next screenshot:



www.it-ebooks.info

Creating a new project using Maven

The previous subsections described how to create a new project for use with the MyFaces libraries, using two specific IDEs. Using Maven, it is possible to create a new project independent of any IDE. The benefits of using Maven over manually configuring a project in an IDE are:

- The project structure is independent of the IDE used. If you're working in a team, this means that no one is forced to use a certain IDE.
- Maven can automatically perform a lot of otherwise manual steps for us. This can save us some time when creating new projects.
- Maven resolves dependencies automatically. This means that we don't have to manually download the third party libraries that the MyFaces libraries depend upon.

Using Maven to create a project also has some downsides:

- The project structure is independent of the IDE used. This means it is sometimes hard to open a Maven-created project in an IDE that uses a different structure.
- Maven can automatically perform a lot of otherwise manual steps for us. Although this seems attractive, it also means that we don't know much of how our project is set up and this can be a pain when problems occur.
- Maven resolves dependencies automatically. This is handy, unless the Maven repository is down or unreachable due to a firewall or proxy server.
- To be able to set up project structures automatically, Maven uses the concept of **Archetypes**. That means if there isn't an Archetype available for the combination of libraries that we want to use in our project, we'll have to create one ourselves. That might eliminate the benefits.

Well, enough discussion about the pros and cons of Maven. Let's see how we can use it for our MyFaces projects. We assume that Maven is already installed on our system. If not, we can download it from `http://maven.apache.org/download.html`. In most Linux distributions, Maven can be installed through a package manager.

The MyFaces project has created some Maven Archetypes. These can help us to create a blank project with the MyFaces libraries in seconds. We don't have to download anything beforehand. We can just start Maven with the following command:

```
mvn archetype:generate -DarchetypeCatalog=http://myfaces.apache.org
```


www.it-ebooks.info

Maven will then download the Archetypes and all of the required libraries. After that, a list of available Archetypes is presented and we have to choose one. Currently, five Archetypes are defined by the MyFaces project:

- `myfaces-archetype-helloworld`: This will create a simple web application with a "hello world" page that uses MyFaces Core as the JSF implementation, and Tomahawk as the component library.
- `myfaces-archetype-helloworld-facelets`: This will create the same project as the first Archetype, but using Facelets instead of JSPs.
- `myfaces-archetype-helloworld-portlets`: This will create a simple "hello world" project that uses the MyFaces Portlet Bridge library.
- `myfaces-archetype-jsfcomponents`: This archetype can be used to create an empty project as a starting point to create your own JSF component(s).
- `myfaces-archetype-trinidad`: Creates a "hello world" style project that uses Trinidad as the JSF component library.

After choosing a number and pressing *Enter*, Maven asks us for a group ID, artifact ID, version, and package. These are the default values used to create the project. After we confirm these settings, Maven creates the new project for us.

Now we can work with the project, add pages, classes, and so on. When we test our application for the first time, we can also use Maven to compile our classes, and build the WAR file that can be deployed to an application server. We can tell Maven to do so with a single command:

```
mvn package
```

This command should be issued in the directory that Maven created, and that has the name that we entered as the artifact ID.

Application server and configuration files

There are many kinds of application servers, ranging from free and open source products to very expensive commercial products. Today, some high-quality commercial products are also available as open source products, which means that they are free to use if you don't need professional support by the manufacturer. However, most of the time, the choice of the application server is not made by a developer. The choice is often not only based on technical arguments; most of the time, costs and company policies (such as preferred suppliers) do have their influence on such a choice. For that reason, we are not going to spend many pages on different application servers here. We will just focus on the configuration steps that have to be made for virtually every application server when it comes to using Apache MyFaces.

www.it-ebooks.info

You should realize that any Java EE-compliant application server bundles a JSF implementation, because JSF is a part of the Java EE standard. Most application servers bundle the Sun JSF Reference Implementation (RI) for that matter. But, for example, the **Apache Geronimo** application server bundles the MyFaces Core as the default JSF implementation. It is possible to supply an alternative JSF implementation in your application.

There are two important configuration files for a JSF application—`web.xml` and `faces-config.xml`. In the next section, we will discuss the basics of both of them. Sometimes, special configurations have to be made when using a specific JSF library. These specific configurations are discussed in other chapters.

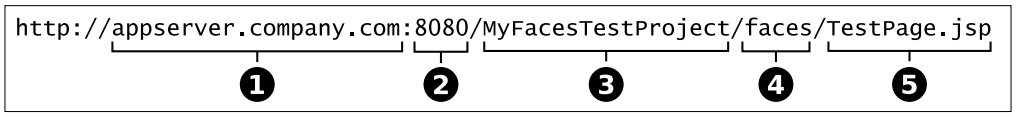
The web.xml configuration file

The `web.xml` configuration file resides in the `/WEB-INF/` directory of the (to be) deployed web application. It configures the web server part of the application. The `web.xml` file can be used to define which file types may be requested by users, which directories can be accessed, and so on. With regards to JSF, the most important task of `web.xml` is to tell the web server that there is such a thing as a **Faces Servlet,** and that URLs containing a certain pattern should be forwarded to that Faces Servlet. A minimal `web.xml` could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance"
         xmlns=
          "http://java.sun.com/xml/ns/javaee"
         xmlns:web=
          "http://java.sun.com/xml/ns/javaee/web-
           app_2_5.xsd"
         xsi:schemaLocation=
          "http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         id="WebApp_ID" version="2.5">
  <display-name>MyFaces Test Project</display-name>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```


www.it-ebooks.info

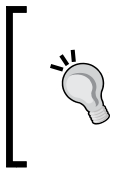
The interesting part of the file is highlighted. The part between the `<servlet>` tags tells the application server that it has to instantiate an object of the `javax.faces.webapp.FacesServlet` class as a Servlet and name it `Faces Servlet`. It will be started at the start up of the application. The part between the `<servlet-mapping>` tags tells the web server that any URL starting with `/faces/` immediately after the address of the server and the location of the application will be handled by that Servlet.



The previous figure shows how the settings made in `web.xml` reflect to the URL of your application. Let's have a look at each part of this URL:

1. The first part of the URL represents the address of our application server. `web.xml` cannot change anything here. This address depends on the domain that our company is using, and the host name of the machine on which the application server is running.
2. The second part of the URL is the port where the application server is listening for requests from users. This is also beyond the influence of `web.xml`. Most application servers let you configure this at the time of installation, and most of the time you can change it through the administration interface of the application server.
3. The third part of the URL is the location of our application, often called the `context root`. This is needed as it is possible to run several applications on the same application server. Most application servers take the name of the WAR file that is used to deploy the application as a default for the context root. Sometimes, it is also possible to configure the context root via an application server specific configuration file and/or via the management interface of the application server.
4. The fourth part of our URL is the URL pattern that is configured in the `web.xml` file. Anything that follows this pattern will be handled by the Faces Servlet.
5. The fifth part of the URL is the name of a page. Note that there is no direct relationship between this name in the URL and a file on the file system of the application server. It is for the Faces Servlet to decide what action to take on this filename. However, most of the time, the Faces Servlet is configured to render a page based on a file with this name that actually resides on the file system of the application server. But it's good to realize that this does not have to be the case all the time.

www.it-ebooks.info



www.it-ebooks.info

Discussing all the details that can be configured through `faces-config.xml` goes beyond the scope of this book. This should be covered by any general JSF book. Settings that are specific to one of the MyFaces libraries will be discussed in the appropriate chapter of this book. For easy reference, we will list the most important top-level elements of the `faces-config.xml` file here:

- `application`: This element is for application-wide settings and definitions. Things such as message bundles and locale configuration go here. In Eclipse, these settings can be found on the **Other** tab of the graphical editor. In JDeveloper, they can be found on the **Overview** tab under **Application**.
- `converter`: Use this element when you've implemented your own `Converter` class, in order to register it with the Faces Servlet. Edit this section on the **Component** tab in Eclipse under **Converters**. In JDeveloper, you can find it on the **Overview** tab.
- `validator`: This element is for registering your own implementation of `Validator`. In Eclipse, you can find it on the **Component** tab under **Converters**, and in JDeveloper you can find it on the **Overview** tab.
- `managed-bean`: Depending on how the backend or model layer of an application is set up, this element is frequently used to manage Java Beans that form a facade or service layer between the view and the model. In Eclipse, there's a **ManagedBean** tab for editing managed beans. In JDeveloper, managed beans can be edited on the **Overview** tab, under **Managed Beans**.
- `navigation-rule`: This is probably one of the most used elements in a typical `faces-config.xml`. It is used to define the navigational structure of the application. When editing navigation rules, you'll be very happy with a graphical editor. You can edit navigation rules on the **Navigation Rule** tab in Eclipse, and on the **Diagram** tab in JDeveloper.
- `render-kit`: This can be used to register a custom `RenderKit` implementation. Some component sets use their own `RenderKit`. In this case, you have to register it here. It can be found on the **Component** tab in Eclipse and on the **Overview** tab in JDeveloper.

www.it-ebooks.info

The following is an example of a small `faces-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                                                  instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/
                                                  javaee
                           http://java.sun.com/xml/ns/javaee/
                                    web-facesconfig_1_2.xsd"
              version="1.2">
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>en_US</supported-locale>
    </locale-config>
     <message-bundle>
                      inc.monsters.mias.Messages
    </message-bundle>
  </application>
  <converter>
    <description>Case converter for text values</description>
    <converter-id>convertCase</converter-id>
    <converter-class>
                 inc.monsters.mias.conversion.CaseConverter
    </converter-class>
  </converter>
  <managed-bean>
    <description>
                A bean to hold the user's preferences
    </description>
    <managed-bean-name>userPreferences</managed-bean-name>
    <managed-bean-class>
                inc.monsters.mias.UserPreferences
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/Login.xhtml</to-view-id>
    </navigation-case>
```


www.it-ebooks.info

```
    <navigation-case>
      <from-outcome>loginError</from-outcome>
      <to-view-id>/LoginError.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
  <validator>
    <validator-id>firstNameValidator</validator-id>
    <validator-class>
              inc.monsters.mias.validators.FirstNameValidator
    </validator-class>
  </validator>
</faces-config>
```

Settings for specific application servers

The example case that will be introduced in the next section is tested on a GlassFish application server. To use MyFaces Core as a JSF implementation on GlassFish, some extra configuration has to be done. Other application servers might need comparable settings to be executed. We'll focus on the extra configuration for GlassFish in the following subsection.

Settings for MyFaces Core on GlassFish

GlassFish comes with the **Mojarra** implementation of the JSF standard. Mojarra is the reference implementation of the JSF standard, and is thus also known as **Sun JSF RI**. As discussed in Chapter 1, it is not necessary to use MyFaces Core instead of Mojarra, as both implement the same JSF standard. However, as MyFaces Core provides more diagnostic and debug information in the server log files, it might be worth using MyFaces Core.

To use MyFaces Core as the JSF implementation for our application, we have to make some additional settings in a GlassFish-specific configuration file—`sun-web.xml`. This file has to be in the `WEB-INF` folder of our project, along with most of the other configuration files, such as `web.xml` and `faces-config.xml`. The contents of the file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
                     Application Server 8.1 Servlet 2.4//EN"
                     "http://www.sun.com/software/appserver/
                     dtds/sun-web-app_2_4-1.dtd">
<sun-web-app>
  <class-loader delegate="false"/>
  <property name="useMyFaces" value="true"/>
</sun-web-app>
```


www.it-ebooks.info

The highlighted lines are the lines that disable the default implementation and force GlassFish to use MyFaces Core instead. Of course, we have to make sure that the MyFaces Core libraries are added properly to our application and configured correctly, as described in the previous sections.

Other application servers

If you want to use another application server with MyFaces Core as a JSF implementation, you should refer to the documentation of that application server. Most servers have a vendor-specific configuration file named `vendor-web.xml`, such as the `sun-web.xml` file for GlassFish. The Apache Geronimo application server uses MyFaces Core as the default JSF implementation, so no additional configuration is needed for that one.

Introduction to the example case

Throughout this book, we will be building example JSF pages to test the possibilities of the various Apache MyFaces Core libraries. To eliminate the need to come up with a new example every time, we are going to work on a single case in the rest of the book. This section introduces the case.

We are building a new administrative system for our client, Monsters, Inc. (You might know this company from the movie with the same name.) Monsters, Inc. is the power company of Monstropolis. The company generates power by scaring children and collecting their screams. The employees of Monsters, Inc. get to the children by means of teleportation doors that are set up on the work floor. The company wants a better knowledge of the performance of its employees.

Therefore, every scarer should log his activities in the new system. Every time a child is scared, the scarer should log:

- At which time the scaring started
- How much time it took to scare the child and collect the scream
- How much energy was produced
- Through which door he reached the child
- How brave the kid was on the braveness scale
- Who was the scare assistant

www.it-ebooks.info

The floor manager must have the possibility to get a list of top scarers on a daily, weekly, and monthly basis. He or she also needs an overview of the total generated energy daily, weekly, and monthly. The system is also used by the Human Resources department to store personal information of all employees, and to keep track of their salaries and bonuses. Floor managers can assign bonuses to scarers that perform better than expected.

Throughout the rest of this book, we will build parts of the administrative system that will be used by the company to measure the performance of their employees.

Summary

In this chapter we had a look at the possibilities of two IDEs (Eclipse and JDeveloper) with regards to creating JSF applications that use Apache MyFaces components. We learned how to configure both IDEs to work with MyFaces. We saw that both have the capability to edit JSP files in a graphical way. We also saw that, unfortunately, not all component libraries are supported by those graphical editors. And we also saw how we can set up a project that uses MyFaces with Maven. In the second section of this chapter, we learned about the purpose of the `web.xml` and `faces-config.xml` configuration files. The third section introduced a business case that will be used throughout the rest of the book to base the examples on.

The next chapter will introduce Facelets, the view technology that we'll be using instead of JSP in the rest of the book.

www.it-ebooks.info

3
Facelets

One of the strong properties of JSF is the fact that it has the ability to use various **view technologies**. The default view technology in JSF 1.x is **JavaServer Pages (JSP)**. Because the JSP standard was defined before JSF even existed, it was never optimized for use with JSF. Therefore, JSP has quite a few shortcomings as a view technology for JSF. **Facelets** aims to overcome these shortcomings. Although Facelets is not an official standard yet, it doesn't make sense not to use it. In JSF 2.0, Facelets is part of the standard and is the preferred view technology.

Facelets is not part of Apache MyFaces. This chapter is included in this book as not much documentation is available on Facelets yet.

After reading this chapter, you should be able to:

- Convince anyone that Facelets should be used as the view technology for any new JSF project
- Set up a JSF project with Facelets
- Create and use Facelets page templates
- Create and use you own composition components with Facelets

Why Facelets?

Facelets has a lot of improvements over JSP as a view technology for JSF. This section provides an overview of the most important improvements of Facelets over JSP, starting with content interweaving, in the next subsection.

www.it-ebooks.info

Content inverweaving

One of the things that makes the combination of JSF with JSP complicated is the problem of **content interweaving**. Although the situation is somewhat improved since JSF 1.2 and JSP 2.1, both technologies still create their own representation of a page in memory. This is not only inefficient, but can also generate problems if JSF components are mixed with non-JSF content. (Think of problems with regards to the order in which elements are shown in the rendered page, and for user interface elements that are not aware of each other.) For that reason, it is advisable not to make such mix-ups when using JSP.

Facelets fixes this issue by creating a single component tree. This tree contains both JSF components and non-JSF elements such as plain text and XHTML. Because of this, it is possible to mix XHTML markup with JSF components and use **Expression Language** in all elements, even in plain text. This means we can use each of them for the tasks they're good at. XHTML is perfect for creating a nice layout, while most JSF components specialize in user interaction and data entry. It also means that we can let a web designer design a plain XHTML page and simply add our JSF components to that page. A nice example of the benefits of Facelets' content interweaving solution is given in the *Using inline texts* section of this chapter.

Templating

JSP has no built-in templating system, and neither has JSF. In plain JSP (without JSF), the lack of a templating system could be compensated by using `<jsp:include>` component. It wasn't a perfect solution, but it did the job. With JSF, the inclusion of JSP (fragments) is no longer an option. (It is possible to use the `<jsp:include>` component with the JSF `<f:subview>` component, but that's still not a real templating system.) Some JSF component libraries try to fill this gap by introducing components that act as a kind of template. For example, Trinidad has a `<tr:panelPage>` component that divides the page in several areas for content, navigation, and branding. This only solves part of the problem. For example, although such a component helps to make the page layout consistent, we still have to add the navigation component(s) to every page. Another downside of such a component is that it isn't very flexible—we don't have much influence over the positioning of the items on the page.

So, it is clear that we need a proper templating solution for JSF. We must be able to define a layout that is applied to every page. Also, should we ever want to change the layout for all pages, it should not be necessary to edit all the pages. Facelets gives us such a solution; see the *Templating with Facelets* section of this chapter for details.

www.it-ebooks.info

Don't Repeat Yourself (DRY)

Don't Repeat Yourself (DRY) is one of the most important paradigms in programming. Code that isn't DRY is harder to maintain and more sensitive to bugs. Unfortunately, JSF with JSP makes it very hard not to repeat ourselves, in a number of ways. For instance, the lack of a templating solution can lead to repeated navigation and layout code on every page.

But on a smaller scale, there are unnecessary repetitions too. In a form, for example, you have to add a separate label and text field for every field. If we have our label texts in a resource bundle and we use the (database) field names as keys in that bundle, we only need one key (the field name) for both the label and the field. Wouldn't it be nice if we could compose our own component that ties together a label and a field based on that single key? Facelets gives us the opportunity to do so! See the section *Creating and using composition components* for details. (Some component libraries solve this problem by adding their own composition components. This can be a good solution in many cases, but being able to compose our own components gives us far more flexibility.)

Expanding the Expression Language

JSF defines a very useful Expression Language (EL). With this EL, it is easy to refer to the properties of JavaBeans; but sometimes we will want to execute a function. This is not easy to achieve in the standard JSF EL. Facelets gives us the possibility to register any static Java method that is to be called from within an EL expression. An example will be given in the *Using Static Functions* section of this chapter.

Summarizing the benefits of Facelets

As we have seen in this section, there are many rather fundamental reasons why Facelets is a better option than JSP for the view technology in our JSF project. Some of the problems with JSP can be solved by components. However, that is not always the best way of solving these problems, and often leads to a sub-optimal solution. With Facelets, most of JSP's shortcomings can be overcome in a more fundamental way, and without the need for a special component set. So let's see how we can set up our project to use Facelets, in the next section.

www.it-ebooks.info

Setting up a Facelets project

Of course, installing Facelets starts with downloading it. The Facelets library can be downloaded from `https://facelets.dev.java.net/`. The latest release can be found through **Documents & files** in the **Project tools** menu. Stable releases are under the **releases** node of the tree menu. Unzip the downloaded file to the location where all other JSF libraries are stored.

Now that we have the Facelets library, we need to make sure that it will be in the `WEB-INF/lib` directory of our web application. We can follow the same procedure as described in Chapter 2, where we added the other JSF libraries. Make sure that you select the appropriate option to force the Facelets library to be included on deployment. Only the `jsf-facelets.jar` file is needed, which is in the root of the Facelets distribution.

Preparing web.xml



www.it-ebooks.info

Preparing faces-config.xml

This is the part where we really activate Facelets. We tell JSF to use an alternative view handler. This means that the **render response** and **restore view** phases will be executed by this alternative view handler, which is Facelets in our case. Defining an alternative view handler is done by adding the following code snippet to the `faces-config.xml` file:

```
<application>
  <view-handler>
     com.sun.facelets.FaceletViewHandler
  </view-handler>
</application>
```

If there is already an `application` section in the `faces-config.xml`, we could simply add the highlighted part of the above code snippet to that section.

Creating a test page

Now let's test if the Facelets installation has succeeded. Let's create a standard XHTML page. It will look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Insert title here</title>
</head>
<body>
</body>
</html>
```

The first thing we have to do is expand the XML namespace of our document to include both the Facelets and JSF namespaces. This will change the `<html>` tag to something like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
```

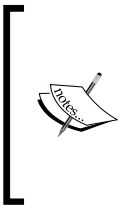

www.it-ebooks.info

Note that the default prefix for the Facelets namespace is `ui`.

Now, as we have extended our namespace, we can add some components. Let's create a simple login page for the MIAS system. It could look as follows: (The `<?xml>` and `<!DOCTYPE>` declarations are omitted for brevity.)

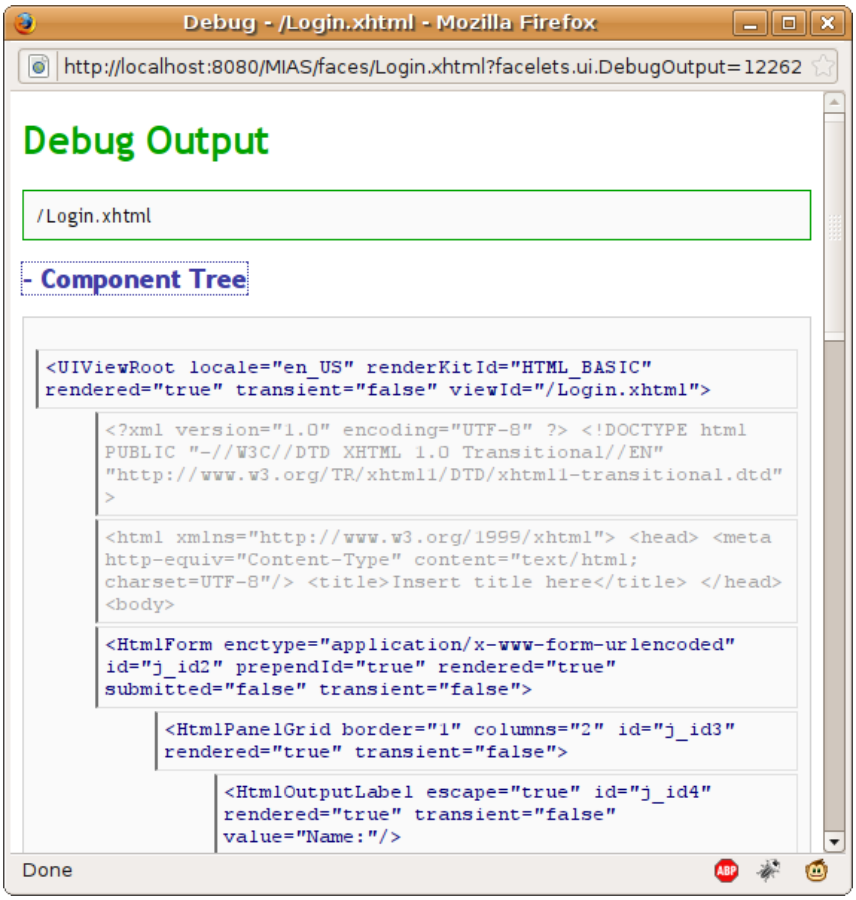
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
  <f:loadBundle basename="inc.monsters.mias.Messages"
                var="msg"/>
  <title>Login</title>
</head>
<body>
  <h:form id="login">
    <h:panelGrid columns="4">
      <h:outputLabel value="#{msg.userName}" />
      <h:inputText value="#{loginBean.userName}"
                   id="userName" required="true">
        <f:validateLength minimum="6"/>
      </h:inputText>
      <h:outputText value="*"/>
      <h:message for="userName"/>
      <h:outputLabel value="#{msg.password}" />
      <h:inputSecret value="#{loginBean.password}"
                     id="password" required="true">
        <f:validateLength minimum="3"/>
      </h:inputSecret>
      <h:outputText value="*"/>
      <h:message for="password"/>
      <h:outputLabel value="" />
      <h:commandButton value="OK"
                       action="#{loginBean.login}" />
      <h:outputLabel value="" />
      <h:outputLabel value="" />
    </h:panelGrid>
  </h:form>
</body>
</html>
```


www.it-ebooks.info



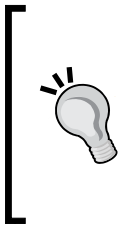
www.it-ebooks.info

Now when we run our page, we can activate the debug window by pressing *CTRL* + *SHIFT* + *D*. The pop up might be blocked by the browser's pop-up killer. It might be a good idea to disable the pop-up killer for the domain in which your server is running (in a development environment, this is typically `localhost`). An example of a debug window is shown in the next screenshot:



The window has two expandable sections. The first section shows the component tree that was used to render the page. The second section shows all JSF-scoped variables, separated up per scope.

www.it-ebooks.info



www.it-ebooks.info

```
    <title>
      <ui:insert name="title">** NO TITLE SET **
      </ui:insert>
    </title>
  </head>
  <body>
    <div id="header">
      <ui:insert name="header">
        <ui:include src="header.xhtml" />
      </ui:insert>
    </div>
    <div id="content">
      <h2><ui:insert name="title" /></h2>
      <ui:insert name="content" />
    </div>
    <div id="footer">
      <ui:insert name="footer">
        <ui:include src="footer.xhtml" />
      </ui:insert>
    </div>
  </body>
  </html>
```

Note how we can reuse a defined value. The `title` is used both in the `<head>` and `content` sections. We also added an `<f:loadBundle>` component to load our message bundle, so we don't have to worry about that in the individual page definitions.

Using the template

Now we will use the template that we created. Let's refactor our login page to use the template. To use a template, we can use the `<ui:composition>` tag. If we saved our template with the name `template.xhtml`, then we can use it by adding the following line to our login page:

```
<ui:composition template="template.xhtml">
```

Now Facelets will apply the template to our page. But we still have to let Facelets know what values to insert in the placeholders. This can be done by using the `<ui:define>` tag. To define a title for our page, we could add the following to our page:

```
<ui:define name="title">Login</ui:define>
```


www.it-ebooks.info

We can also nest a whole tree of components in the `<ui:define>` section, as shown in the following code snippet:

```
<ui:define name="content">
  <h:form id="login">
    <h:panelGrid columns="4">
      <h:outputLabel value="#{msg.userName}" />
      <h:inputText value="#{loginBean.userName}"
                   id="userName" required="true">
        <f:validateLength minimum="6"/>
      </h:inputText>
      <h:outputText value="*"/>
      <h:message for="userName"/>
      <h:outputLabel value="#{msg.password}" />
      <h:inputSecret value="#{loginBean.password}"
                    id="password" required="true">
        <f:validateLength minimum="3"/>
      </h:inputSecret>
      <h:outputText value="*"/>
      <h:message for="password"/>
      <h:outputLabel value="" />
      <h:commandButton value="OK"
                       action="#{loginBean.login}" />
    </h:panelGrid>
  </h:form>
</ui:define>
```

In this code snippet, we simply pasted all of the content of the original login page into the appropriate `<ui:define>` section. The whole page should now look as listed here. (Again, the `<?xml>` and `<!DOCTYPE>` declarations are omitted for brevity.)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<body>
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Login</ui:define>
  <ui:define name="content">
    <h:form id="login">
      <h:panelGrid columns="4">
        <h:outputLabel value="#{msg.userName}" />
        <h:inputText value="#{loginBean.userName}"
                     id="userName" required="true">
          <f:validateLength minimum="6"/>
```


www.it-ebooks.info



www.it-ebooks.info

```
    </ui:define>
</ui:composition>
```

Although this is a valid XML and works fine with Facelets, it might be harder to edit or preview with (X)HTML-oriented editors. However, it has the advantage of being a bit shorter, and will prevent confusion about elements that are in the file but not rendered because they're outside the `<ui:composition>` element.

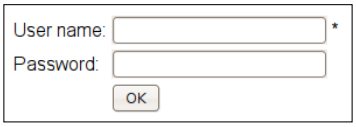
Using comments in Facelets page definitions

It's a good habit to add comments to your source code files. In the examples presented here, comments are left out to save space, and because an explanation is given in the text anyway. But if you write your own Facelets page definitions, it's a good idea to add comments where applicable. That said, some caution is required in regard to comments in Facelets. Facelets adds comments to the JSF component tree just as it does with XHTML elements, plain text, and JSF components. This can cause unexpected behavior sometimes. Consider this code snippet:

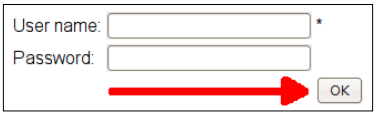
```
<h:panelGrid columns="4">
  <h:outputLabel value="#{msg.userName}" />
  <h:inputText value="#{loginBean.userName}"
               id="userName" required="true">
    <f:validateLength minimum="6"/>
  </h:inputText>
  <h:outputText value="*"/>
  <h:message for="userName"/>
  <h:outputLabel value="#{msg.password}" />
  <h:inputSecret value="#{loginBean.password}"
                 id="password" required="true">
    <f:validateLength minimum="3"/>
  </h:inputSecret>
  <h:outputText value="*"/>
  <h:message for="password"/>
  <!-- This empty label is added to position the
       OK button aligned with the input fields -->
  <h:outputLabel value="" />
  <h:commandButton value="OK"
                   action="#{loginBean.login}" />
</h:panelGrid>
```


www.it-ebooks.info

We would expect it to render as shown in the next figure:



But as the comment is treated like a component, `<h:panelGrid>` reserves a cell for it. The result is shown in the next image:



Further proof of this can be found in the debug window. As shown in the next screenshot, the component tree shows our comment as a separate component.



www.it-ebooks.info

Now, how do we prevent this from happening without having to remove all of the comments? The only thing we have to do is to set an extra context parameter in our `web.xml` file, as shown in the following code snippet:

```
<context-param>
  <param-name>facelets.SKIP_COMMENTS</param-name>
  <param-value>true</param-value>
</context-param>
```

This will force Facelets to simply skip all comments in the page definition files. As the default value of this context parameter is `false`, comments are not skipped by default.

Are Facelets files XHTML?

Although the files are saved with the `.xhtml` extension, Facelets files are clearly not really XHTML files. They're just XML files that Facelets parses to XHTML that can be displayed in a browser. But it is convenient to handle them as XHTML most of the time. For example, if you have a web designer in your team, he or she can simply create a template as a valid XHTML file. You can either add the `<ui:insert>` tags yourself, or tell him or her how to do it themselves. However, there are some limitations to this because of the fact that the resulting Facelets pages and templates are not real XHTML.

In (X)HTML, a huge list of **named entities** is defined. This is frequently used to insert special characters in an XHTML document. For example, ` ` is frequently used to insert a "non-breaking space", and `©` is used to insert a © sign. As Facelets pages are not XHTML, these entities cannot be used. Only the named entities that are defined as part of the XML standard can be used. The XML-named entities are listed in the table that we are about to see. As an alternative to named entities, numbered entities can be used. Appendix A contains a list of all HTML numbered entities.

Named entity	Symbol	Number
`&`	&	`&`
`<`	<	`<`
`>`	>	`>`
`'`	'	`'`
`"`	"	`"`

www.it-ebooks.info

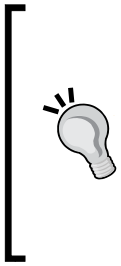
An alternative to using XML entities is to use Unicode characters. In this case, you should make sure that the page definitions are stored in a UTF-8 file format and that the application server is configured to use UTF-8. The browser client should also be able to receive UTF-8 pages. In short, using Unicode characters instead of entities increases the risk of special characters getting "lost in translation".

Creating and using composition components

At first, you might be wondering why you should ever create your own components. Especially when you look at those extensive sets of readymade components in the Trinidad, Tobago, and Tomahawk project, you might wonder why there is still a need to create your own composition components. The reason is simple—no one but you can create a component that exactly fits your needs and application. This means that you often have to combine different components, validators, and converters.

Combining is not a problem, but what if you have to make the same combination of components a couple of times? Then you're breaking the DRY principle; you are repeating yourself! We will look at the login page of the MIAS system as an example. As this is a rather simple page with a very limited number of fields, the benefits of using composition components are perhaps not that clear to see in this example. But imagine if the page is twice as large and you have dozens of pages like that in your project. In this case, you can imagine that a lot of repetition can be prevented by using composition components.

Creating a tag library



www.it-ebooks.info

Let's name our tag library definition file `mias.taglib.xml`. An empty tag library definition looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "facelet-taglib_1_0.dtd">
<facelet-taglib>
  <namespace>http://www.monsters.inc/</namespace>
</facelet-taglib>
```

The highlighted line defines the XML namespace. Although this is a URL, it does not have to point to an existing website. It is only used as a unique identifier by which we can refer to this tag library (just like packages in Java). Now let's add a component to the library:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "facelet-taglib_1_0.dtd">
<facelet-taglib>
  <namespace>http://www.monsters.inc/</namespace>
  <tag>
    <tag-name>field</tag-name>
    <source>field.xhtml</source>
  </tag>
</facelet-taglib>
```

The highlighted lines show a new tag definition. The tag has only a `tag-name` and a `source`. The `source` must be the filename of an existing file; this file contains the definition of the composition component. The `tag-name` is the name of the tag that is used in the Facelets page definitions. As with normal component libraries, we will assign a short prefix to our namespace. In this way, we are able to add tags such as `<prefix.tag-name />` to our page definitions, as we will see next.

Once we have defined our tag library, we have to let Facelets know that there is a tag library, and where this can be found. This can be done by adding some lines to our `web.xml` file:

```
<context-param>
  <param-name>facelets.LIBRARIES</param-name>
  <param-value>/WEB-INF/tags/mias.taglib.xml</param-value>
</context-param>
```


www.it-ebooks.info

Adding these lines to our `web.xml` file will tell Facelets that there is a tag library defined by a file that can be found at `/WEB-INF/tags/mias.taglib.xml`. Note that the file is placed in a subdirectory called `tags`. It's a good idea to keep tag (library) definitions separate from "ordinary" pages.

Creating the composition component itself

Now comes the interesting part. Let's create our own component. The next subsections discuss the steps involved in doing this.

Identifying redundancies

To determine how our component should look, let's take a closer look at a part of the login page that we created in the *Templating* section:

```
<h:outputLabel value="#{msg.userName}" />
<h:inputText value="#{loginBean.userName}"
id="userName" required="true">
<f:validateLength minimum="6"/>
</h:inputText>
<h:outputText value="*"/>
<h:message for="userName"/>
```

We can see that five components are used for one single input field:

- `<h:outputLabel>`: To render the label for the field
- `<h:inputText>`: To render the input field itself
- `<f:validateLength>`: To set a minimum length
- `<h:outputText>`: To render an asterisk (*) to indicate that the field is required
- `<h:message>`: To render possible error messages

Note that a lot of redundant information is present:

- The fact that the field is required is reflected by the `required="true"` attribute in the `<h:inputText>` component, and by the extra `<h:outputText>` component
- The name or ID of the component (`"userName"`) is used four times in this code snippet

www.it-ebooks.info

We want a component that does exactly the same as the six JSF components in the code snippet that we just saw, but requires only one line of code in our page, such as this:

```
<mias:field id="userName" required="true"
            bean="loginBean" min="6"/>
```

Creating a skeleton for the composition component

To create our composition component, we simply create an XHTML file, as we did for our page definitions. As with a page definition, a composition component starts with a `<ui:composition>` tag. So, an empty component looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:mias="http://www.monsters.inc/mias">
<ui:composition>
<!-- component definition goes here -->
</ui:composition>
</html>
```

Note that everything outside the `<ui:composition>` tag will be ignored by Facelets, but is necessary for the page to be valid XHTML. Apart from the `ui`, `h`, and `f` namespaces, we also include the `c` and `fn` namespaces. Facelets allows us to use the **JSP Standard Tag Library (JSTL)** for some scripting-like features. If we want to use the JSTL, we have to include its namespaces. Our own namespace (`mias`) is included to enable the nesting of our own components.

www.it-ebooks.info

Defining the actual composition component

Now let's build the actual definition of the component. As mentioned earlier, we would like a `required` attribute that sets the input component's `required` attribute and renders an asterisk (*) next to the input box. It would be nice if we could leave out the `required` attribute if an input value is not required, saving some extra typing. This can be accomplished as follows:

```
<c:if test="#{empty required}">
  <c:set var="required" value="false" />
</c:if>
```

We use the `<c:if>` function from the JSTL to test if the `required` attribute is set. If it is not set, we use `<c:set>` to set it to the default value—`false`. Now we can use the `required` variable throughout the rest of our component without having to worry about whether it is set or not.

The next thing to do is to add the label for our input field. That's pretty straightforward:

```
<h:outputLabel value="#{msg[id]}" />
```

We simply add an `<h:outputLabel>` component with the correct value. For the value, we assume that a message bundle is bound to the `msg` variable. In this case, this is a pretty safe assumption as the component will only be used in our own application and have we already added a `<f:loadBundle>` component to our template. By using the `[]` operator instead of the `.` (dot) operator, the value of `id` will be substituted. If we had used the dot operator, `id` would have been interpreted as a string literal. So if we make sure that the message bundle contains a key that is identical to the `id` of our component, a nice text label will be printed next to the input field.

Now we have to add the input field itself. This is a little bit more complex. As we want the "password behavior" as an optional feature of our component, we'll have to add either `<h:inputText>` or `<h:inputSecret>`, depending on the `secret` variable. We can use the `<c:choose>` statement of the JSTL, as shown in the following code snippet:

```
<c:choose>
  <c:when test="#{secret}">
    <h:inputSecret value="#{bean[id]}" id="#{id}"
                   required="#{required}" />
  </c:when>
  <c:otherwise>
    <h:inputText value="#{bean[id]}" id="#{id}"
                 required="#{required}" />
  </c:otherwise>
</c:choose>
```


www.it-ebooks.info

If the value of `secret` evaluates to `true`, we use an `<h:inputSecret>` component. In all other cases (including the case that `secret` is not defined at all), we use an `<h:inputText>` component. To get the value from the bean, we use the `[]` operator, as we did with the message bundle.

Adding validators without violating the DRY principle

We're not finished with our input field yet. We still have to add the length validators, depending on the values of `minLength` and `maxLength`. The difficulty is that we don't want to add a validator if no minimum or maximum length is set. If only a minimum limit is set, we don't want to set a maximum limit, and vice versa. To accomplish this, we could use another `<c:choose>` as follows:

```
<c:choose>
  <c:when test="#{not empty minLength
                  and empty maxLength}">
    <f:validateLength minimum='#{minLength}'/>
  </c:when>
  <c:when test="#{empty minLength
                  and not empty maxLength}">
    <f:validateLength maximum='#{maxLength}'/>
  </c:when>
  <c:when test="#{not empty minLength
                  and not empty maxLength}">
    <f:validateLength minimum='#{minLength}'
                      maximum='#{maxLength}'/>
  </c:when>
</c:choose>
```

However, it would be against the DRY principle to add this lengthy code block inside both the `<h:inputSecret>` and `<h:inputText>` tags. To avoid this redundancy, we create another composite component and call it `lengthValidator`. This component contains the code that we just saw, nested within a `<ui:composition>` tag. We can now use this component within our input fields:

```
<c:choose>
  <c:when test="#{secret}">
    <h:inputSecret value="#{bean[id]}" id="#{id}"
                   required="#{required}">
      <mias:lengthValidator minLength="#{minLength}"
                            maxLength="#{maxLength}"/>
    </h:inputSecret>
  </c:when>
```


www.it-ebooks.info

```
    <c:otherwise>
      <h:inputText value="#{bean[id]}" id="#{id}"
                   required="#{required}">
        <mias:lengthValidator minLength="#{minLength}"
                              maxLength="#{maxLength}"/>
      </h:inputText>
    </c:otherwise>
  </c:choose>
```

Note how the values of `minLength` and `maxLength` are passed through. Also note that if `minLength` and `maxLength` are both not set, `<mias:lengthValidator>` will not render any component at all.

Putting it all together

We are nearly finished now. We only have to add the required indicator and the `<h:message>` component. That's straightforward:

```
    <h:outputText value="#{required ? '*' : ' '}"/>
    <h:message for="#{id}"/>
```

We use the JSTL conditional operator to render an asterisk if `required` evaluates to `true`, and a space otherwise. We could have chosen to not render anything at all if `required` was not true, but that could have caused problems with the `<h:panelGrid>` component. To summarize, the complete component now looks like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:mias="http://www.monsters.inc/mias">
<ui:composition>
  <c:if test="#{empty required}">
    <c:set var="required" value="false" />
  </c:if>
  <h:outputLabel value="#{msg[id]}" />
  <c:choose>
    <c:when test="#{secret}">
      <h:inputSecret value="#{bean[id]}" id="#{id}"
                     required="#{required}">
        <mias:lengthValidator minLength="#{minLength}"
                              maxLength="#{maxLength}" />
```


www.it-ebooks.info

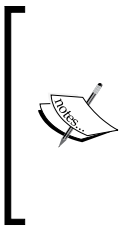
```
        </h:inputSecret>
      </c:when>
      <c:otherwise>
        <h:inputText value="#{bean[id]}" id="#{id}"
                     required="#{required}">
          <mias:lengthValidator minLength="#{minLength}"
                                maxLength="#{maxLength}" />
        </h:inputText>
      </c:otherwise>
    </c:choose>
    <h:outputText value="#{required ? '*' : ' '}"/>
    <h:message for="#{id}"/>
  </ui:composition>
  </html>
```

The `lengthValidator` component is defined as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:mias="http://www.monsters.inc/mias">
<ui:composition>
  <c:choose>
    <c:when test="#{not empty minLength
                    and empty maxLength}">
      <f:validateLength minimum='#{minLength}'/>
    </c:when>
    <c:when test="#{empty minLength
                    and not empty maxLength}">
      <f:validateLength maximum='#{maxLength}'/>
    </c:when>
    <c:when test="#{not empty minLength
                    and not empty maxLength}">
      <f:validateLength minimum='#{minLength}'
                        maximum='#{maxLength}'/>
    </c:when>
  </c:choose>
</ui:composition>
</html>
```


www.it-ebooks.info

Using the composition component



www.it-ebooks.info

Using static functions

Another powerful feature of Facelets is the possibility to create static functions. Static functions let you expand the possibilities of the JSF Expression Language. You can, in fact, configure any static Java function that is on the classpath to be available as a static function in the Expression Language. This is fairly simple to do.

Let's assume that the CEO of Monsters, Inc—Mr. Waternoose—wants to add an inspirational quote of the day to the login page of the MIAS system. Let's see how we can implement this by using the Facelets static function feature. First, we need a static method that returns a quote. As we want to display the author of the quote as well, let's create two functions, as follows:

```
package inc.monsters.mias;

public class FaceletsFunctions {
   private static Quote lastQuote;
   private static Quotes quotesList;

   private static Quotes getQuotes() {
      if (quotesList == null) {
         // fill the quotes list from a file or something
      }
      return quotesList;
   }

   public static String getQuoteOfTheDay() {
      lastQuote = getQuotes().getRandomQuote();
      return lastQuote.getText();
   }

   public static String getAuthorOfTheDay() {
      if (null == lastQuote) {
         return "";
      } else {
         return lastQuote.getAuthor();
      }
   }
}
```

We assume that there is a `Quotes` class that can return a random `Quote` object. A `Quote` object has both a `getText()` method and a `getAuthor()` method. The full source code of this example is available for download at my website (`http://www.bartkummel.net`) and contains the full implementation, including the code to read the quotes from an XML file.

www.it-ebooks.info

Next, we have to define the highlighted methods as static EL functions. This can be done in the `mias.taglib.xml` file that we created in the previous section:

```
<function>
  <function-name>getQuoteOfTheDay</function-name>
  <function-class>
      inc.monsters.mias.FaceletsFunctions
  </function-class>
  <function-signature>
      java.lang.String getQuoteOfTheDay()
  </function-signature>
</function>
<function>
  <function-name>getAuthorOfTheDay</function-name>
  <function-class>
      inc.monsters.mias.FaceletsFunctions
  </function-class>
  <function-signature>
      java.lang.String getAuthorOfTheDay()
  </function-signature>
</function>
```

Note that all class names—including the class names of classes from the Java API—are fully qualified class names. In this example, the methods do not accept any arguments, but it is possible to accept arguments–simply put the types of the arguments in `function-signature`. Remember to use fully qualified names, unless you're using primitive types.

There isn't much more to discuss about defining static functions. Let's have a look instead at how we can use them. We can now add the quote of the day to the login page, as shown in the following example:

```
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Login</ui:define>
  <ui:define name="content">
    <h:form id="login">
      <h:panelGrid columns="4">
        <mias:field id="userName" bean="#{loginBean}"
                    required="true" minLength="6" />
        <mias:field id="password" bean="#{loginBean}"
                    required="false" minLength="3"
                    secret="true" />
        <h:outputLabel value="" />
        <h:commandButton value="OK"
                         action="#{loginBean.login}" />
```


www.it-ebooks.info

```
        </h:panelGrid>
      </h:form>
      <i>
        <h:outputText value="#{mias:getQuoteOfTheDay()}" />
      </i>
      <h:outputText value="#{mias:getAuthorOfTheDay()}" />
    </ui:define>
  </ui:composition>
```

As you can see, we can now call our static methods as we would call any other Expression Language function. This feature can be used in various ways. A common use is to create a function that checks whether a user has a certain role. By returning a `boolean`, such a method can be used to enable or disable certain components. Static functions can also be used for various internationalization tasks, or to create all sorts of functions that you are missing in the standard Expression Language.

Using inline texts

Facelets is, in a lot of ways, more "relaxed" than JSP as a view technology for JSF. One of the ways is the way we can use inline texts. With JSF, you always need a component to render text on the output. This can make our page definitions more complicated than necessary. Facelets will keep the XHTML tags that we mix with our JSF components, and will also retain any plain texts. (This is called content interweaving.) This means that we no longer need JSF components for rendering static elements on a page. We can even incorporate some dynamic or internationalized texts, because Facelets allows us to use Expression Language within inline texts. This means, the quote of the day example from the previous section can be rewritten in a more elegant way, as follows:

```
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Login</ui:define>
  <ui:define name="content">
    <h:form id="login">
      <h:panelGrid columns="4">
        <mias:field id="userName" bean="#{loginBean}"
                    required="true" minLength="6" />
        <mias:field id="password" bean="#{loginBean}"
                    required="false" minLength="3"
                    secret="true" />
        <h:outputLabel value="" />
        <h:commandButton value="OK"
                         action="#{loginBean.login}" />
      </h:panelGrid>
```


www.it-ebooks.info

```
        </h:form>
        <i>#{mias:getQuoteOfTheDay()}</i>
        #{mias:getAuthorOfTheDay()}
      </ui:define>
    </ui:composition>
```

Of course, this feature should be used wisely. For most dynamic content, a JSF component (or, of course, a composition component) is the right answer. But there are cases, as we just saw, where some dynamically-generated inline text is more elegant.

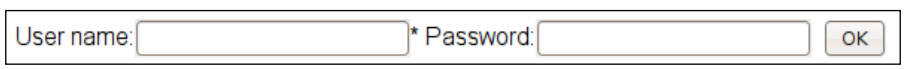
Facelets tags overview

This section provides an overview of all the tags in the Facelets namespace. This is a useful reference. Some tags are not covered in the previous sections. You don't need these (not previously covered) tags to get started with Facelets, but they may come in handy for advanced users. These tags are given a slightly longer description here.

<ui:component> tag

The `<ui:component>` tag is nearly the same as the `<ui:composition>` tag. It can be used to make a composition component out of several JSF components, as discussed in the *Creating and using composition components* section of this chapter. The difference between the `<ui:component>` tag and the `<ui:composition>` tag is that the former adds the composed components as one single component to the JSF component tree, whereas the latter adds every component of the composition to the tree separately.

This can be used in combination with components that depend on the number of child components that they have, such as the `<h:panelGrid>` component. If you compose a composition component out of three components, these components will get a separate cell each in the generated grid if we use `<ui:composition>` tag. But if we use `<ui:component>` tag instead, the three components will be added to a single cell. If we replace `<ui:composition>` tag with `<ui:component>` tag in our `field.xml` and render the login page without changing it, it will look like the following image:



The reason for this is that the label, input field, required indicator, and message component have been added as a single component to the JSF component tree. This means that the `<h:panelGrid>` component counts them as one component. Of course, this isn't very useful in this example, but it can be useful in some cases.

www.it-ebooks.info

As the `<ui:component>` tag adds its children as a component to the JSF component tree, the component can be bound to a backing bean by using the `binding` attribute. This means that we can access the composition component from within our Java code, just as any other JSF component.

The following are the attributes of the `<ui:component>` tag:

- `template`: The URI where the template can be found. For example, `templates/template.xhtml`.
- `binding`: An optional bean variable to which the generated component will be bound.

<ui:composition> tag

The `<ui:composition>` tag can be used to create a composition component out of multiple JSF components. There can only be one `<ui:composition>` tag in a given file. Everything that is outside the `<ui:composition>` tag will be ignored by Facelets. More information, and an example, can be found in the *Creating and using composition components* section of this chapter.

The following is the only attribute of `<ui:composition>` tag:

- `template`: The URI where the template can be found. For example, `templates/template.xhtml`.

<ui:debug> tag

The `<ui:debug>` tag can be used to enable the debug feature of Facelets and assign a shortcut key to it. Adding the following code to any page will enable the debug feature on that page:

```
<ui:debug hotkey="d" rendered="true"/>
```

In this example, the shortcut key for activating the debug window is *CTRL* + *SHIFT* + *D*. Any other letter can be used. If the `rendered` attribute is set to `false`, the debug feature will be disabled. This is useful if you want to dynamically enable or disable the debug feature with Expression Language.

The following are the attributes of `<ui:debug>` tag:

- `hotkey`: Defines which key to use (in combination with *CTRL* and *SHIFT*) for activating the debug window
- `rendered`: Can be set to `false` to disable the debug functionality

www.it-ebooks.info

<ui:decorate> tag

The `<ui:decorate>` tag is an alternative to the `<ui:composition>` tag, for use with a template. In the *Using the template* section, we used the `<ui:composition>` tag with the `template` attribute to tell Facelets to use the template that we created earlier. A side effect of the `<ui:composition>` tag is that all of the code outside this tag will be ignored by Facelets. Most of the time, this is perfectly fine, but sometimes we might want to add content before or after the template. This could be the case if we want to add extra information to the XHTML `head`, which is already defined in the template. Suppose we want to add a `meta` tag to our `head`, we could change the example from the *Using the template* section to the following:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
  <meta name="keywords" content="login,mias"/>
</head>
<body>
<ui:decorate template="templates/template.xhtml">
  <ui:define name="title">Login</ui:define>
  <ui:define name="content">
    <!-- page content left out for brevity -->
  </ui:define>
</ui:decorate>
```

In this example, Facelets will merge the elements from the `<head>` section of the template with the elements from this page. However, care should be taken if we had added a `<title>` tag to the `<head>` tag. In this example, the title of the template would be overwritten by the title set in our example page.

The following is the only attribute of the `<ui:decorate>` tag:

- `template`: The URI where the template can be found. For example `templates/template.xhtml`.

<ui:define> tag

As we have seen in the *Using the template* section, the `<ui:define>` tag is used to define named content that is to be inserted in a template. The name given in the `name` attribute should match the name of a `<ui:insert>` tag in the template.

www.it-ebooks.info

The following is the only attribute of the <ui:define> tag:

- name: The name of the corresponding <ui:insert> tag where the defined content should be inserted

<ui:fragment> tag

The <ui:fragment> tag is to the <ui:component> tag what the <ui:decorate> tag is to the <ui:composition> tag. So, the <ui:fragment> tag will add all of its children to the component tree as a single component, just like <ui:component> tag does. But instead of ignoring everything outside the tag, as the <ui:component> tag does, this tag will cause everything outside of the tag to be rendered just as the <ui:decorate> tag does. To minimize the confusion, the following table gives an overview of the four ways to add components to the tree.

	Everything outside the tag is ignored	**Wraps children in a single component**	**binding supported**
<ui:composition>	Yes	No	No
<ui:component>	Yes	Yes	Yes
<ui:decorate>	No	No	No
<ui:fragment>	No	Yes	Yes

To summarize, one could say that <ui:composition> tag is, most of the time, the best solution for composition components. The <ui:component> tag can be used if a composition component should act more like a single component. The <ui:decorate> tag is useful if we want to override things in a template but we don't have an appropriate <ui:insert> "hook" for it. The <ui:fragment> tag can be used in the rare case that we want to create some sort of "on the fly" component.

The following are the attributes of the <ui:fragment> tag:

- id: An optional unique ID for the component that will be generated. If omitted, Facelets will generate a unique ID for you.
- binding: An optional bean variable to which the generated component will be bound.

www.it-ebooks.info

<ui:include> tag

The `<ui:include>` tag includes other files. This is very useful when creating a template where some parts of the template are defined in a separate file, as discussed in the *Creating a template* section of this chapter.

The following is the only attribute of the `<ui:include>` tag:

- `src`: The URI of the file to include

<ui:insert> tag

The `<ui:insert>` tag is used in a template to generate the placeholders where content is substituted when the template is used. See the *Creating a template* section of this chapter for more information.

Following is the only attribute of the `<ui:insert>` tag:

- `name`: The name that a `<ui:define>` component can refer to

<ui:param> tag

The `<ui:param>` tag can be used instead of the `<ui:define>` tag. Whereas the `<ui:define>` tag passes content snippets to the template, the `<ui:param>` tag passes only values. The value to be passed has to be specified in the `value` attribute.

An extra feature of the `<ui:param>` tag is that it can be used in combination with the `<ui:include>` tag. For example, if we want to pass the title of the page to our `header.xhtml` file, we could use the following code:

```
<ui:include src="header.xhtml">
  <ui:param name="title" value="MIAS"/>
</ui:include>
```

We can now use `title` as an Expression Language variable in the `header.xhtml`.

Following are the attributes of the `<ui:param>` tag:

- `name`: The name of the variable by which the value will be referred to
- `value`: The value to bind to the variable

www.it-ebooks.info

<ui:remove> tag

The `<ui:remove>` tag can be used to exclude a part of the file from the result. There's not much use for this tag other than quickly disabling a set of lines for testing or debugging. The tag has no attributes. Everything inside of the `<ui:remove>` tag will be ignored, and everything outside of it will be rendered normally.

<ui:repeat> tag

The `<ui:repeat>` tag is a replacement for the `<c:forEach>` tag from the JSTL. Although most of the JSTL can be used without any problems with Facelets, the `<c:forEach>` tag can be problematic. Therefore, the `<ui:repeat>` tag has been introduced. For example, if we want to create a list of all users, we could use the following code:

```
<ul>
  <ui:repeat value="#{loginBean.users}" var="user">
    <li>#{user}</li>
  </ui:repeat>
</ul>
```

Summary

In this chapter, we saw the many improvements that Facelets introduces over JSP. We saw that it doesn't make any more sense to use JSP as view technology for new JSF projects. Creating templates with Facelets was introduced, and we saw how we can use a previously-generated template. We also looked into generating composition components, which turned out to be a very powerful feature of Facelets.

We saw how composition components can help us to obey the DRY paradigm. We saw that composition components, once created, can be used in the same way as any other JSF component. Some more advanced and very useful features, such as inline texts and static functions, were also introduced. The chapter concluded with an overview of all specific Facelets tags.

You should understand that this is only an introduction to Facelets. There is a lot more to Facelets than can be covered in a single chapter.

In the next chapter, we will dive into the extensive collection of advanced JSF components that MyFaces Tomahawk has to offer.

www.it-ebooks.info

www.it-ebooks.info

4
Tomahawk

Tomahawk is the set of components that was originally developed together with the first version of MyFaces Core. Tomahawk was designed to extend the standard JSF components in two ways. First, all of the existing standard components were extended with some extra features; second, some extra components were added to expand the possibilities even further. This chapter focuses on how the Tomahawk components extend the JSF standard. We only cover the JSF 1.2 version of Tomahawk. And, of course, we will pay some extra attention to learn how to use the Tomahawk components in conjunction with Facelets.

After reading this chapter, you will be able to:

- Download and configure Tomahawk.
- Create feature-rich data tables by using Tomahawk components. You can use features such as data pagination, inline details, row banding, and so on.
- Create inline and pop-up calendars to let users choose a date on an input form.
- Create file upload capabilities on an input form.
- Use the CAPTCHA component to make sure that your input form is filled in by a human, and not by a machine.
- Use some specialized validators to validate email addresses, credit card numbers, and to check for equality.
- Use the regular expression validator for all other validation needs.

www.it-ebooks.info

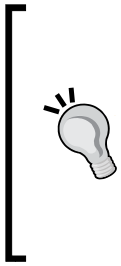
Setting up Tomahawk

To use the Tomahawk component set, we first have to do some set up. The following steps have to be taken:

1. Download Tomahawk and add it to our project
2. Make some settings in the `web.xml` file
3. Add some extra libraries to resolve a few dependencies

The following subsections will focus on each of these tasks.

Downloading Tomahawk



www.it-ebooks.info

Configuring web.xml

Tomahawk uses a special **Extensions Filter**. This filter is used to serve common resources such as images and JavaScript files. These resources are needed by some of the components, and are provided in the Tomahawk JAR file. The Extensions Filter can provide resources transparently; it doesn't matter if the resources are located as separate files within the project or in a JAR file in the classpath. We have to add some lines to our `web.xml` file to make this extensions filter work.

First, we have to define the filter itself:

```
<filter>
<filter-name>MyFacesExtensionsFilter</filter-name>
<filter-class>
org.apache.myfaces.webapp.filter.ExtensionsFilter
</filter-class>
<init-param>
<param-name>uploadMaxFileSize</param-name>
<param-value>20m</param-value>
</init-param>
</filter>
```

The `filter-name` can be anything, but it's a good idea to choose a descriptive name. The `filter-class` is the fully-classified name of the `Filter` implementation that is provided by Tomahawk. This class can be found in the Tomahawk JAR. The `ExtensionsFilter` is also used to handle file uploads for the `<t:inputFileUpload>` component. Therefore, we have to set the maximum file size for uploads here. Any file size can be set, and the following suffixes can be used:

- No suffix for bytes
- `k` for kilobytes
- `m` for megabytes
- `g` for gigabytes

Now that the filter is defined, we have to add at least one filter mapping to make it work. Tomahawk uses two filter mappings.

The first filter mapping looks like this:

```
<filter-mapping>
<filter-name>MyFacesExtensionsFilter</filter-name>
<servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```


www.it-ebooks.info

The `filter-name` value refers to the name that we defined in the filter definition. The `servlet-name` refers to the name that we gave to our Faces Servlet. If you gave your Faces Servlet another name, make sure that you use that name here. This mapping will cause all pages that are handled by the Faces Servlet to pass the through the Extensions Filter. This enables the Extensions Filter to scan each page for links to resources, and then change those links so they point to the Extensions Filter.

In the second filter mapping, we need to map a URL prefix to the Extensions Filter. This will cause all requests that start with this pattern to be routed directly to the Extensions Filter. The Extensions Filter can then serve up the requested resource. This filter mapping looks like this:

```
<filter-mapping>
  <filter-name>MyFacesExtensionsFilter</filter-name>
  <url-pattern>
      /faces/myFacesExtensionResource/*
  </url-pattern>
</filter-mapping>
```

Again, the `filter-name` should match the name defined in the filter definition. The `url-pattern` can be anything, but it should start with the pattern that is used for the Faces Servlet (`/faces/*` in our case). It is also a good idea to have some relation between the name of the filter and the second part or the URL pattern. Now we're ready to go!

Resolving dependencies

Various Tomahawk components depend on one or more external libraries, which are not distributed within the Tomahawk package. Unfortunately, these dependencies are not very well documented by the Tomahawk project itself. Of course, you don't need to download the dependencies yourself if you're using Maven. But if you're not, you can refer to the following table to resolve the dependencies manually:

Library	Version	Needed by
Apache Batik	1.6	`<t:captcha>`
Apache Commons IO	1.3.2	`<t:inputFileUpload>`
Apache Commons Validator	1.3.1	Tomahawk validator components

The listed versions apply to Tomahawk 1.1.9. You should understand that different versions of Tomahawk may depend on different versions of the listed libraries, or even on other libraries. More information can be found in the section of this chapter where the "needed by" components are discussed. We're finished configuring Tomahawk now, so let's have a look at the Tomahawk components, in the next sections.

www.it-ebooks.info

Using extended versions of standard components

As mentioned in the introduction, Tomahawk offers an extended version of every component that is in the standard JSF component set. The following table lists the extra attributes that can be used on all Tomahawk components:

Attribute	Type	Description
`disabledOnClientSide`	`boolean`	If the standard `disabled` attribute is set to `true`, the component will be disabled and the value of the component will not be submitted on a postback of the containing form. This `disabledOnClientSide` attribute will not only render a disabled component, but will also render a hidden input field, causing the value of the component to be submitted with the containing form.
`displayValueOnly`	`boolean`	If set to `true`, only the value of the component is rendered, without an input widget. Of course, `<h:outputText>` tag could be used instead, but by using this attribute, this behavior can be activated through Expression Language.
`displayValueOnlyStyle`	`String`	This is the equivalent of the standard `style` attribute when `displayValueOnly` is `true`.
`displayValueOnlyStyleClass`	`String`	This is the equivalent of the standard `styleClass` attribute when `displayValueOnly` is `true`.
`enabledOnUserRole`	`String`	This attribute will render the component on which it is used as disabled if the user is not in the given role. This assumes that standard Java EE security mechanisms (JAAS) are in use.

www.it-ebooks.info

Attribute	Type	Description
visibleOnUserRole	String	This attribute will render the component invisible if the user does not have the given role assigned to them. This assumes that standard Java EE security mechanisms (JAAS) are in use.
forceIdIndex	boolean	The Tomahawk components that are in a repeating part of the page (such as a list or a data table) and have forceId set to true, will have a numeric row index appended to their id, to guarantee the uniqueness of the ID. If forceIndex is set to false, the row index will not be appended. However, this does not seem to work with Facelets, as Facelets generates an error before the HTML is rendered if a duplicate id is found inside a repeating piece of the component tree, even if forceIdIndex is true.

As a conclusion, one can say that today most of the functionality that the extra attributes can be achieved with Facelets. For example, the behavior of the displayValueOnly attribute can be mimicked by using a Facelets composition component that renders an <h:outputText> component or an input component, depending on a condition. Although the use of displayValueOnly might seem simpler, the advantage of the solution with Facelets is that it can be used with all sorts of components, and not just the Tomahawk components.

For the user-role related attributes, there is a good alternative too. There is a JSF-Security project (see http://jsf-security.sourceforge.net/) that extends the JSF expression language with a securityScope, making it easy to write, for example:

```
<h:inputText enabled="#{securityScope.userInRole['xyz']}">
```

instead of:

```
<t:inputText enabledOnUserRole="xyz">
```

Again, the solution that uses enabledOnUserRole seems a little shorter and simpler, but the solution that uses JSF-Security also works with non-Tomahawk components. In general, one could say that the extra attributes that Tomahawk adds to the standard components were perhaps useful in the early days of JSF when the Tomahawk component set was designed. But nowadays, there are more universal solutions that work with any JSF component. Far more interesting are some of the extended components, which we'll explore in the next section.

www.it-ebooks.info

Extended components

The extended components are the more interesting part of Tomahawk. Unfortunately, the extended components do not seem to be designed as a coherent set. It seems more like a collection of useful components, designed by different persons, on different occasions. But that does not mean that none of these components are useful! This section covers the most interesting extended components from the Tomahawk set. In addition, some components that are of no use when using Facelets are listed, to save you the work of figuring out yourself that you don't need them.

<t:aliasBean> and <t:aliasBeanScope> components

The `<t:aliasBean>` and `<t:aliasBeanScope>` components allow us to define one or more aliases for beans or literal values. The alias name can be referred to from all the children of the component. These two components can be seen as a work-around for the lack of such a feature in JSF. However, Facelets offers us better solutions, such as composition components and templates. So when using Facelets, there's no need for the `<t:aliasBean>` and `<t:aliasBeanScope>` components.

<t:buffer>

The `<t:buffer>` component is another example of a work-around that we don't need anymore. In JSF 1.1, all components were created and rendered in the order in which they appeared in the JSF tree. This meant that it was not possible to refer to a component that was lower on the page. Tomahawk's `<t:buffer>` component provided a work-around for that, but this issue was fixed in JSF 1.2, so we don't need the `<t:buffer>` component anymore.

<t:captcha> component

This component will render a so-called **CAPTCHA** (**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part) image. As the acronym says, this can be used as a way to verify that information on a page was really entered by a human and not some robot that tries to post spam. This method is often used by online publishing systems and discussion forums. If you're building an application for internal use, it probably doesn't make sense to use CAPTCHA.

In case we need it, here's how to use Tomahawk's `<t:captcha>` component. The component can be added to a page like this:

```
<t:captcha captchaSessionKeyName="#{bean.sessionKeyName}" />
```


www.it-ebooks.info

This will render an image with garbled text. An input field where the user can enter the code should be added manually. The component will put the "correct answer" in the given variable in the bean. If the expression in the `captchaSessionKeyName` does not point to a variable in a bean, a variable with the given name will be created in the session scope.

As an example, we could add a CAPTCHA to the MIAS login page. The `<h:panelGrid>` section on that page can be altered as shown in the following code:

```
<h:panelGrid columns="4">
  <mias:field id="userName" bean="#{loginBean}"
              required="true" minLength="6" />
  <mias:field id="password" bean="#{loginBean}"
              required="false" minLength="3" secret="true" />
  <h:outputLabel />
  <t:captcha captchaSessionKeyName="captcha" />
  <h:outputLabel />
  <h:outputLabel />
  <mias:field id="captcha" bean="#{loginBean}"
              required="true" >
    <f:validator validatorId="mias.captchaValidator" />
  </mias:field>
  <h:outputLabel />
  <h:commandButton value="OK" action="#{loginBean.login}" />
  <h:outputLabel />
  <h:outputLabel />
</h:panelGrid>
```

We used empty `<h:outputLabel>` components to put the image in the correct column of the grid. Note that we don't use a bean variable, but let the `<t:captcha>` component put the correct answer in a session variable. We added an extra input field, using our `<mias:field>` composition component, where the user has to enter the text that he sees in the CAPTCHA image. We created a custom validator, which we added to the input field. The custom validator compares the session variable with the user input. The validator's `validate()` method is implemented as follows:

```
public void validate(FacesContext context,
                     UIComponent  toValidate,
                     Object       o)
                                  throws ValidatorException {

  boolean valid = true;
  String value = "";

  if (null != o && o instanceof String) {
```


www.it-ebooks.info

```
      value = (String) o;
      if (!value.equals(getCaptcha(context))) {
        valid = false;
      }
    } else {
      valid = false;
    }

    if (!valid) {
      // throw ValidatorException
    }
  }
```

Most of the code is just standard validator stuff. The interesting line is highlighted. The `getCaptcha()` method uses a `FacesContext` object to retrieve the value of the session variable, as follows

```
private String getCaptcha(FacesContext context) {
    return (String)context.getExternalContext()
                          .getSessionMap().get("captcha");
}
```

When using this component, you might run into a situation where the image is not rendered. The tip we are about to see tells us how this can be resolved.

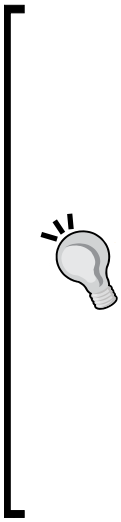


Image not rendered?

The `<t:capthca>` component uses some AWT classes to render the image. Using **Abstract Window Toolkit (AWT)** on an application server may cause problems because AWT will try to access the display of the machine, which may not be allowed (the server may not have a display at all). An indication that you're having this problem is `java.lang.InternalError` in the log files of the application server, with the following message: **Can't connect to X11 window server using ':0.0' as the value of the DISPLAY variable**.

As it is not necessary to use the display, you can configure the **Java Virtual Machine (JVM)** on which the application server is running to be "headless". In this case, AWT will not try to open the display. To do this, use the `-Djava.awt.headless=true` JVM option. Depending on which application server you're using, you'll have to add this option via some administration interface, or you'll have to edit the startup script for the application server.

www.it-ebooks.info



www.it-ebooks.info

Setting up a data table

First, we have to create a new page called `Kids.xhtml`. To add a data table to this page, we use the `<t:dataTable>` component. This component has several attributes, of which the most important ones are:

- `var`: This attribute sets the name of the variable that will contain a data object for each row in the table. This works just like the `var` attribute of the standard JSF `<h:dataTable>` component. In our table, each row will show the data for a specific kid. So, `kid` seems to be a good name for our variable.
- `value`: This must be a JSF EL expression that evaluates to a list of rows. The result of the expression can be of one of the following types:
    - `java.util.List`
    - `java.sql.ResultSet`
    - `javax.servlet.jsp.jstl.sql.Result`

  As our `getKids()` method returns a `List` of `Kid` objects, the value expression can be something like `#{kidsList.kids}` in our case.
- `rows`: This attribute controls the pagination behavior of the data table. If this attribute is not set (or is set to `0`), then all of the rows in the data set will be shown at once. This can be handy for small data sets. If this attribute is set to a value larger than `0`, then that value will be used as the maximum number of rows that will be shown on a screen. The user will then have to use the pagination controls to navigate through the pages. Our kid database contains 1,000 kids, so that will not fit on a single screen. Let's settle for `20` rows on a screen in our example.
- `id`: This attribute gives a unique name to the table. This is necessary, for example, for a `<t:dataScroller>` component to be able to refer to the table (we will see this in detail in the *Using pagination* section). `kids` would be a good ID for our table.

Combining all of this, our table component would look like this:

```
<t:dataTable var="kid" value="#{kidsList.kids}"
rows="20" id="kids">
```


www.it-ebooks.info

Adding columns to the table

Now that we have good values for the attributes of the `<t:dataTable>` component itself, we'll have to add column definitions to our table; otherwise the table wouldn't be able to display any data. A column can be added by a `<h:column>` component or a `<t:column>` component. The `<t:column>` component is an extended version of the `<h:column>` component. One of the extra options that the Tomahawk variant of the column component offers is the possibility to use column spanning. We will use this option to have a single header (`"Name"`) above the columns for first name and last name. A definition of these columns would look like this:

```
<t:column headercolspan="2">
  <f:facet name="header">
    <h:outputText value="Name" />
  </f:facet>
  <h:outputText value="#{kid.firstName}" />
</t:column>
<t:column>
  <h:outputText value="#{kid.lastName}" />
</t:column>
```

The `headercolspan` attribute is set to `2`, which forces the header of the first column to span over the first two columns of the table. The header itself is defined within the `<f:facet>` component with the name `header`. In this case, the `Name` literal text is used. An `<h:outputText>` component is used to render the value for the column. In the first column, the value is defined by a piece of JSF EL—`#{kid.firstName}`. In this piece of EL, `kid` is the variable that was defined by the `var` attribute of the `<t:dataTable>` component. As we know that this will be a `Kid` object, we can use the `firstName` and `lastName` attributes to get the data from the object. (At runtime, this will cause the `getFirstName()` and `getLastName()` methods to be called on each `Kid` object in the `List`.) The second column definition does not have a header facet, as the header of the first column will also span over this column.

If we added all of the columns to our table in this way, the column definition section in our page would be rather lengthy. We would also have some repetition, as the definition of the column header would contain a name that is very similar to the name of the bean property that gives us our data. It's probably a good idea to use a construction as we did for the input fields on the login page, creating a Facelets composition component, and smart usage of the resource bundle. Based on the code that we just saw, the main part of our composition component definition will look like the following code:

```
<t:column headercolspan="#{headerColSpan}">
  <f:facet name="header">
    <h:outputText value="#{msg[headerName]}" />
  </f:facet>
```


www.it-ebooks.info

```
    <h:outputText value="#{bean[columnName]}">
      <ui:insert />
    </h:outputText>
  </t:column>
```

The `headerColSpan` variable is used to be able to set column spanning. We should add some extra code to set a smart default value (`1`) in case no `headerColSpan` is set. The `headerName` variable is used to look up the text for the column header in the message bundle. Some extra conditional code should set the value of `headerName` to the same value as `columnName` if no `headerName` is set. In this way, we can leave out a separate header name for most columns and the header text will be looked up in the message bundle by using the `columnName`. The `bean` variable should be set to the object containing the row data (`kid` in our case). The `<ui:insert>` component is added to be able to add some extra components, such as converters, within the `<h:outputText>` component.

We can use the code that we just saw to create a new `column.xhtml` file. If we register this file in our `mias.taglib.xml` file, and add the extra checks that we just discussed to it, we can use our composition component to reduce the column definition to just a few lines of code:

```
<mias:column columnName="firstName" bean="#{kid}"
             headerColSpan="2" headerName="name" />
<mias:column columnName="lastName" bean="#{kid}" />
<mias:column columnName="age" bean="#{kid}" />
<mias:column columnName="lastScared" bean="#{kid}" />
<mias:column columnName="braveness" bean="#{kid}">
  <f:convertNumber minFractionDigits="1"
                   maxFractionDigits="1"
                   minIntegerDigits="1"
                   maxIntegerDigits="2" />
</mias:column>
```

The `<f:convertNumber>` component is added to format the `double` that is returned by the `getBraveness()` method, so that only one decimal place is shown.

Using pagination

As we set the `rows` attribute of our table to `20`, only the first 20 rows of data will be shown. To give the user the opportunity to browse through all of the data, we need to add pagination controls. For this very purpose, Tomahawk has the `<t:dataScroller>` component. Let's have a look at the most important attributes of this component:

- `for`: This attribute is used to associate the `<t:dataScroller>` component with a `<t:dataTable>` component. The `id` of the `<t:dataTable>` component should be used, which in our case is `kids`.

www.it-ebooks.info

- `paginator`: By default, the `<t:dataScroller>` component will show a paginator, which is a list of page numbers. The user can click on a page number to jump directly to that page. The paginator is also used to indicate the current page, giving the user a sense of where he or she is within the data set. By setting this attribute to `false`, the paginator will be omitted. However, this is not advisable, as there will be no indication of the current position within the dataset.
- `paginatorMaxPages`: This is the maximum number of pages that will be shown in the paginator. The paginator will always try to position the current page in the center with the same number of pages before and after the current page. (Of course, this is not possible for the first few and last few pages of the set.) Therefore, choosing an odd number of pages will give the best results.
- `fastStep`: The `<t:dataScroller>` component has the possibility to add controls for skipping more than one page at a time, which is called "fast forward" and "fast rewind". This attribute defines how many pages are skipped when this control is used. It makes sense to choose a value near the `paginatorMaxPages` value because in that case, the user can use the "fast" controls to skip beyond the boundary of the visible page numbers.

The controls for browsing have to be added via facets. The `<t:dataScroller>` component defines six facets:

- `first` and `last` are used to skip to the first or last page of the set.
- `previous` and `next` are used to browse to the previous or next page in the set.
- `fastrewind` and `fastforward` are used to skip a number of pages. How many pages are to be skipped is set by the `fastStep` attribute of the `<t:dataScroller>` component.

If a facet is left out, the associated control will not be added. For smaller data sets, `fastrewind` and `fastforward` can probably be left out. The contents of the facets will be shown as a link within the web page. If a facet contains only text, a simple link containing the given text will be added. Although texts such as "previous" and "last" could be used, this is probably not the most intuitive solution for the user. It makes sense to use some arrow-shaped images instead. In our example, the code could look like this:

```
<t:dataScroller id="scroller" for="kids" fastStep="10"
paginatorMaxPages="9">
<f:facet name="first">
<t:graphicImage url="../images/start.png"
border="0" />
</f:facet>
```


www.it-ebooks.info

```
    <f:facet name="last">
      <t:graphicImage url="../images/end.png"
                      border="0" />
    </f:facet>
    <f:facet name="previous">
      <t:graphicImage url="../images/back.png"
                      border="0" />
    </f:facet>
    <f:facet name="next">
      <t:graphicImage url="../images/forward.png"
                      border="0" />
    </f:facet>
    <f:facet name="fastforward">
      <t:graphicImage url="../images/fastforward.png"
                      border="0" />
    </f:facet>
    <f:facet name="fastrewind">
      <t:graphicImage url="../images/fastbackward.png"
                      border="0" />
    </f:facet>
  </t:dataScroller>
```

Note that a `<t:graphicImage>` component is used to render an image within the facets. The `border` attribute is used to remove the border from the image. As the image is formatted as an HTML hyperlink, a border will be added by default. (Of course, it would be better to remove the border by using CSS, because then we wouldn't have to repeat the `border="0"` setting for each image.)

Changing the looks of the data table

A data table will be rendered as a standard XHTML table with table rows (`<tr>`) and table cells (`<td>`). We have to add some CSS styling to make these tables look good and (thus) easy to read. Data tables are not the only components that use XHTML tables. So it is probably not a good idea to define CSS styles for all XHTML `<table>` elements and their children. Tomahawk offers a lot of possibilities to set CSS classes for components. We can use these classes to apply the CSS styles to only those XHTML elements that we want the styles to be applied to.

www.it-ebooks.info

This section focuses on giving our components the correct classes. Diving into the details of CSS styling goes beyond the scope of this book. However, a basic example can be found in the source code for this book. Tomahawk components not only have the possibility to add classes to components, but also to provide inline CSS styling. As it is never a good idea to mix style and structure, we will focus on using classes.

Styling the data table itself

Although it is possible to give every XHTML element an ID, it is probably neither desirable nor necessary. If we start out by giving our data table a CSS class, we can use this class to refer to both the table and all of its child XHTML elements. We add a CSS class to our table by changing the table definition to:

```
<t:dataTable var="kid" value="#{kidsList.kids}"
             rows="20" id="kids"
             styleClass="tDataTable">
```

We can now use the name `tDataTable` in our CSS file in order to apply some styling to the table:

```
.tDataTable {
   border: 2px solid black;
}
```

Now, if we want to set the margin and padding for the cells of the table, we don't have to give the cells a class just for this. We can simply use the class of the parent table element:

```
.tDataTable td {
   padding: 2px;
   margin: 2px;
}
```

However, there are some things for which we do need some extra CSS classes. A common way to make data tables more readable is to apply a different background color to odd and even rows. The Tomahawk `<t:dataTable>` component uses the same solution as the standard `<h:dataTable>` component—we can define a comma-separated list of style classes:

```
<t:dataTable var="kid" value="#{kidsList.kids}"
             rows="20" id="kids"
             styleClass="tDataTable"
             rowClasses="rowOdd,rowEven">
```


www.it-ebooks.info

The style classes will be applied to the `<tr>` elements repeatedly. We are not limited to two-style classes. For example, if we add some extra space after each fifth row, we could even change the `rowClasses` value to:

```
rowClasses="rowOdd,rowEven,rowOdd,rowEven,rowFifth,
            rowEven,rowOdd,rowEven,rowOdd,rowTenth"
```

For columns, there is comparable functionality. This is very handy, for example, to change the text alignment for numeric columns. The name of the attribute is not hard to guess—`columnClasses`. To right-align the age, scare date, and braveness columns in our example, we can expand the table definition to:

```
<t:dataTable var="kid" value="#{kidsList.kids}"
             rows="20" id="kids"
             styleClass="tDataTable"
             rowClasses="rowOdd,rowEven"
             columnClasses="textColumn,textColumn,numberColumn,
                            numberColumn,numberColumn">
```

Styling the data scroller

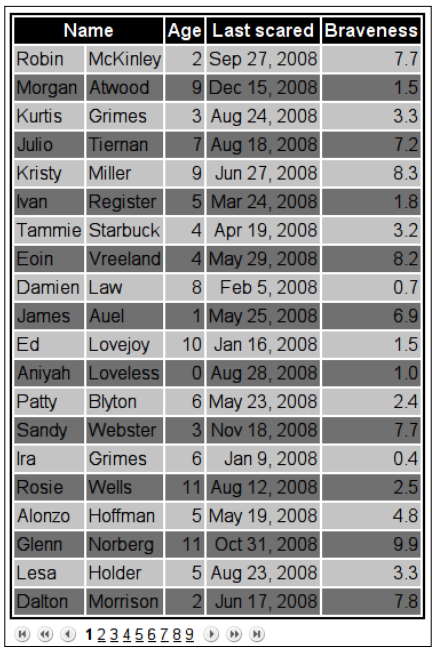
Just like the `<t:dataTable>` component, the `<t:dataScroller>` component has a `styleClass` attribute. The data scroller will be rendered as a single row XHTML table. The CSS class will be applied to that table. We can use the data scroller's class to apply extra formatting, such as slightly smaller text, no coloring for followed links, and so on.

To apply a different style to the number of the current page, the `<t:dataScroller>` component has an extra attribute—`paginatorActiveColumnClass`. The given class name will be applied to the `<td>` element in the single row table that contains the number of the current page.

www.it-ebooks.info

Looking at the result

Applying the discussed styling to our data table will result in a stylish, readable table, as can be seen in this screenshot:



Name		Age	Last scared	Braveness
Robin	McKinley	2	Sep 27, 2008	7.7
Morgan	Atwood	9	Dec 15, 2008	1.5
Kurtis	Grimes	3	Aug 24, 2008	3.3
Julio	Tiernan	7	Aug 18, 2008	7.2
Kristy	Miller	9	Jun 27, 2008	8.3
Ivan	Register	5	Mar 24, 2008	1.8
Tammie	Starbuck	4	Apr 19, 2008	3.2
Eoin	Vreeland	4	May 29, 2008	8.2
Damien	Law	8	Feb 5, 2008	0.7
James	Auel	1	May 25, 2008	6.9
Ed	Lovejoy	10	Jan 16, 2008	1.5
Aniyah	Loveless	0	Aug 28, 2008	1.0
Patty	Blyton	6	May 23, 2008	2.4
Sandy	Webster	3	Nov 18, 2008	7.7
Ira	Grimes	6	Jan 9, 2008	0.4
Rosie	Wells	11	Aug 12, 2008	2.5
Alonzo	Hoffman	5	May 19, 2008	4.8
Glenn	Norberg	11	Oct 31, 2008	9.9
Lesa	Holder	5	Aug 23, 2008	3.3
Dalton	Morrison	2	Jun 17, 2008	7.8

1 2 3 4 5 6 7 8 9

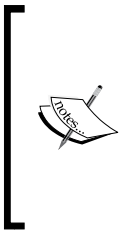
Using advanced data table features

In many data-centric applications, just displaying data "as is" in a stylish table isn't enough. Luckily, Tomahawk offers us a lot of advanced data table features, which we will explore in this section.

Sorting

A very common requirement is that tables need to be sortable. A lot of users expect data to be sortable on a specific column by clicking on that column's header. Tomahawk makes it very easy to implement a sorting feature by taking much of the work out of our hands. The simplest way to make a table sortable is by setting the `sortable` attribute of the `<t:dataTable>` component to `true`. In this case, the `<t:dataTable>` component will make every column in the table sortable. Unfortunately, this feature does not work if a Facelets composition component is used to define each column, which is the case in our example. This is due to the way the auto sorting feature is implemented in Tomahawk, as explained in the following information box.

www.it-ebooks.info



www.it-ebooks.info

```
        <c:when test="#{sortable}">
          <t:commandSortHeader columnName="#{columnName}"
                               arrow="true"
                               propertyName="#{columnName}">
            <h:outputText value="#{msg[headerName]}" />
          </t:commandSortHeader>
        </c:when>
        <c:otherwise>
          <h:outputText value="#{msg[headerName]}" />
        </c:otherwise>
      </c:choose>
    </f:facet>
    <h:outputText value="#{bean[columnName]}">
      <ui:insert />
    </h:outputText>
  </t:column>
```

In the first highlighted line, we simply pass the value of `sortable` to the `<t:dataTable>` component. The second highlighted line is a `<c:when>` test from the standard JSTL library. The code within that block will only be executed if `sortable` is `true`. If `sortable` is not `true`, the code in the `<c:otherwise>` block will be executed. In this way, the `<t:commandSortHeader>` component will only be added to the column header when `sortable` is `true`. Note how we reuse the value of `columnName` for both the `columnName` attribute and the `propertyName` attribute. The `<t:ouputText>` component that renders the contents of the column heading is showing up twice in the code, which is not DRY, but there seems to be no elegant way to prevent this.

We can now update our column definitions to set which columns are sortable and which are not:

```
  <t:dataTable var="kid" value="#{kidsList.kids}"
               rows="20" id="kids" styleClass="tDataTable"
               rowClasses="rowOdd,rowEven"
               columnClasses="alignLeft,alignLeft,alignRight,
                              alignRight,alignRight">
    <mias:column columnName="firstName" bean="#{kid}" />
    <mias:column columnName="lastName" bean="#{kid}"
                 sortable="true" />
    <mias:column columnName="age" bean="#{kid}"
                 sortable="true" />
    <mias:column columnName="lastScared" bean="#{kid}"
                 sortable="true" />
    <mias:column columnName="braveness" bean="#{kid}"
                 sortable="true">
```


www.it-ebooks.info



www.it-ebooks.info

Showing details inline

A nice feature of Tomahawk's `<t:dataTable>` component is the possibility to show extra detailed information per row within the table. This can either be used to show related records if a relational database is used, or just some additional fields that are not shown as table columns. We are going to take the latter approach to explore the possibilities of this feature in our kids overview page.

There are currently two properties of our `Kid` object that are not showing in our table—the `birthDate` and the `country`. Let's add an inline detail view to show those properties. First, we have to prepare our `<t:dataTable>` component to be able to show inline details:

```
<t:dataTable var="kid" value="#{kidsList.kids}"
             rows="20" id="kids"
             styleClass="tDataTable"
             rowClasses="rowOdd,rowEven"
             columnClasses="alignLeft,alignLeft,alignRight,
                            alignRight,alignRight"
             varDetailToggler="detailToggler">
```

We only have to declare a variable to which a "detail toggler" object will be assigned—see the highlighted line in the previous code. We can use this to dynamically show or hide the details on a per-row basis.

We also need an extra column where the show/hide link will appear for each row. As this column will not show a property of a bean, we have to adapt our `column.xhtml` a little to allow "custom" contents. We only have to change the section where the cell contents are rendered:

```
<c:choose>
  <c:when test="#{custom}">
    <ui:insert />
  </c:when>
  <c:otherwise>
    <h:outputText value="#{bean[columnName]}">
      <ui:insert />
    </h:outputText>
  </c:otherwise>
</c:choose>
```


www.it-ebooks.info

We wrap this section in a `<c:choose>` tag and add a test to check if the `custom` attribute is set to `true`. If so, we don't add an `<h:outputText>` component, but just accept all of the child components via a `<ui:insert>` tag—see the highlighted rows in the previous code. The contents of the `<c:otherwise>` branch are the unchanged lines that we had before this change. Now we can add the column for the show/hide links in our `Kids.xhtml` page:

```
<mias:column columnName="details" custom="true">
  <h:commandLink action="#{detailToggler.toggleDetail}">
    <h:outputText value=
                      "#{detailToggler.currentDetailExpanded
                                  ? msg.hide : msg.show}"/>
  </h:commandLink>
</mias:column>
```

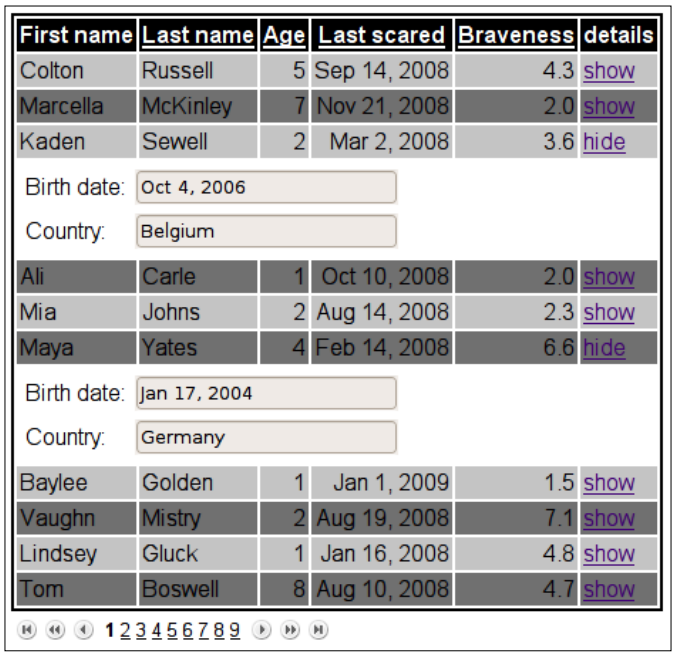
The `action` attribute of the `<h:commandLink>` component takes care of calling the `toggleDetail` property on the `detailToggler` object. (Remember, we assigned the name `detailToggler` via the `varDetailToggler` attribute of the `<t:dataTable>` component.) We use an EL conditional expression within the `<h:ouputText>` component to show different texts if the detail section is expanded.

Now the only thing we have to do is to define what is to be displayed if the detail section is expanded. For that, a facet is defined in the `<t:dataTable>` component. We just have to fill it with some JSF code in order to render the contents of the detail section:

```
<t:dataTable ...>
  <f:facet name="detailStamp">
    <h:panelGrid columns="4">
      <mias:field id="birthDate" bean="#{kid}"
                  readOnly="true"/>
      <mias:field id="country" bean="#{kid}"
                  readOnly="true"/>
    </h:panelGrid>
  </f:facet>
</t:dataTable>
```


www.it-ebooks.info

We added the facet to our `<t:dataTable>` component and added some code to it. We decided to keep it relatively simple and just added a `<h:panelGrid>` component and two `<mias:field>` tags. If we had used a relational database and wanted to show related records, we could have added a second data table instead of the panel grid. The following figure shows the adapted data table with two "expanded" rows:



First name	Last name	Age	Last scared	Braveness	details
Colton	Russell	5	Sep 14, 2008	4.3	show
Marcella	McKinley	7	Nov 21, 2008	2.0	show
Kaden	Sewell	2	Mar 2, 2008	3.6	hide
Birth date:	Oct 4, 2006				
Country:	Belgium				
Ali	Carle	1	Oct 10, 2008	2.0	show
Mia	Johns	2	Aug 14, 2008	2.3	show
Maya	Yates	4	Feb 14, 2008	6.6	hide
Birth date:	Jan 17, 2004				
Country:	Germany				
Baylee	Golden	1	Jan 1, 2009	1.5	show
Vaughn	Mistry	2	Aug 19, 2008	7.1	show
Lindsey	Gluck	1	Jan 16, 2008	4.8	show
Tom	Boswell	8	Aug 10, 2008	4.7	show

1 2 3 4 5 6 7 8 9

Note that all rows are "collapsed" by default. Should you want to have all rows "expanded" by default, you can add `detailStampExpandedDefault="true"` to the `<t:dataTable>` component.

Linking to an edit form

What's the use of an application if it isn't possible to edit any data? Let's see how we can link to an edit form directly from our data table. In this way, the user can select a row to edit and go to the edit form in just one click. We have to choose where to add the link to the edit page. We can make one of the existing columns into a link or add a new column with a dedicated edit link. Although the former method seems efficient, it is not always clear to the user what will happen when he or she clicks on the link. So let's choose the latter method, adding an extra column.

www.it-ebooks.info

We again have to use the "custom" option of our `<mias:column>` composition component:

```
<mias:column columnName="edit" headerName="emptyTableHeader"
             custom="true">
  <t:commandLink action="edit" immediate="true">
    <h:outputText value="#{msg.edit}" />
    <t:updateActionListener
                property="#{editKidForm.selectedKid}"
                value="#{kid}"/>
  </t:commandLink>
</mias:column>
```

The contents of this column consists of a `<t:commandLink>` component, which has two children. The first child is an ordinary `<h:outputText>` component that renders the text for the link. The second child (highlighted) is the interesting part. The `<t:updateActionListener>` component will assign the value defined by the `value` attribute to the variable defined by the `property` attribute. In this way, the `Kid` object of the applicable row will be assigned to the `selectedKid` property of the `editKidForm` object. For this to work, we have to define a session scope managed bean that has a `selectedKid` property and that can be used as backing bean for the edit form. Let's call this managed bean `EditKidForm.java`:

```
public class EditKidForm {
   private Kid selectedKid;

   public void setSelectedKid(Kid selectedKid) {
      this.selectedKid = selectedKid;
      this.firstName = selectedKid.getFirstName();
      this.lastName = selectedKid.getLastName();
      this.birthDate = selectedKid.getBirthDate();
      this.age = selectedKid.getAge();
      this.lastScared = selectedKid.getLastScared();
      this.braveness = selectedKid.getBraveness();
      this.country = selectedKid.getCountry();
   }
   ...
}
```

(Note that only a part of the class is shown here.)

www.it-ebooks.info

The `setSelectedKid()` method both saves a reference to the `Kid` object and copies all of the properties of the `Kid` object into private variables of the bean itself. We can now use these private properties in our form. The reason that we don't use the properties of the `Kid` object in our form directly is that we don't want the `Kid` object to be updated if the page is submitted, but want it to be updated only if a specific action is performed. Therefore, we also implement an `apply()` method and a `save()` method:

```
public String save() {
   doApply();
   return "ok";
}

public String apply() {
   doApply();
   return "apply";
}

private void doApply() {
   if(selectedKid != null) {
      selectedKid.setFirstName(firstName);
      selectedKid.setLastName(lastName);
      selectedKid.setBirthDate(birthDate);
      selectedKid.setLastScared(lastScared);
      selectedKid.setBraveness(braveness);
      selectedKid.setCountry(country);
      age = selectedKid.getAge();
   }
}
```

Note that the value of `age` is reinitialized at the end of the `doApply()` method. This is because it is a calculated value. Also note that the only difference between `save()` and `apply()` is the `String` value that they return. This value is used to either navigate back to the page with the table, or to stay on the edit page.

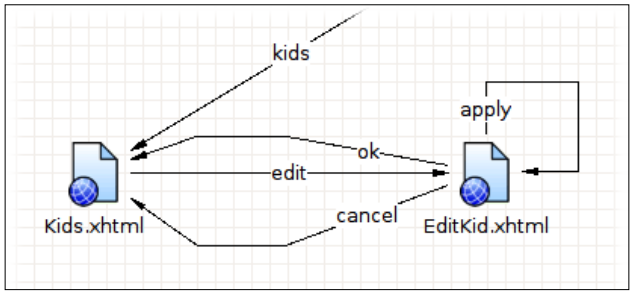
Now, only if the `save()` or `apply()` methods are called, the values of the form are saved into the `Kid` object. Let's see how we use this in our form. We will create a new XHTML file. Let's call it `EditKidForm.xhtml`. The main content section of this file looks like the following code:

```
<h:panelGrid columns="4">
  <mias:field id="firstName" bean="#{editKidForm}" />
  <mias:field id="lastName" bean="#{editKidForm}" />
  <mias:field id="birthDate" bean="#{editKidForm}" />
  <mias:field id="age" bean="#{editKidForm}"
              readOnly="true"/>
  <mias:field id="country" bean="#{editKidForm}" />
  <mias:field id="braveness" bean="#{editKidForm}" />
  <h:outputLabel />
```


www.it-ebooks.info

```
    <h:panelGroup>
      <h:commandButton value="#{msg.apply}"
                       action="#{editKidForm.apply}"/>
      <h:commandButton value="#{msg.ok}"
                       action="#{editKidForm.save}"/>
      <h:commandButton value="#{msg.cancel}"
                       action="cancel" immediate="true" />
    </h:panelGroup>
    <h:outputLabel />
    <h:outputLabel />
  </h:panelGrid>
```

This is just a straightforward JSF data entry form, and no special Tomahawk features are used here. To make the actions a bit clearer, please refer to the next image. This image is a screenshot of the "Navigation Rule" view of the `faces-config.xml` file in Eclipse.



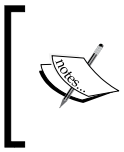
That's all there is to do for the edit form. Most of this is just standard JSF stuff. The most important part is the use of the `<t:updateActionListener>` component, which saves us the work of writing our own action listener. Note that we could have used the `<f:setPropertyActionListener>` component from the standard JSF library as well. This component works more or less the same, except that the `property` attribute is called `target`.

Grouping rows

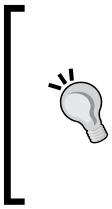
The `<t:dataTable>` components also offer the possibility to group rows. In our example, we could group by age. Grouping by a specific column is as simple as adding `groupBy="true"` to the `<t:column>` component. This means that if we want to use this feature, then we have to extend our `<mias:column>` composition component to pass on the value of the `groupBy` attribute. This will change the column definition in our `column.xhtml` file to:

```
<t:column headercolspan="#{headerColSpan}" id="#{columnName}"
          sortable="#{sortable}" groupBy="#{groupBy}">
```


www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

```
fos.write(photoFile.getBytes());
fos.close();
} catch (FileNotFoundException e) {
// handle exception ...
} catch (IOException e) {
// handle exception ...
}
}
}
```

Notice how the `getName()` and `getBytes()` methods of the `UploadedFile` object are used to get the name and the contents of the uploaded file. A string constant (`IMAGE_DIRECTORY`) is used to prepend the filename with the path to the directory where the images are stored. The exception-handling code is left out here, for brevity. (This is "proof of concept" code. Of course, all sorts of errors can occur with directory creation, and the `fos.close()` statement should be in the final block, to name just a few shortcomings).

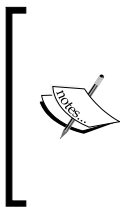
Now we can expand the form itself to incorporate an upload field. To be consistent, we create a new Facelets composition component called `<mias:fileUploadField>`. Here's the most important part of the definition of this component:

```
<c:if test="#{empty required}">
<c:set var="required" value="false" />
</c:if>
<h:outputLabel value="#{msg[id]}:" />
<t:inputFileUpload id="#{id}"
value="#{bean[id]}" />
<h:outputText value="#{required ? '*' : ' '}"/>
<h:message for="#{id}"/>
```

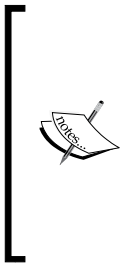
Note that this is largely analogous to how we defined the `<mias:field>` component. The most important part is, of course, the highlighted statement, where the `<t:inputFileUpload>` component is used. `<t:inputFileUpload>` has a few interesting attributes:

- `accept`: As stated on the MyFaces website, *This property appears to have no purpose at all. It certainly has no documentation*. However, a little test shows that Tomahawk 1.1.9 seems to simply ignore this attribute. Perhaps some sensible behavior will be implemented in a newer version of Tomahawk.
- `value`: This attribute must be an EL expression that evaluates to a property of the `UploadedFile` type. In our example, the `photoFile` property of the `EditKidForm` backing bean is a good candidate.

www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

pop up or something similar to input a date. To cater for this need, Tomahawk offers us two different date input components—`<t:inputCalendar>` and `<t:inputDate>`. The former displays a little calendar, either inline or as a pop up. The latter displays separate fields for day, month, and year, where the month field is displayed as a drop-down list. Both can optionally let the user enter a time, too. We will only take a look at the `<t:inputCalendar>` component in this book. The `<t:inputDate>` component has some issues with Facelets and also lacks the possibility to change the order of the day, month, and year fields. This makes this component rather useless.

The `<t:inputCalendar>` component can display calendars either inline or as pop up. If the pop up is used, the date will be displayed in an ordinary input field with a button beside it that will show the pop up when clicked. If no pop up is used, no input field will be rendered; only a calendar will be shown inline, where the user can select a date by clicking on the date in the calendar.

Using a pop-up calendar

Whether the `<t:inputCalendar>` component will be rendered as pop up or displayed inline is determined by the `renderAsPopup` attribute. The default value is `true`, so if this attribute is left out, the component will be rendered as a pop up. If it is set to `false`, an inline calendar will be rendered. There are a lot of attributes involved in showing the pop up. These are listed in the following table:

Attribute	Default	Description
`renderAsPopup`	`true`	If this is set to `true`, or is not present, the component will render a pop-up calendar; if `false`, an inline calendar will be rendered.
`addResources`	`true`	If this is set to `false`, no links to JavaScript or CSS files will be added to the head of the rendered XHTML page. You'll have to add these links yourself. This can be handy if some other component shares the same JavaScript and CSS dependencies.
`popupDateFormat`	The default short format for the configured locale.	This attribute is intended to set a format string for the input field. However, the implementation seems to be buggy. Luckily, there's an alternative to using this attribute—just add a `<f:convertDateTime>` as a child component of the `<t:inputCalendar>` component.

www.it-ebooks.info

Attribute	Default	Description
popupTodayDateFormat		Sets the date format string for the current date, which is displayed at the bottom of the pop up.
id	Randomly generated ID.	This attribute sets a unique ID for every component.
value		Should be an expression that evaluates to a bean property that will hold the date value.
javascriptLocation		Some additional JavaScript resources are required in order to render the pop-up calendar on the client side. These `.js` files are supplied within the Tomahawk JAR. If you want to serve them from another location, or want to use an adapted JavaScript, you can point to the directory where those `.js` files reside.
styleLocation		As with the JavaScript resources, this can be used to specify an alternate location for the CSS resources that are used for the pop-up calendar. If not supplied, the CSS files will be served from the Tomahawk JAR.
popupSelectMode	`"day"`	This is the unit the user may select from the pop up, and can be one of the following: `"day"`: Allow the user to select a day `"week"`: Only allow the user to select a week `"month"`: Only allow the user to select a month `"none"`: Don't allow the user to select anything In week or month mode, the first day of the week or month will be returned as a date. It is the responsibility of the backing code to interpret this date as a week or month.
renderPopupButtonAsImage	`false`	If this attribute is set to `true`, then the button that activates the pop up will be replaced by a calendar icon. A default image is available, but a custom image can be set as well using the `popupButtonImageUrl` attribute.

www.it-ebooks.info

Attribute	Default	Description
`popupButtonImageUrl`		The URL of an image that will be used for the icon that activates the pop-up window. If `renderPopupButtonAsImage` is set to `false`, this attribute is ignored.
`popupButtonString`	`"..."`	Sets the text that will be displayed on the button that triggers the pop up. This is only applicable if `renderPopupButtonAsImage` is `false`.
`popupLeft`	`false`	If this attribute is set to `true`, then the pop-up calendar will be rendered on the left-hand side of the button (or icon) instead of on the right-hand side.

Localizing the pop-up calendar

Localization is never easy when it comes to dates. Luckily, the hardest part of localization is done by the `<t:inputCalender>` component itself. So we don't have to worry about the names of the days and months, the first day of the week, and things like that. However, there are some texts rendered in the pop-up calendar that are not localized automatically. Therefore, some extra attributes are supplied in which we can set these values. Of course, we can use our resource bundle to supply these values in different languages. The following table lists all of the localization attributes for the pop-up calendar:

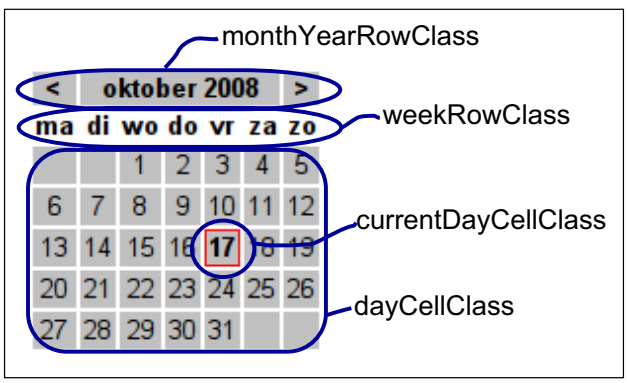
Attribute	Default text	Description
`popupButtonString`	...	The text on the button that triggers the pop up.
`popupGotoString`	**Go To Current Month**	The current date is displayed at the bottom of the pop up. If this date is clicked, the calendar will scroll to the current month. This text will be displayed as a tooltip for the current date.
`popupScrollLeftMessage`	Doesn't work in Tomahawk 1.1.9	
`popupScrollRightMessage`	Doesn't work in Tomahawk 1.1.9	
`popupSelectDateMessage`	Doesn't work in Tomahawk 1.1.9	
`popupSelectMonthMessage`	Doesn't work in Tomahawk 1.1.9	
`popupSelectYearMessage`	Doesn't work in Tomahawk 1.1.9	
`popupTodayString`	**Today is**	Will be prepended to the current date, which is displayed at the bottom of the pop up.

www.it-ebooks.info

Attribute	Default text	Description
`popupWeekString`	**Wk**	To the left of the calendar, a column with week numbers is shown. This text will be used as header for that column.

Using an inline calendar

Although they are both rendered by the same JSF component, the inline calendar is an entirely different thing to the pop-up calendar. So when we set `renderAsPopup` to `false`, we have to deal with a totally different calendar. Whereas the pop-up calendar works with CSS "themes", the inline calendar has to be styled by hand. A couple of attributes are used to set the CSS classes for the various parts of the inline calendar. You have to assign a class name via these attributes, and then supply the CSS styling for these classes via a stylesheet. The available attributes are `monthYearRowClass`, `weekRowClass`, `currentDayCellClass`, and `dayCellClass`. The first two are applied to a `<tr>` element, and the last two are applied to a `<td>` element. See the following image for further clarification:



www.it-ebooks.info

Using the calendar in a form

To use the `<t:inputCalendar>` in our input form, we have to create another Facelets composition component, let's call it `<mias:dateField>`. The most important part of the `dateField.xhtml` file looks like this:

```
<c:if test="#{empty popup}">
<c:set var="popup" value="true" />
</c:if>

<h:outputLabel for="#{id}" value="#{msg[id]}:" />
<t:inputCalendar id="#{id}" value="#{bean[id]}"
renderAsPopup="#{popup}"
required="#{required}"
renderPopupButtonAsImage="false"
popupGotoString="#{msg.gotoCurrentMonth}"
popupTodayString="#{msg.todayIs}"
popupWeekString="#{msg.wk}"
popupTodayDateFormat="#{msg.datePattern}"

styleClass="inlineCalendar"
currentDayCellClass="currentDayCell"
dayCellClass="dayCell"
monthYearRowClass="monthYearRow"
weekRowClass="weekRow">
<f:convertDateTime pattern="#{msg.datePattern}"/>
</t:inputCalendar>
<h:outputLabel value="" />
<h:message for="#{id}"/>
```

The following points should be noted:

- We use only one component to render either an inline or a pop-up calendar. The `renderAsPopup=#{popup}` code ensures that we can use the `popup` argument of our composition component to switch between the two.
- Some arguments are only applicable to either the inline calendar or the pop-up calendar. That's why some empty lines have been added. The first group of arguments applies to both, the second group applies to only the pop-up calendar, and the third group applies to only the inline calendar.
- The date pattern for the `<f:convertDateTime>` converter component, as well as for the `popupTodayDateFormat` argument that applies to the pop-up calendar, is served from the application's message bundle, making it easy to localize the pattern.

www.it-ebooks.info

Without using CSS, the inline calendar does not look very nice and is also pretty useless, as there is no way to see what the selected date is. To achieve a basic look as shown in the previous image, the following CSS styling should be applied:

```
.inlineCalendar td {
   background-color: silver;
   text-align: center;
   font-size: small;
}
.inlineCalendar a {
    text-decoration: none;
    color: black;
}
.inlineCalendar a:visited {
    text-decoration: none;
    color: black;
}
.inlineCalendar a:active {
    text-decoration: none;
    color: black;
}
.inlineCalendar td.currentDayCell {
   border: 1px solid red;
   font-weight: bold;
}
.inlineCalendar td.dayCell {
    border: 1px solid silver;
}
.inlineCalendar .monthYearRow td {
   font-weight: bold;
}
.inlineCalendar .weekRow td{
   background-color: white;
   font-weight: bold;
}
```

The highlighted lines apply styling to the selected date. Also note that three definitions are added for `.inlineCalendar a`, `.inlineCalendar a:visited`, and `.inlineCalendar a:active`. These are to ensure that the date numbers that are rendered as XHTML hyperlinks do not get the default formatting for unvisited, visited, and active hyperlinks, because that would not make any sense here.

www.it-ebooks.info

We can now use our `<mias:dateField>` component in the same way as we use the `<mias:field>` component in our form:

```
<mias:dateField id="birthDate" bean="#{editKidForm}"
                required="true" popup="true" />
```

Extra validators

Tomahawk adds some extra validators in addition to the standard JSF validators. We'll have a quick look at these in the next sections.

Validating equality

The `<t:validateEqual>` component can be used to ensure that the values entered in two different fields are equal to each other. This is, of course, useful for a password change form that has a "new password" field and a "confirm new password" field. The usage is as follows:

```
<h:inputField id="newPassword" value="#{bean.newPassword}"/>
<h:inputField value="#{bean.newPasswordConfirm}">
  <t:validateEqual for="newPassword" />
</h:inputField>
```

Note how the `for` attribute in the validator refers to the `id` of the first input field.

Validating e-mail addresses

To validate an e-mail address, a special `<t:validateEmail>` validator component is available. This uses the e-mail validation of the Apache Commons Validator library, which is rather strict. For example, the address has to end with a top-level domain. That means this e-mail validator does not allow internal e-mail addresses without a top-level domain. This should not be a problem, as such e-mail addresses are rarely used these days. The e-mail validator does not have any special attributes, so the usage is pretty simple:

```
<h:inputField value="#{bean.emailAddress}">
  <t:validateEmail />
</h:inputField>
```


www.it-ebooks.info

Validating credit card numbers

The `<t:validateCreditCard>` component may come in handy when you're building a webshop. It checks if the given string is a valid credit card number. If desired, certain credit card types can be excluded. The recognized credit card types are:

- `amex` for American Express cards
- `discover` for Discover cards
- `mastercard` for Master Card cards
- `none` to allow none of the credit card types
- `visa` for Visa cards

By default, all card types are allowed. By setting a card type attribute to `false`, the corresponding card type is disabled. The following example adds credit card number validation for Master Card and Visa credit cards:

```
<h:inputField value="#{bean.creditCardNumber}">
<t:validateCreditCard amex="false" discover="false" />
</h:inputField>
```

Note that a clever thing to do is to use Expression Language for the `Boolean` card type values, and then implement a back-end data structure to enable or disable the credit card types.

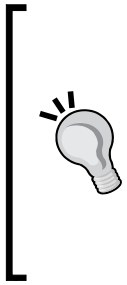
Validating against a pattern

The most versatile of the Tomahawk validators is without a doubt the `<t:validateRegExpr>` component. This validates the user's input against a regular expression. Regular expressions are a very powerful way of specifying text matching patterns, albeit somewhat hard to read by humans. Regular expressions can often save you a lot of custom code. Regular expressions are often used to validate phone numbers, email addresses, postal codes, and so on.

The regular expression validator can be used as follows:

```
<h:inputField value="#{bean.postalCode}">
  <t:validateRegExpr pattern="[0-9]{4}[A-Z]{2}" />
</h:inputField>
```


www.it-ebooks.info



www.it-ebooks.info

www.it-ebooks.info

5
Trinidad—the Basics

Trinidad started its life as Oracle ADF Faces, before Oracle donated it to the Apache MyFaces project. When creating ADF Faces, Oracle focused on creating a set of components that offers a complete solution for building rich web applications. Oracle designed the components to be fully-featured yet not too complicated. This resulted in an extensive set of components that are versatile and easy to use. The components are well designed and their looks can be customized by using skins. All of the Trinidad components were designed to be part of a set. The components have a lot of shared functionality, and have a consistent look and feel. If you have worked with one of the components, then you will find that the other components of the set work just as you would expect them to.

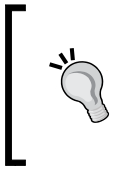
In this chapter we will explore many different components of the Apache MyFaces Trinidad project. We will have a look at the data input components, as well as the output components. Special attention will be given to the many layout components of Trinidad. Of course, we will keep using Facelets as the view technology and we will see how we can get the most out of the combination of Facelets and Trinidad.

After reading this chapter, you will be able to:

- Set up a JSF project to use Trinidad
- Build data-entry pages using Trinidad's input components
- Represent tabular data in nice, formatted, sortable, and pageable tables with Trinidad table components
- Create a navigation structure for you application
- Use the navigation structure to dynamically generate navigation controls on your pages
- Lay out your pages with the many layout components that come with Trinidad

www.it-ebooks.info

Setting up Trinidad



www.it-ebooks.info

```
    <servlet-name>resources</servlet-name>
    <url-pattern>/adf/*</url-pattern>
  </servlet-mapping>
```

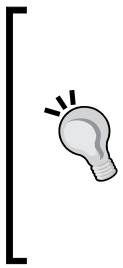
The resource Servlet is mapped to the `/adf/*` URL. This is a remainder from the days before Oracle donated its ADF Faces components to the MyFaces project. The URL cannot be changed, as some parts of Trinidad expect to be able to access CSS and JavaScript files via this URL.

In order to let Trinidad play nice with Facelets, we have to add one extra piece of configuration to our `web.xml` file:

```
  <context-param>
    <param-name>
            org.apache.myfaces.trinidad.ALTERNATE_VIEW_HANDLER
    </param-name>
    <param-value>
            com.sun.facelets.FaceletViewHandler
    </param-value>
  </context-param>
```

This makes the Trinidad components aware of the fact that Facelets is being used instead of JSP, as the view technology.

Configuring the faces-config.xml file



www.it-ebooks.info

Configuring the trinidad-config.xml file

Trinidad also introduces another configuration file—`trinidad-config.xml`. As the name implies, this file can be used to adjust several Trinidad-specific settings. This file is optional, which means that we don't have to create it if Trinidad's default values are fine for our project. A basic `trinidad-config.xml` file looks like this:

```
<?xml version="1.0"?>
<trinidad-config xmlns=
                 "http://myfaces.apache.org/trinidad/config">
  <debug-output>true</debug-output>
  <skin-family>minimal</skin-family>
</trinidad-config>
```

Setting `debug-output` to `true` will cause Trinidad to add some helpful comments to the generated pages, and format the XHTML nicely. Of course, this comes at the expense of performance, so you should disable this setting in production environments.

The `skin-family` setting can be used to select an alternative skin for Trinidad. See the section on skinning in Chapter 7. Many more settings can be made in the `trinidad-config.xml` file, which will be discussed where applicable throughout this chapter. See the *Tuning Trinidad* section in Chapter 7 for an overview of all options.

Adapting our template

In order to get the most out of Trinidad, we'll have to adapt our template slightly. Trinidad has a special component for rendering the `<html>`, `<head>`, and `<body>` XHTML tags. The rationale behind this `<tr:document>` tag is that the generated document doesn't necessarily have to be an XHTML document. By plugging in a different renderer, Trinidad is able to render pages optimized for mobile devices, or even for text-based interfaces, such as telnet. Although you may not be planning to add these features to your application, it's always a good idea to include such options, if it is relatively simple. The inclusion of `<tr:document>` is also necessary if we want to use Trinidad's skinning features, as it takes care of loading the skin's CSS and JavaScript files. See Chapter 7 for more on skinning.

So, basically, we have to remove all `<html>`, `<head>`, and `<body>` tags from our documents. In the case of our `template.xhtml` file (see Chapter 3, *Facelets*), this will cause some trouble because this is the only place where we add some actual content to the `<head>`. Luckily, `<tr:document>` has a `metaContainer` facet that lets us add content to the XHTML `<head>`. So our `template.xhtml` will now look as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
```


www.it-ebooks.info

```
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<tr:document xmlns="http://www.w3.org/1999/xhtml"
             xmlns:ui="http://java.sun.com/jsf/facelets"
             xmlns:h="http://java.sun.com/jsf/html"
             xmlns:f="http://java.sun.com/jsf/core"
             xmlns:tr="http://myfaces.apache.org/trinidad">
  <f:facet name="metaContainer">
    <h:panelGroup>
      <link rel="stylesheet"
            href="/MIAS/templates/mias.css"/>
      <title>
        <ui:insert name="title">
            ** NO TITLE SET **
        </ui:insert>
      </title>
    </h:panelGroup>
  </f:facet>
<!-- rest of template left out for brevity -->
</tr:document>
```

Note that we have enclosed all of the contents of the `metaContainer` facet in a `<h:panelGroup>`. This is because a facet cannot have more than one child component at the top level. For consistency, we should change all of the other `.xhtml` documents by removing the `<html>`, `<head>`, and `<body>` tags, and adding a `<tr:document>` component. But note that Facelets will ignore those `<tr:document>` components anyway, because they're outside the `<ui:composition>` tag.

Creating data tables

The `<tr:table>` component is a powerful component for rendering data tables. Let's explore its possibilities by building a page similar to the one that we built in the previous chapter. The basic table definition we're starting out with is very similar to the first table definition that we used in our Tomahawk example in the previous chapter:

```
<tr:table var="kid" value="#{kidsList.kids}"
rows="20" id="kids">
```

There are some important differences that are not visible in this simple definition, though. These are as follows:

- Although, in our example, the `#{kidsList.kids}` still evaluates to a simple `java.util.List`, we could have used an object of type `org.apache.myfaces.trinidad.model.CollectionModel` instead. In fact, Trinidad will automatically convert our simple `List` to a `CollectionModel` anyway.

www.it-ebooks.info

- By setting `rows` to `20`, Trinidad will not only limit the number of rows in the table to 20, but will also automatically add controls for navigating through the data set, should it contain more than 20 rows. No separate pagination control is needed. As a free bonus, this pagination feature uses AJAX to get the data from the server; we get that for free.

Adding columns

A `<tr:table>` component can only have `<tr:column>` components for the columns, so we can't use the standard `<h:column>` component. The `<tr:column>` component has some extra features; one of these is a quick way to set the header text, which makes the use of a facet superfluous. The other extra feature is the possibility to create column groups by nesting several `<tr:column>` components. This will cause the column header to span over all of the columns in the group. A simple column definition for two columns grouped together could look like this:

```
<tr:column headerText="Name">
  <tr:column>
    <h:outputText value="#{kid.firstName}" />
  </tr:column>
  <tr:column>
    <h:outputText value="#{kid.lastName}" />
  </tr:column>
</tr:column>
```

Note how the `headerText` attribute saves us some typing—at the expense of flexibility, of course. But let's take our Facelets approach to table columns and see how we can create a `<mias:column>` composition component optimized for Trinidad. This composition component also has to allow us to use advanced features such as sorting and column groups. Suppose we've already created a `column.xhtml` file, as described in the Chapter 3, *Facelets*. The main content could look similar to this:

```
<tr:column id="#{columnName}" sortProperty="#{columnName}"
           sortable="#{sortable}"
           headerText="#{msg[headerName]}">
  <c:choose>
    <c:when test="#{custom}">
      <c:set var="headerName" value=""/>
      <c:set var="custom" value=""/>
      <ui:insert />
    </c:when>
    <c:otherwise>
      <h:outputText value="#{bean[columnName]}">
      <ui:insert />
      </h:outputText>
    </c:otherwise>
  </c:choose>
</tr:column>
```


www.it-ebooks.info



www.it-ebooks.info

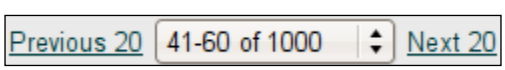
```
maxFractionDigits="1"
minIntegerDigits="1"
maxIntegerDigits="2" />
</mias:column>
<mias:column columnName="edit"
headerName="emptyTableHeader" custom="true">
<tr:commandLink action="edit" immediate="true">
<tr:image source="../images/pencil.png"
inlineStyle="border-width: 0px;" />
<tr:setActionListener to="#{editKidForm.selectedKid}"
from="#{kid}"/>
</tr:commandLink>
</mias:column>
</tr:table>
```

A few interesting things can be said about this piece of code:

- We've nested the first name and last name columns into a column group. Note the `custom="true"` setting on the enclosing `<mias:column>`.
- We've used the same `custom` setting in another way for the last column, which contains a link to the edit form. The `<tr:image>` component will render a nice pencil icon to click on. We've used a little bit of inline CSS to remove the border that otherwise would have been added to the image.
- The `<tr:setActionListener>` component is used to copy the object contained in the `kid` variable to the `selectedKid` property of the backing bean of the edit form. The `<tr:setActionListener>` is more-or-less equivalent to the `<t:updateActionListener>` component from the Tomahawk component library.

Using pagination

As stated earlier, pagination can be activated by setting the `rows` attribute of the `<tr:table>` to some value larger than `0`. A pagination control is then automatically added to the table by Trinidad. This control will consist of two links: one page back and one page forward. If the total size of the data set is known, a drop-down list will be added between the links that allows the user to jump directly to a certain page within the set. The pagination controls will be added at the top of the table. Extra controls can be added at this position by using the `actions` facet. An example of a pagination control is shown in the following image:



www.it-ebooks.info

The table keeps the row number of the first row in the current page in an attribute called `first`. This attribute can also be used to jump to a specific page without using the pagination control. We could, for example, use a literal value or a piece of expression language to set the start position of the table.

It is also possible to perform some special actions when the table jumps to another page. We can use the `rangeChangeListener` attribute to register a "listener method" that will be called every time the table has updated its `first` attribute. We should use expression language to point to a method in one of our managed beans. As mentioned before, the pagination mechanism gets new pages of data from the server via AJAX. This means that, instead of refreshing the whole page, only the contents of the table are replaced.

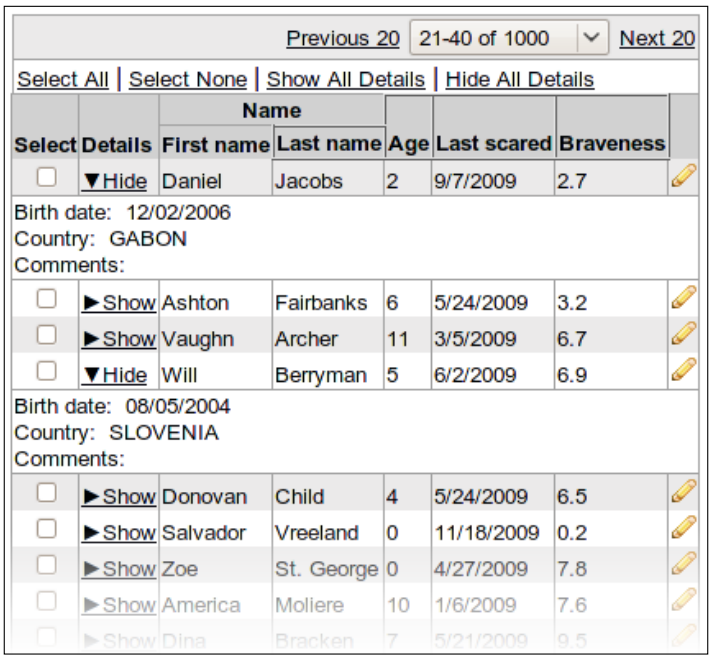
Displaying inline details

Just like the Tomahawk `<t:dataTable>` component, the Trinidad `<tr:table>` component can show extra details in a collapsible detail view. How the details are displayed is defined in a facet called `detailStamp`. With Tomahawk, we had to add a trigger ourselves to show and hide the details; Trinidad does that for us, automatically. This comes at a price, though—we cannot choose where the detail toggle is shown. It is always at the beginning of the row. So to add an inline detail view, the only thing that we have to do is to add this facet to our `<tr:table>`:

```
<f:facet name="detailStamp">
  <h:panelGroup>
    <mias:field id="birthDate" bean="#{kid}" readOnly="true">
      <f:convertDateTime pattern="#{msg.datePattern}" />
    </mias:field>
    <mias:field id="country" bean="#{kid}" readOnly="true"/>
  </h:panelGroup>
</f:facet>
```


www.it-ebooks.info

The `<tr:table>` has the option to display two links in the table header, to show or hide all of the inline details at once. This option is disabled by default, and can be enabled by setting the `allDetailsEnabled` attribute of the `<tr:table>` to `true`. The following image shows how the inline details of the example above are rendered. Note that each row in the table can show or hide its details individually:



Configuring banding and grid lines

By default, all rows in the Trinidad table have the same background, and grid lines are placed between every row and column. This can be changed by using the following attributes of the `<tr:table>` component:

- `rowBandingInterval`: An integer value that sets the interval of the "banding" effect on the rows. If this attribute is set to `1`, the background will change for every alternate row. If it is set two `2`, the background will change after every two rows, and so on.
- `columnBandingInterval`: The banding effect can also be applied to columns in a similar way to the `rowBandingInterval`.
- `horizontalGridVisible`: This enables or disables the grid lines between the rows.
- `verticalGridVisible`: This enables or disables the grid lines between the columns.

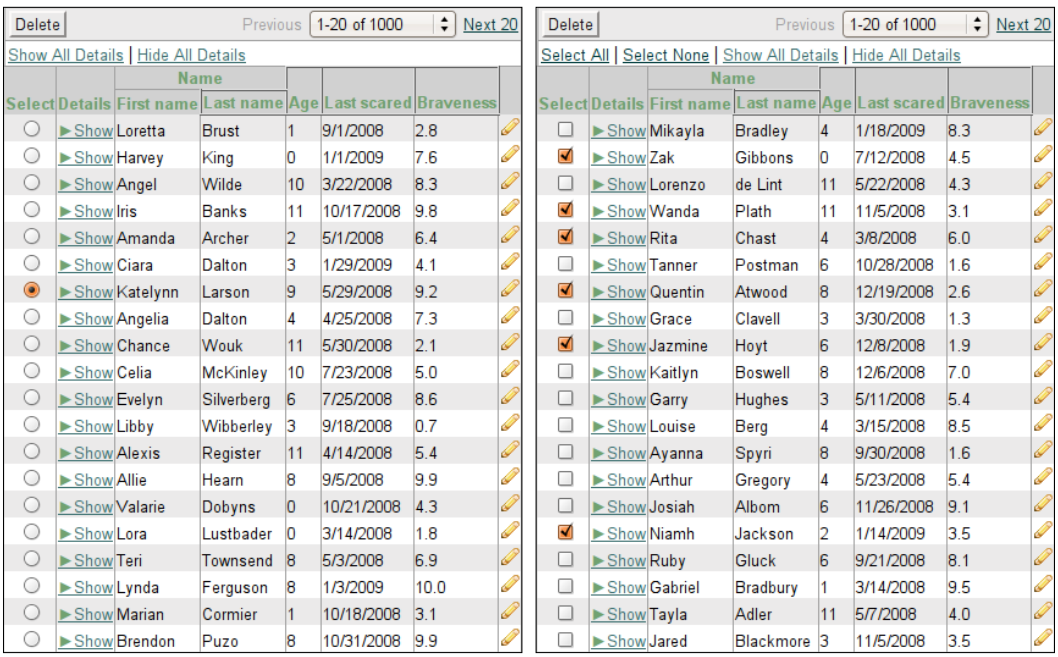
www.it-ebooks.info

The actual appearance of the banding effect and the grid lines depends on the skin that is used. See the *Skinning* section in Chapter 7.

Using row selection

The Trinidad data table component can be configured to allow the user to select a single row or multiple rows in the table. For example, this can be very useful if we want to perform certain actions on a selected set of rows. Let's see how we can use this feature to allow the user of MIAS to delete one or more selected kids from the system.

First, we have to enable multiple selection for the table. This is done by setting the `rowSelection` attribute of the `<tr:table>` component to `multiple`. Other valid values are `single` and `none`, of which the latter is the default. Setting `rowSelection` to `multiple` will add an extra column to the left of the table, with a selection box in each row. If we set `rowSelection` to `single`, this column will contain radio buttons instead of checkboxes, thus allowing only one column to be selected. See the following screenshot as an example:



Delete | Previous 1-20 of 1000 | Next 20

Show All Details | Hide All Details

Select	Details	Name: First name	Name: Last name	Age	Last scared	Braveness
○	▶Show	Loretta	Brust	1	9/1/2008	2.8
○	▶Show	Harvey	King	0	1/1/2009	7.6
○	▶Show	Angel	Wilde	10	3/22/2008	8.3
○	▶Show	Iris	Banks	11	10/17/2008	9.8
○	▶Show	Amanda	Archer	2	5/1/2008	6.4
○	▶Show	Ciara	Dalton	3	1/29/2009	4.1
◉	▶Show	Katelynn	Larson	9	5/29/2008	9.2
○	▶Show	Angelia	Dalton	4	4/25/2008	7.3
○	▶Show	Chance	Wouk	11	5/30/2008	2.1
○	▶Show	Celia	McKinley	10	7/23/2008	5.0
○	▶Show	Evelyn	Silverberg	6	7/25/2008	8.6
○	▶Show	Libby	Wibberley	3	9/18/2008	0.7
○	▶Show	Alexis	Register	11	4/14/2008	5.4
○	▶Show	Allie	Hearn	8	9/5/2008	9.9
○	▶Show	Valarie	Dobyns	0	10/21/2008	4.3
○	▶Show	Lora	Lustbader	0	3/14/2008	1.8
○	▶Show	Teri	Townsend	8	5/3/2008	6.9
○	▶Show	Lynda	Ferguson	8	1/3/2009	10.0
○	▶Show	Marian	Cormier	1	10/18/2008	3.1
○	▶Show	Brendon	Puzo	8	10/31/2008	9.9

Delete | Previous 1-20 of 1000 | Next 20

Select All | Select None | Show All Details | Hide All Details

Select	Details	Name: First name	Name: Last name	Age	Last scared	Braveness
☐	▶Show	Mikayla	Bradley	4	1/18/2009	8.3
☑	▶Show	Zak	Gibbons	0	7/12/2008	4.5
☐	▶Show	Lorenzo	de Lint	11	5/22/2008	4.3
☑	▶Show	Wanda	Plath	11	11/5/2008	3.1
☑	▶Show	Rita	Chast	4	3/8/2008	6.0
☐	▶Show	Tanner	Postman	6	10/28/2008	1.6
☑	▶Show	Quentin	Atwood	8	12/19/2008	2.6
☐	▶Show	Grace	Clavell	3	3/30/2008	1.3
☑	▶Show	Jazmine	Hoyt	6	12/8/2008	1.9
☐	▶Show	Kaitlyn	Boswell	8	12/6/2008	7.0
☐	▶Show	Garry	Hughes	3	5/11/2008	5.4
☐	▶Show	Louise	Berg	4	3/15/2008	8.5
☐	▶Show	Ayanna	Spyri	8	9/30/2008	1.6
☐	▶Show	Arthur	Gregory	4	5/23/2008	5.4
☐	▶Show	Josiah	Albom	6	11/26/2008	9.1
☑	▶Show	Niamh	Jackson	2	1/14/2009	3.5
☐	▶Show	Ruby	Gluck	6	9/21/2008	8.1
☐	▶Show	Gabriel	Bradbury	1	3/14/2008	9.5
☐	▶Show	Tayla	Adler	11	5/7/2008	4.0
☐	▶Show	Jared	Blackmore	3	11/5/2008	3.5

www.it-ebooks.info

Now that we have multiple row selection enabled, we have to add a button to allow the user to do something with their selection. Let's use the `actions` facet to add a `<tr:commandButton>` to the header of our table:

```
<tr:table var="kid" value="#{kidsList.kids}" rows="20"
id="kids" rowSelection="multiple"
binding="#{kidsTable.table}">
....
<f:facet name="actions">
<tr:commandButton
actionListener="#{kidsTable.deleteSelected}"
text="#{msg.delete}"/>
</f:facet>
</tr:table>
```

Note that we reference a new managed bean twice—`kidsTable`. This is a backing bean that we use for our page-specific Java code.

The first highlighted line in the previous code binds the table component to a variable in our bean so that we have access to our table component from within our Java code. The second highlighted line registers a method in our bean as an action listener that will be called whenever the button is clicked. The definition of our backing bean in the `faces-config.xml` is straightforward:

```
<managed-bean>
<managed-bean-name>
kidsTable
</managed-bean-name>
<managed-bean-class>
inc.monsters.mias.backing.KidsTable
</managed-bean-class>
<managed-bean-scope>
session
</managed-bean-scope>
</managed-bean>
```

Now comes the hard part. We have to write some code for our action listener method that will delete the selected kids when the button is clicked. Let's take a step-by-step approach. First, we will create our class and add a variable to hold a reference to the `<tr:table>` component:

```
public class KidsTable {
private UIXTable table;
public UIXTable getTable() {
return table;
}
```


www.it-ebooks.info

```
    public void setTable(UIXTable table) {
      this.table = table;
    }
  }
```

The `UIXTable` type is the class that implements the table component. Its fully classified name is `org.apache.myfaces.trinidad.component.UIXTable`. Now we're going to use the table component to delete the selected rows from the system.

```
    public void deleteSelected(ActionEvent event) {
      Object oldRowKey = getTable().getRowKey();
      Iterator<Object> selectedKeys =
                  getTable().getSelectedRowKeys().iterator();
      Map<Integer, Kid> map = Util.getKidsMap();
      while(selectedKeys.hasNext()) {
        Object key = selectedKeys.next();
        getTable().setRowKey(key);
        Kid kid = (Kid) getTable().getRowData();
        map.remove(kid.getId());
      }
      getTable().setSelectedRowKeys(null);
      getTable().setRowKey(oldRowKey);
    }
  }
```

In the first line, we save the current row key into the local variable `oldRowKey`. We need that to make sure we leave the table in the same state at the end of the method. The second line retrieves an `Iterator` that can iterate over the set of selected row keys. The third line uses a utility method to get access to the underlying data model. Note that we could have used `getTable().getValue()` instead. But in this case, we used a `java.util.Map` to implement our data model. And because the `<tr:table>` component only accepts `java.util.Lists` or arrays as the data model, we used the `values()` method of our `Map` and populated a `List` with those values. So a `getValue()` method on our table object will only give us the values from our `Map`, but we need the `Map` itself to remove entries from it.

In the `while` loop, we use our `Iterator` to iterate over the keys of the **selected** rows. For each key we can use the `setRowKey()` method on the table to make the row with that key the **current** row. After that, we can use the `getRowData()` method to get the data of the row that we just made the current row. That data has to be a `Kid` object, so we can get its ID and use it to remove the object from the map. After the `while` loop, we call `setSelectedRowKeys(null)` to reset the selected rows. Next, we will call `setRowKey()` to make the row that was current at the start of our method current again.

www.it-ebooks.info

So much for the `<tr:table>` component. In the next section, we're going to create input and edit forms by using various Trinidad components.

Creating input and edit forms

Trinidad has an extensive list of input components that can be used on input and edit forms. The Trinidad input components have a lot in common. Therefore, we'll first have a look at the common features that all Trinidad input components share. After that, we will cover the specific features of the individual input components.

Exploring the common features of input components

The Trinidad input components have a lot of extra features when compared to the standard JSF components. One of the most notable differences is the all-in-one approach that all of Trinidad's input components have in common. This means that a single input component can be used to render the component itself, as well as the associated label, an associated error message component, and an indicator for required fields. To nicely align these "embedded" components, a `<tr:panelFormLayout>` component can be used. This is discussed in the *Creating layouts for our pages* section at the end of this chapter.

Using automatic label rendering

As mentioned earlier, all Trinidad input components have the ability to automatically render a label that is associated with the component. For rendering the label, a few attributes are of interest:

- `label`: A string that will be used as the text for the associated label. Of course, expression language can be used to get this text from a message bundle.
- `accessKey`: A single character that will be used as the "access key" (or "mnemonic") for this component. This means that the user can quickly jump to this field by typing this character. If the character is a part of the label text, then this character will be displayed with an underline.
- `labelAndAccessKey`: With this attribute, the label and access key can be set in one step. The string should contain the text that will be shown on the label, where we have to prepend the access character by an ampersand (&), for example, `"First &name"`. In this case, the label will show **First <u>n</u>ame,** and the field will get the focus when the user types *n*.

www.it-ebooks.info

- `simple`: If this attribute is set to `true`, the automatic generation of a label and error message component will be disabled. This also means that the `label`, `accessKey`, `labelAndAccessKey`, and `showRequired` (see the next section) attributes will be ignored.

Using error message support and the required indicator

As mentioned earlier, the input components also add message support automatically. This can be disabled by setting the `simple` attribute to `true`, as discussed in the previous section. If the `required` attribute is set to `true`, the form cannot be submitted if the field is empty, just as with the standard JSF input controls. But Trinidad will also add a visual indicator to show the user that the field is required. This automatic addition of a required indicator can be disabled by setting the `showRequired` attribute to `false`.

There is also an attribute called `requiredMessageDetail` that can be used to set a custom error message that will be shown if the form is submitted while the field is empty. You can use the placeholder `{0}` in this string, which will be replaced by the text of the label that is associated with the component.

Using auto submit

All Trinidad input components have an **auto submit** feature, which can be enabled by setting the `autoSubmit` attribute on the component to `true`. If auto submit is enabled, the entire enclosing form will be submitted whenever the value of the component changes. This can be useful when some calculated value needs to be updated and the calculation is made in the data model. However, you should realize that all form validation will be triggered when the page is submitted. This means that an auto submission triggered by component A can cause a validation error on component B to be fired. You should also be aware that auto submit will generate extra data traffic and, depending on how a submit is handled at the server side, may have some impact on the performance of your application. On the other hand, with auto submit, nice AJAX behavior can be implemented easily.

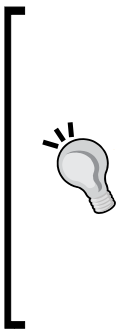
www.it-ebooks.info

Creating plain text input fields

The `<tr:inputText>` component is the normal text input component of Trinidad. Apart from the common Trinidad input features, the following attributes are also important:

- `secret`: If this attribute is set to `true`, the input field can be used for password input. The actual input will be hidden from the user. (Standard JSF has a separate `<h:inputSecret>` component for this.) `secret` does not work if the number of `rows` (see the fourth attribute, below) is larger than `1`.
- `autoComplete`: By default, the user's browser will remember values that were previously entered in a specific text field, and will perform some kind of auto completion. By setting `autoComplete` to `false`, this behavior will be disabled.
- `columns`: This can be used to set the width of the text field. One column approximately corresponds to the space of a single character.
- `rows`: This is the number of rows of the input field. The default is `1`. If it's larger than one, the `secret` attribute will be ignored.
- `wrap`: This sets the wrapping behavior for multiline inputs. The possible values are:
    - `"soft"`: This is the default. Text will automatically be wrapped if it doesn't fit on a single line. The text that will be submitted will not contain any carriage returns.
    - `"hard"`: Text will only be wrapped if it contains a carriage return.
    - `"off"`: Wrapping is disabled. This means that all text will be on a single line, and a scroll bar will appear if the text doesn't fit on the line.
- `maximumLength`: Sets the maximum number of characters that can be entered in the field. This should not be confused with a validator. Setting the `maximumLength` will simply not allow the user to type more characters without giving a prompt. Set this attribute to `0` to let the user enter an unlimited amount of characters. (`0` is also the default.)

www.it-ebooks.info



www.it-ebooks.info

The first three lines after the `<ui:composition>` tag are for setting the correct default for the `autoComplete` attribute. Normally, we just don't set it if we want the default. However, because we want to be able to set it via our composition component, we have to set it here. When it is not set on the composition component, it will cause the default to be an empty string, and setting `autoComplete` to an empty string is not the same as leaving it out. In most cases where a `boolean` is used, setting it to an empty string will cause it to be interpreted as `false`, as is the case with the `required` attribute here.

The highlighted lines are a work-around for the bug mentioned in the previous tip. We set a very large number (equal to `Integer.MAX_VALUE`) as the maximum length if it is empty. If we don't do that, the component would erroneously interpret that as a `maximumLength` of `0`, which wouldn't allow us to type any text. This shows another benefit of using composition components. If we weren't using composition components, we would have to apply this work-around to all of the places where we used the `<tr:inputText>` component with more than one row.

The `<tr:inputText>` component declaration in our composition component is actually pretty long. That's because the `<tr:inputText>` component has a lot of optional attributes that we do not want to hide from the users of our composition component, which means we have to pass them on.

Note how we reused the `id` of our component three times—for the `id` itself, for the field name within the bean, and for the key to look up the label text in the resource bundle.

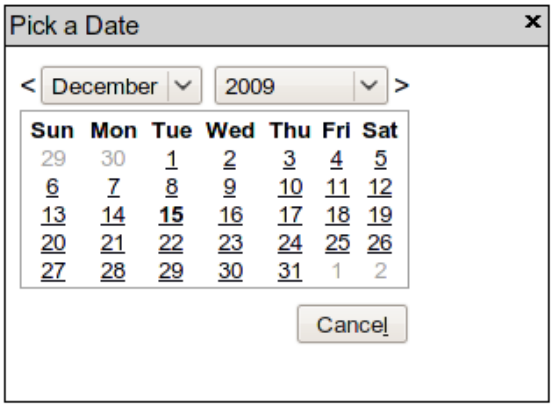
Creating date input fields

The `<tr:inputDate>` component lets users select a date by typing it or selecting it from a pop-up calendar. This component is fairly easy to use; most of its behavior is the same as the `<tr:inputText>` component. The differences are:

- The `<tr:inputDate>` component will automatically add a converter to the input field that will convert the typed value into a date (and will cause an error if the input value can't be converted to a date). This default converter uses the "short" date format that is defined for the current locale. To use another format, we can add a `<f:convertDateTime>` converter with the desired format.
- An icon will be rendered next to the input field that will open a pop-up window with a calendar if the user clicks on it. If Trinidad's "lightweight dialogs" feature is enabled (see the *Tuning Trinidad* section in Chapter 7), the pop up will be rendered via DHTML. Otherwise, a new browser window will be opened.

www.it-ebooks.info

The following image shows a date selection pop up that is rendered as a "lightweight dialog":



Converting dates

Trinidad offers us various possibilities for converting and validating dates, that go beyond the JSF standard. The `<tr:convertDateTime>` component can be used to convert a text string (that the user has typed) into a date. As with the standard `<f:convertDateTime>` component, a pattern can be set, or a date style can be chosen (`default`, `short`, `medium`, `long`, or `full`). The difference is that the Trinidad `<tr:convertDateTime>` component will try harder to interpret the user input as a date. For example, if the date pattern has a slash (`/`) as a separator, the `<tr:convertDateTime>` component will also allow a dash (`-`) or a period (`.`) as a separator. And if the pattern prescribes short month names (`MMM`), the converter will also allow month numbers (`M` and `MM`).

As an extra bonus, the `<tr:convertDateTime>` component adds extended error message support by adding the following attributes:

- `messageDetailConvertBoth`: This error message will be shown when the conversion fails and the type attribute of the `<tr:convertDateTime>` component is set to `both`
- `messageDetailConvertDate`: This message will be shown on conversion failure when the type is `date`
- `messageDetailConvertTime`: This message will be shown when the type is set to `time` and the conversion fails

www.it-ebooks.info

All three messages can use three placeholders within the message:

- `{0}`: This will be replaced by the text of the label of the input component
- `{1}`: This will be replaced by the value that the user entered
- `{2}`: This will be replaced by an example of the expected format

Validating dates

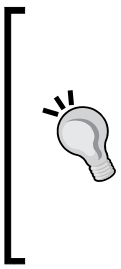
Converting a date adds some implicit validation, as you know for sure that all input data will be a valid date. But often you need the date to be in a certain range, or before, or after, a certain date. Trinidad offers us a `<tr:validateDateTimeRange>` component to make this an easy job. This validator takes the following attributes:

- `minimum`: No dates before this date can be entered. This can be an expression that evaluates to a date, or a `String` literal that contains a date formatted in the following pattern: `yyyy-MM-dd`. (There is a bug with the literal values. See the next tip.)
- `maximum`: No dates after this date can be entered. The same input restrictions as with the `minimum` apply.
- `messageDetailMinimum` and `messageDetailMaximum`: The error message that will be shown if the date is before the minimum or after the maximum date. The following placeholders can be used in the error message:
    - `{0}`: This will be replaced by the label of the associated component
    - `{1}`: This will be replaced by the value that is entered by the user
    - `{2}`: This will be replaced by the minimum or maximum date
- `messageDetailNotInRange`: This error message will be shown if both the `minimum` and `maximum` are set, and the date is outside that range. We can use the following placeholders:
    - `{0}`: This will be replaced by the label of the associated component
    - `{1}`: This will be replaced by the value that the user entered
    - `{2}`: This will be replaced by the minimum date, or the lower end of the range
    - `{3}`: This will be replaced by the maximum date, or the upper end of the range

www.it-ebooks.info



www.it-ebooks.info

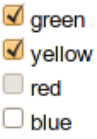


www.it-ebooks.info

Note how we added two validators to our composition component. They will be inserted at the position of the `<ui:insert>` tag in the component definition. The first validator is added to make sure that no "last scared" date can be added before the birth date of the kid. The second validator forces "scare dates" to be on working days only.

Creating selection lists

Trinidad has quite a few components for letting the user choose one or more options from a list. The following table gives an overview of the options:

	One	Multiple
Checkboxes / radio buttons	`<tr:selectOneRadio>` 	`<tr:selectManyCheckbox>` 
List	`<tr:selectOneListbox>` 	`<tr:selectManyListbox>` 
Pull-down list	`<tr:selectOneChoice>` 	not available
Shuttle	not available	`<tr:selectManyShuttle>` 

www.it-ebooks.info

Adding list contents

All of these components require a child component that fills the list of options. A single `<f:selectItems>` component can be used, or one or more `<f:selectItem>` components or `<tr:selectItem>` components can be used. We'll use the `<tr:selectItem>` components here, as they offer us the most flexibility. The list with four colors from the example images in the table could thus be created as follows:

```
<tr:selectItem value="1" label="green"
               shortDesc="the color green" />
<tr:selectItem value="2" label="yellow"
               shortDesc="the color yellow" />
<tr:selectItem value="3" label="red"
               shortDesc="the color red"
               disabled="true"/>
<tr:selectItem value="4" label="blue"
               shortDesc="the color blue" />
```

Note that the `shortDesc` attribute will be used by the component to render a tool tip where possible. Also note that the "red" option is disabled, which means that the user cannot choose this color.

Although fixed lists are nice for demonstrations, most of the time the list of possible values will (and should) be generated by some backend system. Now, assume that we get a `java.util.List` of `ColorBean` objects from one of our beans. We could then change our fixed list into this:

```
<c:forEach var="color" items="#{bean.colors}">
  <tr:selectItem value="#{color.value}"
                 label="#{color.name}"
                 shortDesc="#{color.description}"
                 disabled="#{!color.enabled}" />
</c:forEach>
```

This example assumes that the `ColorBean` class has the `value`, `name`, `description`, and `enabled` members.

Optional empty selection for single selection lists

All of the single selection components (`<tr:selectOneChoice>`, `<tr:selectOneRadio>`, and `<tr:selectOneListbox>`) have an `unselectedLabel` attribute. If this attribute is not empty, its value will be shown as the first item in the list, and selecting it will be equal to selecting nothing. This means that the validation will fail if the component's `required` attribute is `true`. Otherwise, a `null` value will be passed to the variable that is referenced in the `value` attribute.

www.it-ebooks.info

Options for all selection components

All selection components have the ability to add a value change listener. By using expression language, you can set the value of `valueChangeListener` to a method on a backing bean, like this:

```
valueChangeListener="#{myBackingBean.selectionChanged}"
```

This will cause the `selectionChanged()` method to be called every time the selection changes. This value change listener method could look as follows:

```
import javax.faces.event.ValueChangeEvent;
public class MyBackingBean {
  // other methods ...
  public void selectionChanged(ValueChangeEvent event) {
    // do something
  }
}
```

The selection components also have an attribute called `valuePassThru`. This attribute's default value is `false`. If it is set to `true`, then the `value` of the `selectItem` component will be passed to the client as the index of the item in the choice list. Otherwise, an integer index would have been generated. You may need this feature if you want to use a client-side JavaScript function that does need the `value` of the `selectItem` component.

Checkboxes and radio buttons

The `<tr:selectManyCheckbox>` and `<tr:selectOneRadio>` components render a list of checkboxes and radio buttons respectively. This is mainly useful for relatively short lists. The benefit of this type of lists is that the user can see all possible choices at once. For making multiple choices, the checkbox approach is also the most intuitive. Both components have a `layout` attribute, which can be set to `horizontal` or `vertical`. The former value will cause the checkboxes or radio buttons to be on a single line. The latter value (which is also the default) will cause the boxes or buttons to be stacked vertically.

www.it-ebooks.info

Listboxes

Listboxes look identical, no matter which of the two components—`<tr:selectManyListbox>` or `<tr:selectOneListbox>`—is used. Using a listbox doesn't make much sense for single selection, though. A choice list (or "combobox"; see next section) is just as easy to use and takes up much less space on the screen. For longer lists, where multiple selections can be made, the use of a listbox does make sense. In such a case, a list of checkboxes would probably take up too much space on the screen, whereas a listbox uses a scroll bar to minimize the space used. However, it is less intuitive to the user because he or she has to hold down the *SHIFT* or *CTRL* key while selecting multiple items. Both the `<tr:selectManyListbox>` and `<tr:selectOneListbox>` components have a `size` attribute that can be used to set the number of items that are visible without using the scroll bar.

Choice list

The choice list (also called "combobox" or "pull-down list") is only available for single selections. It is easy to use for both short and medium-size lists. For very long lists, it may be difficult to use, depending on how the user's browser renders the list if it doesn't fit on one screen. The largest benefit of the choice list is, of course, that it takes up little space on the screen. There are no special formatting options for this component.

Shuttle

Although it takes up a lot of screen space, the shuttle offers a user-friendly way of choosing multiple items from a long list. The user may need some time to get used to this component, as shuttles are not used very often. The shuttle consists of two lists—one containing all unselected items, and the other containing the selected ones. Between the two lists are controls to move selected items or all items from left to right, or the other way round. Unlike the other selection components, the `<tr:selectManyShuttle>` component has a lot of special features, which are listed here:

- The `leadingHeader` and `trailingHeader` attributes allow us to add a header above the list of available items (leading) and the list of selected items (trailing). The reason the terms leading and trailing are used instead of left and right is that the lists will be shown in a different order if used in a page that uses a right-to-left layout (for right-to-left languages).
- The `leadingDescShown` and `trailingDescShown` are `boolean` attributes that control whether an extra text field is shown below each of the lists. This extra text field will contain the description of the first selected item in the list. The description is taken from the `shortDesc` attribute on the `<tr:selectItem>` component.

www.it-ebooks.info

- Like the listboxes, the shuttle has a `size` attribute that determines the size of the lists. Both lists will have the same size. The size must be between 10 and 20. Sizes lower than 10 will be interpreted as 10, and sizes over 20 will be interpreted as 20.
- Although the shuttle has a `label` attribute, no label will be shown. The value of the `label` attribute will only be used to identify the shuttle in error messages.
- The `<tr:selectManyShuttle>` component has a facet called `filter`. The purpose of this facet is to allow you to add a control that the user may use to filter the contents of the leading list (the list with available items). However, there's no special mechanism provided to implement this, meaning that we'll have to implement a filtering function ourselves. The contents of this facet will be rendered above the leading list. This will cause the leading list to start a bit lower on the page than the trailing list, which does not look very nice.
- There are also two facets called `leadingFooter` and `trailingFooter`, which can be used to add components just below the leading or trailing list. There's only limited space reserved for this, so the height of the contents of these footers should not exceed 26 pixels.

The controls to move items from the leading to the trailing list and back are rendered as simple hyperlinks by default, with the labels **Move**, **Move all**, **Remove**, and **Remove all**. An example is shown in the overview of selection components a few pages back. It is probably more intuitive for the users to replace these text links by some arrow-shaped icons. This can be done by using Trinidad's skinning features that will be discussed in Chapter 7.

Ordering shuttle

There's also a component called `<tr:selectOrderShuttle>`. This is almost the same as the `<tr:selectManyShuttle>` component, except that it adds the possibility to order the selected items. Four extra controls are added to the side of the trailing list for that purpose. The `<tr:selectOrderShuttle>` component also has an extra attribute, `reorderOnly`. If set to `true`, this attribute will cause the leading list to disappear. Therefore, you should make sure that the items to reorder are already selected.

www.it-ebooks.info

Creating a universal composition component for selections

As we did for text input fields and date fields, we can use Facelets' composition component feature to create a single component that can be used for selections. Of course, we could create a component that can create all sorts of selection lists, but it is probably a better idea to first determine which types of selection lists are needed in an application. For our MIAS example application, let's assume that we only need a single selection. Now let's see how we can create a composition component that meets all of our selection needs.

To be able to render different types of single selection lists with one composition component, we need an attribute to set the list type; let's call that attribute `type`. We can then use a JSTL `<c:choose>` element to render one of Trinidad's choice list components, depending on the value of `type`. This will result in code like this:

```
<ui:composition>
  <c:if test="#{empty type}">
    <c:set var="type" value="choice" />
  </c:if>
  <c:choose>
    <c:when test="#{type == 'radio'}">
      <tr:selectOneRadio value="#{bean[id]}" id="#{id}"
                         required="#{required}"
                         readOnly="#{readOnly}"
                         label="#{msg[id]}:"
                         partialSubmit="#{partialSubmit}"
                         autoSubmit="#{autoSubmit}">
        <mias:items items="#{items}"
                    itemValue="#{itemValue}"
                    itemLabel= "#{itemLabel}"/>
      </tr:selectOneRadio>
    </c:when>
    <c:when test="#{type == 'listBox'}">
      <!-- tr:selectOneListbox omitted for brevity -->
    </c:when>
    <c:otherwise> <!-- "choice" is the default type -->
      <tr:selectOneChoice value="#{bean[id]}" id="#{id}"
                          required="#{required}"
                          readOnly="#{readOnly}"
                          label="#{msg[id]}:"
                          partialSubmit="#{partialSubmit}"
                          autoSubmit="#{autoSubmit}">
        <mias:items items="#{items}"
```


www.it-ebooks.info

```
                    itemValue="#{itemValue}"
                    itemLabel= "#{itemLabel}"/>
      </tr:selectOneChoice>
    </c:otherwise>
  </c:choose>
</ui:composition>
```

The code for this composition component is longer than other components that we created. This is because of the `<c:choose>` element, which is needed to render different components depending on the value of the `type` attribute. We use an extra composition component to render the items for the lists, in order to prevent repetition of the code. We pass the values of `items`, `itemValue`, and `itemLabel` to this component. The following listing shows how we could create this items component:

```
<ui:composition>
  <c:if test="#{not (empty itemValue and empty itemLabel)}" >
    <c:if test="#{empty itemValue}">
      <c:set var="itemValue" value="#{id}" />
    </c:if>
    <c:if test="#{empty itemLabel}">
      <c:set var="itemLabel" value="#{id}" />
    </c:if>
  </c:if>
  <c:if test="#{empty itemValue and empty itemLabel}" >
    <c:forEach var="item" items="#{items}">
      <tr:selectItem value="#{item}" label="#{item}" />
    </c:forEach>
  </c:if>
  <c:if test="#{not (empty itemValue and empty itemLabel)}" >
    <c:forEach var="item" items="#{items}">
      <tr:selectItem value="#{item[itemValue]}"
                     label="#{item[itemLabel]}" />
    </c:forEach>
  </c:if>
</ui:composition>
```

We will distinguish two different types of lists here. The first one is the most simple. For this list, we assume that the value to be shown to the user is the same as the value to be set. In other words, when `items` contains a list of `String`s, we show each `String` as an option in the list, and when the user selects one item, the selected `String` is copied in the field where the selection should be stored. In this case, we don't use the `itemValue` and `itemLabel` attributes.

www.it-ebooks.info

The second type of list is a bit more complicated. For this list, we assume that `items` is a list of objects. The values of `itemLabel` and `itemValue` point to a property that each object in the list should have. For each object in the list, the value of the property that corresponds with `itemLabel` is shown to the user. When the user selects an item from the list, the value of the property identified by `itemValue` is stored. Let's explore an example to make this clearer. Suppose we want the user to select a country from a list, and we have a list of `Country` objects, where a country object has two properties: `code` and `name`, as shown:

```
public class Country {
  private String name;
  private String code;
  public String getName() {
    return name
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getCode() {
    return code;
  }
  public void setCode(String code) {
    this.code = code;
  }
}
```

Of course, we want to store the value of `code`, and not the value of `name`. Not only because it is more efficient to store a two-letter code instead of a name, but also because the two-letter country codes are language independent. However, we want to present a list of full names of the countries, as we don't expect the user to know all of the country codes. So we want to display `name` in our choice list. With the previously created composition component, we can achieve this as follows:

```
<mias:selectField id="country"
                  bean="#{editKidForm.selectedKid}"
                  type="choice"
                  items="#{mias:getCountries()}"
                  itemValue="code"
                  itemLabel="name" />
```


www.it-ebooks.info

We use a Facelets static function here to get the list of countries; `#{mias:getCountries()}` will return a `List` of `Country` objects. The `itemValue` attribute is set to `"code"`, so the country code will be stored in the `Kid` object. By setting `itemLabel` to `"name"`, we make sure the full name of the country is shown in the selection list. The full example code for the composition component, as well as the country selection example, can be found in the source code of the example application that can be downloaded from the author's website: `http://www.bartkummel.net`.

Creating fields for numerical input

Trinidad has several ways to make the input and validation of numerical values easier. Let's have a look at the possibilities.

Adding conversion to a field

Trinidad has its own `<tr:convertNumber>` converter, which has nearly the same functionality as the standard `<f:convertNumber>` converter. The only difference is that Trinidad's converter adds the possibility to use custom error messages. The converter component has a few extra attributes for setting custom error messages; the one you should use depends on whether you're using a standard `type` or using your own `pattern`. Each message (regardless of the type) can have at least two placeholders—`{0}` will be replaced by the label of the associated input component and `{1}` will be replaced by the value that the user entered. In the following table, you can find which message attribute should be used:

Value of `type`	**Value of** `pattern`	**Message attribute to use**
`currency`	<empty>	`messageDetailConvertCurrency`
`number`	<empty>	`messageDetailConvertNumber`
`percent`	<empty>	`messageDetailConvertPercent`
<empty>	custom pattern	`messageDetailConvertPattern`

The `messageDetailConvertPattern` message can have a third placeholder (`{2}`) that will be replaced by a custom pattern. Although this sounds useful, you should bear in mind that most users don't understand regular expressions.

www.it-ebooks.info

Adding validation to a field

As with conversion, Trinidad also has two number validation components that, compared to their counterparts from the JSF standard implementation, only offer custom error messages as extra functionality. The following table lists the message attributes that can be used. These attributes can be used on both the `<tr:validateLongRange>` and `<tr:validateDoubleRange>` validation components.

Value of `maximum`	**Value of** `minimum`	**Message attribute to use**
set	empty	`messageDetailMaximum`
empty	set	`messageDetailMinimum`
set	set	`messageDetailNotInRange`

The next table lists the placeholders that can be used with the various message attributes, and what they will be replaced with:

	`messageDetailMaximum`	`messageDetailMinimum`	`messageDetailNotInRange`
`{0}`	The label of the associated input component		
`{1}`	The value the user entered		
`{2}`	The maximum value	The minimum value	The minimum value
`{3}`	n/a	n/a	The maximum value

Adding a spin box to an input field

To enter integer values, sometimes a spin box can be an easy way to set the value, especially if the range of values is not too long. To use a spin box for integer input, Trinidad has the `<tr:inputNumberSpinbox>` component. This component behaves and looks much like a normal `<tr:inputText>` component, except that two little buttons are rendered at one side of the component. These buttons allow the user to increment or decrement the value of the input field. By default, the number is incremented or decremented by `1`, but the step size can be configured by the `stepSize` attribute. Of course, `maximum` and `minimum` attributes can also be configured. The values of these three attributes must all be integer values, so it is not possible to use fractions. The `<tr:inputNumberSpinbox>` component has an implicit number converter, so only numbers can be entered into the input field.

www.it-ebooks.info

File uploading

As the JSF standard does not say anything about uploading files from a JSF page, every component set library offers its own solution for this, and so does Trinidad. Handling file uploads involves some extra steps, which are described in this section. As in the Tomahawk chapter, we are going to add a photo upload facility to the edit kid form as an example.

Meeting the prerequisites

File uploading depends on the Trinidad filter. So before you continue, make sure you have configured this correctly, as described in the *Setting up Trinidad* section at the beginning of this chapter.

We also have to make sure that the XHTML form that is generated as a part of our page accepts files for upload. This means that the `enctype` attribute of the XHTML `<form>` tag has to have the `multipart/form-data` value. If we are using a standard `<h:form>` component to render the `<form>` tag, we should set these values. However, as we're using a `<tr:form>` tag in our application, we can simply set `usesUpload` to `true` and the Trinidad form component will set the correct values in the XHTML form tag for us.

There is one complication, though. We have our `<tr:form>` tag in our Facelets template. This means that all forms will be configured for uploading if we simply add the `usesUpload="true"` setting to this tag. So we have to use an expression, as shown in the following code snippet from `template.xhtml`:

```
<div id="content">
  <tr:form usesUpload="#{usesUpload}">
...
    <h2> <ui:insert name="title" /> </h2>
    <ui:insert name="content" />
  </tr:form>
</div>
```

Now we can set the `usesUpload` variable to `true` in the pages where we need upload facilities, like in this snippet from the `EditKid.xhtml` page:

```
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Edit kid</ui:define>
  <ui:param name="usesUpload" value="true"/>
  <ui:define name="content">
...
</ui:composition>
```


www.it-ebooks.info

Using the file upload component

Now that the prerequisites have been met, we can start using the `<tr:inputFile>` component. We can use it just like any other input component; it shares the same common features of Trinidad, such as embedded labels and indicators for required fields. We can simply bind the `value` attribute of the `<tr:inputFile>` component to a property in a backing bean. That property has to be of the type `org.apache.myfaces.trinidad.model.UploadedFile`. Of course, we have to take some special actions to save the uploaded data to the file system. Let's see how we can do this.

Creating and using a file upload composition component

Of course, we want to add the upload component to our pages, using a composition component that has a similar interface as the other composition components that we created previously. Let's see how we can create such a composition component. The composition component can be fairly simple, as shown in the following code snippet from the `fileUploadField.xhtml` file:

```
<ui:composition>
  <c:if test="#{empty required}">
    <c:set var="required" value="false" />
  </c:if>
  <tr:inputFile id="#{id}" value="#{bean[id]}"
                label="#{msg[id]}:" required="#{required}"/>
</ui:composition>
```

We can now add our composition component to the form in the `EditKid.xhtml` page:

```
<ui:define name="content">
  <tr:panelFormLayout>
...
    <mias:fileUploadField id="photoFile"
                          bean="#{editKidForm}" />
    <mias:photoField id="photoUrl"
                     bean="#{editKidForm}" />
...
  </tr:panelFormLayout>
</ui:define>
```

Note that parts of the page have been omitted, to save space.

www.it-ebooks.info

Note that `photoFile` is a property of the `editKidForm` bean and not of the `selectedKid` object that we use for the other fields in the form. This is because the `Kid` object that `selectedKid` points to doesn't know how to store a file. We therefore let the `editKidForm` backing bean store the file and update the `Kid` object. Note that we also created a `<mias: photoField>` composition component that uses a `<tr:image>` component internally to render the photo once it is uploaded. In the source code that can be downloaded from the author's website the full example can be found.

Saving the file in the backing bean

We have to write some code in our backing bean in order to save the file to the file system. Before we get started, let's have a quick look at some useful methods of the `UploadedFile` class that we can use:

- `dispose()`: This will clear all of the resources that were allocated for this file.
- `getContentType()`: This method will return a `String` containing the MIME type of the uploaded file, such as `"image/png"` or `"text/html"`.
- `getFilename()`: This will return the original filename of the uploaded file.
- `getInputStream()`: This will return a `java.io.InputStream` that we can use to read the contents of the uploaded file. This method may throw a `java.io.IOException`.
- `getLength()`: This method returns the size of the file in bytes.

With these methods, we can copy the uploaded file to the filesystem upon submission of the form. A method that performs this copy action is shown below:

```
private void savePhoto() {
UploadedFile photoFile = getPhotoFile();
if (photoFile != null) {
// Save the filename in the Kid object
selectedKid.setPhoto(photoFile.getFilename());
// Check if the directory exists
File f = new File(IMAGE_DIRECTORY);
if (!f.exists()) {
f.mkdir();
}
// Create the file, delete it if it already exists
f = new File(IMAGE_DIRECTORY + photoFile.getFilename());
if(f.exists()) {
f.delete();
}
FileOutputStream fos = null;
InputStream is = null;
try {
fos = new FileOutputStream(f);
is = photoFile.getInputStream();
```

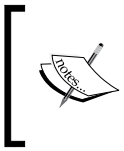

www.it-ebooks.info

```
        // Copy the bytes
        byte[] buffer = new byte[4096];
        for (int n; (n = is.read(buffer)) != -1;) {
          fos.write(buffer, 0, n);
        }
        fos.close();
      } catch (FileNotFoundException e) {
        // handle exception
      } catch (IOException e) {
        // handle exception
      } finally {
        try {
          fos.close();
        } catch (IOException e) {
          // handle exception
        } finally {
          try {
            is.close();
          } catch (IOException e) {
            // handle exception
          }
        }
      }
    }
  }
```

Note that only the filename is saved in the `Kid` object. Also note that all images are stored in the same directory. Depending on the requirements, this is not always desirable, as it allows the users to overwrite each other's images. If the uploaded images belong to a single user, a solution can be to append the username to the path. The file itself is copied to a directory in the filesystem. To be able to display the uploaded image later on, we have to add a method that reconstructs the full path to the file from the filename that is saved to the `Kid` object. Such a method is shown below:

```
public String getPhotoUrl() {
  String photoFileName = selectedKid.getPhoto();
  if (photoFileName != null) {
    photoFileName = "/photos/" + photoFileName;
  } else {
    photoFileName = "";
  }
  return photoFileName;
}
```


www.it-ebooks.info



www.it-ebooks.info

Setting upload limits in web.xml

The following `web.xml` snippet shows how to set the application wide file upload limits:

```
<context-param>
  <param-name>
                org.apache.myfaces.trinidad.UPLOAD_MAX_MEMORY
  </param-name>
  <param-value>102400</param-value>
</context-param>
<context-param>
  <param-name>
            org.apache.myfaces.trinidad.UPLOAD_MAX_DISK_SPACE
  </param-name>
  <param-value>2048000</param-value>
</context-param>
<context-param>
  <param-name>
                  org.apache.myfaces.trinidad.UPLOAD_TEMP_DIR
  </param-name>
  <param-value>/tmp/uploads/</param-value>
</context-param>
```

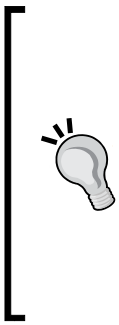
The `UPLOAD_MAX_MEMORY` parameter sets the maximum size of the in-memory part of the file. In this example, the size is set to `102400` bytes, which is 100 kB. The maximum disk space for an uploaded file is set by the `UPLOAD_MAX_DISK_SPACE` parameter. In this example it is set to `2048000` bytes, which is 2,000 kB. The default upload directory is set by the `UPLOAD_TEMP_DIR` parameter, and is set to `/tmp/uploads/` in this example.

Setting upload limits in trinidad-config.xml

As described earlier, setting the upload limits in `trinidad-config.xml` has the benefit of being able to make the upload limits user-configurable. If upload limits are set in `trinidad-config.xml`, any upload limits parameter in `web.xml` are ignored by Trinidad. Suppose we have a bean called `Preferences.java` that holds the preferences, like in the following example:

```
public class Preferences {
  private long maxUploadMemory = 1024L;
  private long maxUploadDiskSpace = 204800L;
  private String tempDir = "/tmp/uploads";
  public long getMaxUploadMemory() {
    return maxUploadMemory;
  }
```


www.it-ebooks.info



www.it-ebooks.info

Using Trinidad's hierarchical navigation features

Trinidad has a navigation framework for page navigation. The framework is based on the assumption that the project is organized around a hierarchical navigation structure. Let's explore the possibilities of this navigation framework by adding some navigation to our MIAS example program. Let's create a super simple hierarchical structure like the following:

```
Home                    Start.jspx
+- Kids overview        Kids.jspx
|  +- Edit kid          EditKid.jspx
+- Employee overview    Employees.jspx
   +- Edit employee     EditEmployee.jspx
```

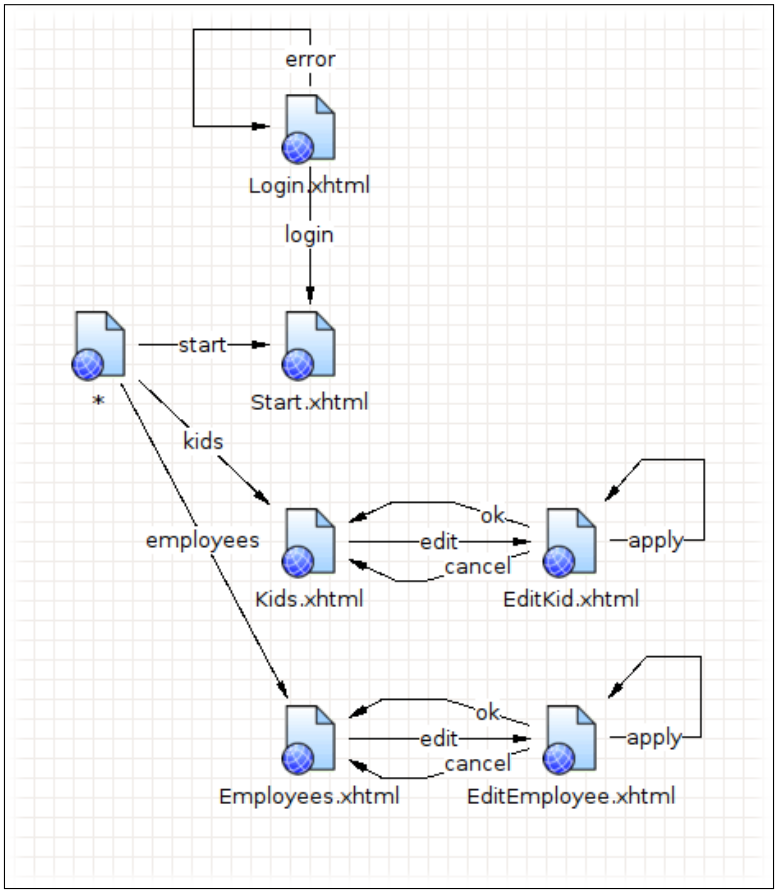
Configuring the hierarchy

The Trinidad navigation framework lets us define this structure in an XML file; let's call it `menu.xml`. For the structure proposed before, the contents of that file should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"
      resourceBundle="inc.monsters.mias.Messages"
      var="msg">
  <itemNode id="menu0" focusViewId="/Start.xhtml"
            label="#{msg.home}" action="start">
    <itemNode id="menu00" focusViewId="/Kids.xhtml"
              label="#{msg.kids}" action="kids">
      <itemNode id="menu000" focusViewId="/EditKid.xhtml"
                label="#{msg.editKid}" action="edit"/>
    </itemNode>
    <itemNode id="menu01" focusViewId="/Employees.xhtml"
              label="#{msg.employees}" action="employees">
      <itemNode id="menu010"
                focusViewId="/EditEmployee.xhtml"
                label="#{msg.editEmployee}" action="edit"/>
    </itemNode>
  </itemNode>
</menu>
```


www.it-ebooks.info

Note that we used our existing resource bundle for the `labels` of the `itemNodes`. It is important to understand that this file does not replace the navigation rules that are set in `faces-config.xml`. The only thing that this file does is to help the various navigation components to determine the navigation path to a certain page. We have to make sure that the `action` and `focusViewId` attributes of every `itemNode` element correspond to the pages and outcomes, in the `faces-config.xml` file. Compare the `actions` and `focusViewIds` from the XML that we just saw with the following image. Note that the `"start"`, `"kids"`, and `"employees"` actions don't have a `from-view-id` specified (indicated in the image by the page icon with a * character). This is done to make sure that we can navigate to the associated pages from wherever we are. This saves us from having to define all possible paths in our `faces-config.xml`.



www.it-ebooks.info

The navigation components that we're going to use later on cannot read the `menu.xml` file themselves. Therefore, we need a managed bean that does this for us. Trinidad comes with an `XMLMenuModel` class that we can use. We can configure it as a managed bean in the `faces-config.xml` file:

```
<managed-bean>
  <managed-bean-name>miasMenu</managed-bean-name>
  <managed-bean-class>
              org.apache.myfaces.trinidad.model.XMLMenuModel
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>source</property-name>
    <value>/WEB-INF/menu.xml</value>
  </managed-property>
</managed-bean>
```

Note that the managed bean is `request` scoped. This means that for every request, a bean is instantiated and the `menu.xml` file is read. This is done so that the beans always know the current location within the hierarchy.

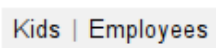
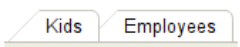
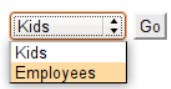
Creating navigation panes

Now that we have configured our navigation hierarchy, we can use this hierarchy with several navigation components. Let's start by adding a simple list of links to our `Start.xhtml` page. We can use the `<tr:navigationPane>` component for that purpose:

```
<tr:navigationPane var="node" value="#{miasMenu}"
                   level="1" hint="list">
  <f:facet name="nodeStamp">
    <tr:commandNavigationItem text="#{node.label}"
                              action="#{node.doAction}"/>
  </f:facet>
</tr:navigationPane>
```


www.it-ebooks.info

The `value` attribute of the `<tr:navigationPane>` component refers to our managed bean, which we called `"miasMenu"`. The `var` attribute defines the name of the variable that we can use in every navigation node. This works just like rows in a table, where we define a `var` that we can use in every row. A `<tr:navigationPane>` component can show one level at a time. To make sure that the links to the pages `Kids.xhtml` and `Employees.xhtml` are shown, we set the `level` attribute to `1`. If we set it to `0`, which is also the default, the top level (in our case only `Start.xhtml`) will be shown. The `nodeStamp` facet will be repeated for every node at the given level. The `<tr:commandNavigationItem>` works just like a `<tr:commandButton>` of `<tr:commandLink>`—if a user clicks on it, the associated `action` will be performed. How the `<tr:commandNavigationItem>` component will be rendered depends on the value of the `hint` attribute of the `<tr:navigationPane>` component. The following table shows the possible values, and an example of the result of each. Note that the exact appearance can be changed by Trinidad's skinning facilities; see Chapter 7.

Hint	Example	Description
`bar`		A bar with links.
`tabs`		A bar with tabs. If you have such a bar on every page, it seems to the user as if he or she is selecting tabs.
`buttons`		Not exactly what you'd expect, given the hint. More or less same as the bar, but without a background.
`choice`		A choice list with a **Go** button. This might be handy if there are a lot of navigation items, but otherwise it just requires an extra click by the user.
`list`		A bullet list of links.

Creating breadcrumbs

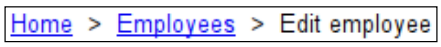
The same navigation hierarchy can also be used to add breadcrumbs. Suppose we would like to indicate to the user where he or she is in the hierarchy, we could add a `<tr:breadCrumbs>` component to our page template to achieve that. This component is very similar to the `<tr:navigationPane>` component. It doesn't have a `level` attribute, of course. This snippet can be added to our template to add breadcrumbs to every page:

```
<tr:breadCrumbs var="crumb" value="#{miasMenu}"
                orientation="horizontal">
```


www.it-ebooks.info

```
    <f:facet name="nodeStamp">
      <tr:commandNavigationItem text="#{crumb.label}"
                                action="#{crumb.doAction}"/>
    </f:facet>
  </tr:breadCrumbs>
```

The `orientation` attribute defaults to `horizontal`. If we set it to `vertical`, the breadcrumbs will be displayed as an indented list. The following image shows an example of the breadcrumbs on the **Edit employee** page.



Creating a hierarchical menu

We can also use our navigation hierarchy to display a hierarchical menu. For this we can use the `<tr:navigationTree>` component. The usage of this component is almost the same as for the other two navigation components. We could add it to one of our pages, as follows:

```
<tr:navigationTree var="node" value="#{miasMenu}">
  <f:facet name="nodeStamp">
    <tr:commandNavigationItem text="#{node.label}"
                              action="#{node.doAction}"/>
  </f:facet>
</tr:navigationTree>
```

There's one important difference with the `<tr:breadCrumbs>` component that you should keep in mind. For the breadcrumbs to show even on the edit pages, we had to add those edit pages to our navigation hierarchy. If we use the same hierarchy for the `<tr:navigationTree>`, we get direct links to the edit pages in our menu. That's not what we want, as the edit pages don't make any sense if no entity is selected to edit. There's no elegant solution for this, other than using either the `<tr:breadCrumbs>` or the `<tr:navigationTree>` component in one project—but not both of them.

www.it-ebooks.info

Creating layouts for our pages

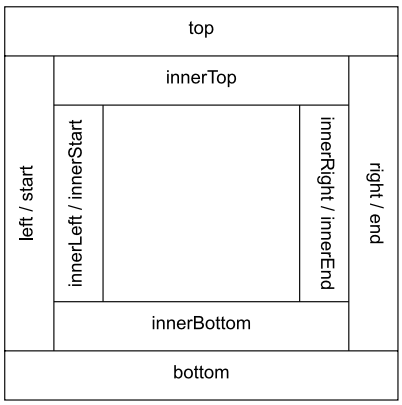
Trinidad offers a set of layout components that help us to lay out our pages. In a way, they are comparable to layout managers in Java Swing. Layout components can easily be recognized by their names, which are always in the form `<tr:panel...Layout>`, where `...` indicates the type of layout. Just like Swing layout managers, pretty much every layout you like can be achieved by nesting different layout components. Layout components do not add visible elements to the page; they only position their children.

There are also a lot of `<tr:panel...>` components that do add visual elements to the page and position their children. These components can be recognized by their names, which also start with `panel`, but do not end with `Layout`.

Using a border layout

The `<tr:panelBorderLayout>` component behaves a bit like a `BorderLayout` manager in Swing. The `<tr:panelBorderLayout>` component places its children in the center. It has 12 facets that are placed around the child component(s) that are in the center. The following image displays how the contents of the facets are placed around the center. Of course, not all facets have to be used. We can simply leave out the facets that we don't need.

Note that on the left and right sides, some facets share the same position. This means that they are mutually exclusive. So you should either use `left`, `innerLeft`, `innerRight`, and `right` or `start`, `innerStart`, `innerEnd`, and `end`. The difference is that the left and right facets are always placed to the left or right, whereas the start and end facets are placed according to the reading direction of the current locale. So in a right-to-left language, the contents of the `start` facet are placed to the right of the center, while the same contents are placed to the left of the center in a left-to-right language.



www.it-ebooks.info

Layout methods

The `<tr:panelBorderLayout>` component can use two different layout methods. They can be set by using the `layout` attribute:

- `layout="positioned"`: The size of the facets can be set by using attributes of the `<tr:panelBorderLayout>` component. For the top and bottom facets, only the height can be set. For the left, right, start, and end facets, only the width can be set. The available attributes are—`topHeight`, `innerTopHeight`, `bottomHeight`, `innerBottomHeight`, `leftWidth`, `innerLeftWidth`, `startWidth`, `innerStartWidth`, `innerEndWidth`, `endWidth`, `innerRightWidth`, and `rightWidth`. All CSS style units can be used for the sizes—`em`, `ex`, `px`, `in`, `cm`, `mm`, `pt`, `pc`, and `%`. The default for all facets is `25%`.
- `layout="expand"`: The component is automatically expanded over the available space to fit the child components and the contents of the facets. This is also the default behavior if no layout method is set. If this layout method is selected, the size attributes (`topHeight`, `innerLeftWidth`, and so on) are ignored.

An example usage of the component is shown next:

```
<tr:panelBorderLayout bottomHeight="100px"
layout="positioned">
<f:facet name="innerLeft">innerLeft</f:facet>
<f:facet name="innerRight">innerRight</f:facet>
<f:facet name="bottom">bottom</f:facet>
child </
tr:panelBorderLayout>
```

In this example, the text “child” goes in the center with the texts “innerLeft” and “innerRight” to the left and right, and the text “bottom” below it. The area in which the bottom contents are added will be 100-pixel high.

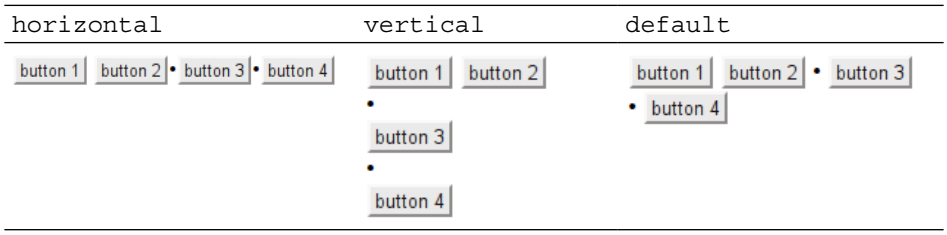
www.it-ebooks.info



www.it-ebooks.info

```
    <f:facet name="separator">&#8226;</f:facet>
</tr:panelGroupLayout>
```

The `•` entity is displayed as a bullet character. The following image shows how this example will be rendered when the `layout` attribute is set to `vertical` or `default` instead of `horizontal`:



No separator is placed inside the group of buttons that is grouped by the `<tr:group>` components. Of course, the button bars are just used as an example here—there are a lot of other situations where we could use the `<tr:panelGroupLayout>` component, such as vertically stacking various elements on a page, creating a "film strip" of images, and so on.

Using a horizontal layout

The `<tr:panelHorizontalLayout>` component is much like the `<tr:panelGroupLayout>` component with its `layout` attribute set to `horizontal`. The differences are as follows:

- The `<tr:panelHorizontalLayout>` component has, obviously, no `layout` attribute that can be used to change the layout.
- The `<tr:panelHorizontalLayout>` has two extra attributes that can be used to change the alignment of the child elements within the available space:
    - `halign`: Sets the horizontal alignment. Valid values are `right`, `start`, `left`, `end`, and `center`. `start` is equivalent to `left` in right-to-left locales, or to `right` in right-to-left locales; `end` is analogous.
    - `valign`: Sets the vertical alignment. Valid values are `middle`, `top`, `baseline`, and `bottom`.

Just like the `<tr:panelGroupLayout>` component, the `<tr:panelHorizontalLayout>` component has a `separator` facet.

www.it-ebooks.info

Creating layouts for input forms

The purpose of the `<tr:panelFormLayout>` layout component is to align various input components and their labels, automatically. This works very well if you use the Trinidad input components with their `label` or `labelAndAccesKey` attributes. In most cases, the default behavior might be good enough for your needs. However, there are some ways to influence the appearance of our input forms.

The `<tr:panelFormLayout>` component is capable of distributing its child components over multiple columns. This is controlled by the `rows` and `maxColumns` attributes. The `maxColumns` attribute sets the maximum number of columns that will be used. This maximum is guaranteed. The `rows` attribute determines after how many rows a new column should be started. However, if there are more components in the form than the value of `rows` multiplied by the value of `maxColumns`, the number of rows will automatically be increased. Thus, the number of rows is not guaranteed.

The width of the labels and the components is automatically determined by the form layout component. However, this process can be influenced by using the `labelWidth` and `fieldWidth` attributes. Both attributes accept absolute values (in pixels) as well as percentages. The latter is the preferred way to influence the width. If percentages are used, the `labelWidth` and `fieldWidth` values should always add up to exactly 100%. If they don't, the `<tr:panelFormLayout>` component will change the percentages such that they add up to exactly 100%. Most of the time we won't need these attributes, but they may be useful when we're trying to line up two different `<tr:panelFormLayout>` components on one page.

Grouping components

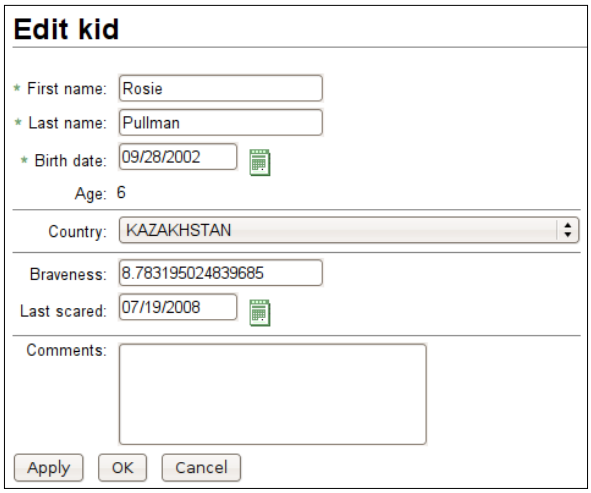
The grouping of components is also supported by the `<tr:panelFormLayout>` component. You can group components by surrounding them with a `<tr:group>` component. Groups of components will be separated by a visual separator. The form layout component will keep groups together if multiple columns are used, which means that a group will always be in one column. The form layout component will count every row in a group as a row in the total form. However, if a group has more rows than the value of the `rows` attribute of the form layout component, then the group will still be in a single column of the form.

www.it-ebooks.info

Label and message

Although the `<tr:panelFormLayout>` component works very well with all of Trinidad's input components, sometimes there may be the need to use another component. Or we might want to combine two or more components in a form as if they're one, with a single label. For these cases, there is the `<tr:panelLabelAndMessage>` component. We can simply put everything that should be handled as one component as a child of the `<tr:panelLabelAndMessage>` component. We can use either the `label` attribute or the `labelAndAccesKey` attribute to set the label, just as we do for normal Trinidad input components. We should not forget to set `simple` to `true` for all Trinidad input components that we add as children to the `<tr:panelLabelAndMessage>` component.

The following image shows an input form formatted with a `<tr:panelFormLayout>` component, without setting either the `width` attributes, or the `rows` or `maxColumns` attributes. As you can see, separators are added between groups of components that are grouped with a `<tr:group>` component.



Footer facet

The `<tr:panelFormLayout>` component has a facet called `footer` that can be used to add buttons to the form. In the previous example, a `<tr:panelButtonBar>` component containing three buttons is added to the `footer` facet as follows:

```
<f:facet name="footer">
  <tr:panelButtonBar>
    <h:commandButton value="#{msg.apply}"
                     action="#{editKidForm.apply}" />
    <h:commandButton value="#{msg.ok}"
```


www.it-ebooks.info

```
                    action="#{editKidForm.save}" />
      <h:commandButton value="#{msg.cancel}"
                    action="cancel" immediate="true" />
    </tr:panelButtonBar>
  </f:facet>
```

Creating an accordion

The accordion component is mainly meant for navigation purposes. It wraps around multiple `<tr:showDetailItem>` components. Normally, a `<tr:showDetailItem>` component shows a link that unveils some details below it when clicked. Grouped in a `<tr:panelAccordion>` component, the `<tr:showDetailItem>` components work together; some sections are closed automatically when others are opened. The exact behavior depends on two attributes, as described in the following table:

discloseMany	discloseNone	**Description**
false	false	This is the default setting. There is always exactly one child expanded. By selecting one child, any other expanded child will be collapsed. It is not possible to collapse a child by clicking on it.
false	true	Only one child can be expanded at a time, but it is also possible to collapse all children.
true	false	Multiple children can be expanded at the same time. Children can be collapsed, but at least one child will stay expanded.
true	true	Multiple children can be expanded at the same time. Children can be collapsed and it is possible to collapse all children at the same time.

The contents of the `<tr:showDetailItem>` component are not formatted in any special way. So if we want to have—for example—a list of hyperlinks in each section of the accordion, we have to take extra provisions to make sure that the hyperlinks are stacked vertically. This can be done by using a `<tr:panelGroupLayout>` component with the `layout` attribute set to `vertical`, like in the following example:

```
<tr:panelAccordion discloseMany="false" discloseNone="true">
  <tr:showDetailItem text="Apache MyFaces websites">
    <tr:panelGroupLayout layout="vertical">
      <tr:goLink text="Main website"
                 destination="http://myfaces.apache.org"/>
      <tr:goLink text="Wiki"
                 destination="http://wiki.apache.org/myfaces/"/>
    </tr:panelGroupLayout>
  </tr:showDetailItem>
```


www.it-ebooks.info

```
    <tr:showDetailItem text="Book resources">
      <tr:panelGroupLayout layout="vertical">
        <tr:goLink text="Packt publishing"
                   destination="http://www.packtpub.com/"/>
        <tr:goLink text="Example sources"
         destination="http://code.google.com/p/jee-examples/"/>
        <tr:goLink text="Author's weblog"
                   destination="http://www.bartkummel.net"/>
      </tr:panelGroupLayout>
    </tr:showDetailItem>
  </tr:panelAccordion>
```

The following image shows how this example could be displayed. Note that in the image, the links line up nicely with the text of the headers. This is not the default. Some CSS had to be applied to the `<tr:panelGroupLayout>` components to achieve this—`inlineStyle="margin-left: 17px;"`. Should you want to change the overall appearance of the accordion, the only way to do so is to use Trinidad's skinning capabilities. (See Chapter 7).



Creating a tabbed panel

The tabbed panel has a lot in common with the accordion. Like the accordion, the `<tr:panelTabbed>` component wraps around multiple `<tr:showDetailItem>` components. The difference is that always exactly one detail item is shown. Another big difference with the `<tr:panelAccordion>` component is the appearance. As the name of the component suggests, the `<tr:panelTabbed>` component renders a set of tabs, where the currently-selected tab is rendered on top of the unselected tabs. Unfortunately, in the default skin of Trinidad, the tabs don't really look like tabs. This can be solved by a custom skin, though. The tab bar can be positioned at the top, bottom, or both by setting the `position` attribute to `above`, `below`, or `both`. The latter is the default value. Here's an example of its use. Note the similarity with the code for an accordion that we just saw.

```
<tr:panelTabbed position="above">
  <tr:showDetailItem text="Apache MyFaces websites">
    <tr:panelGroupLayout layout="vertical">
      <tr:goLink text="Main website"
                 destination="http://myfaces.apache.org"/>
```


www.it-ebooks.info

```
        <tr:goLink text="Wiki"
                   destination="http://wiki.apache.org/myfaces/"/>
      </tr:panelGroupLayout>
    </tr:showDetailItem>
    <tr:showDetailItem text="Book resources">
      <tr:panelGroupLayout layout="vertical">
        <tr:goLink text="Packt publishing"
                   destination="http://www.packtpub.com/"/>
        <tr:goLink text="Example sources"
                  destination="http://code.google.com/p/jee-examples/"/>
        <tr:goLink text="Author's weblog"
                   destination="http://www.bartkummel.net"/>
      </tr:panelGroupLayout>
    </tr:showDetailItem>
  </tr:panelTabbed>
```

This code leads to the layout as displayed in the following screenshot:



Creating a choice panel

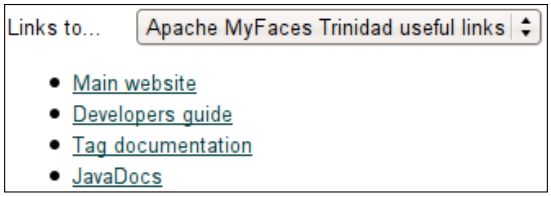
The `<tr:panelChoice>` component is also built around the same principles as the `<tr:panelAccordion>` and `<tr:panelTabbed>` components. It has to have one or more `<tr:showDetailItem>` components as children. The `<tr:panelChoice>` component will be rendered as a combobox with the value of the text attribute of each of the `<tr:showDetailItem>` components' children as options. Whenever an option is chosen, the contents of the corresponding `<tr:showDetailItem>` component are shown. The position of the combobox can be set by the `position` attribute. When set to `top`, the combobox will be above the contents of the `<tr:showDetailItem>` components. When set to `start`, it will be to the left of the contents, or to the right in a right-to-left language. By using the `alignment` attribute, the alignment of the combobox relative to the contents of the `<tr:detailItem>` components can be set to either `top`, `start`, `end`, `bottom`, or `center`. The following is an example:

```
  <tr:panelChoice label="Links to..." position="top"
                  alignment="center">
    <tr:showDetailItem text="Apache MyFaces websites">
      <tr:panelList>
        <tr:goLink text="Main website"
                   destination="http://myfaces.apache.org"/>
        <tr:goLink text="Wiki"
```


www.it-ebooks.info

```
                destination="http://wiki.apache.org/myfaces/"/>
      </tr:panelList>
    </tr:showDetailItem>
    <tr:showDetailItem text="Book resources">
      <tr:panelList>
        <tr:goLink text="Packt publishing"
                   destination="http://www.packtpub.com/"/>
        <tr:goLink text="Example sources"
         destination="http://code.google.com/p/jee-examples/"/>
        <tr:goLink text="Author's weblog"
                   destination="http://www.bartkummel.net"/>
      </tr:panelList>
    </tr:showDetailItem>
  </tr:panelChoice>
```

In the example that we just saw, a `<tr:panelList>` component is used to group the items within the `<tr:showDetailItem>` components. The following image shows how this is rendered:



Creating a radio panel

The `<tr:panelRadio>` component is much like the `<tr:panelChoice>` component. The only difference is that the `<tr:panelRadio>` component renders a list of radio buttons instead of a combobox.

Displaying boxes

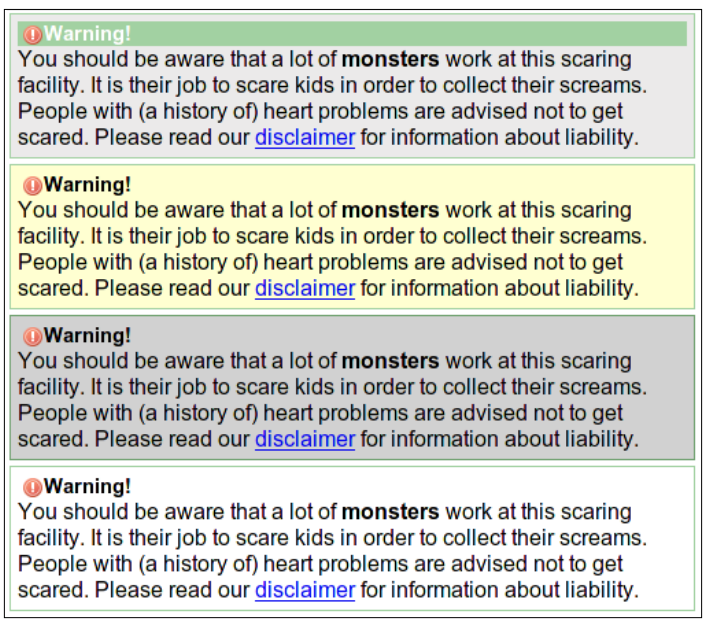
The `<tr:panelBox>` component is, not surprisingly, meant for displaying boxes. A box has some special features, such as the possibility to display an icon (via the `icon` attribute) and a title (via the `text` attribute) on top and four built-in color schemes. By default, a box takes up all available horizontal space and just as much vertical space as needed to display its children. The width can be changed by using the `inlineStyle` attribute and setting it to a fixed or relative width via CSS. The color scheme can be chosen by setting the `background` attribute to `light`, `medium`, `dark`, or `transparent`. The exact meaning of this scheme can be altered by the skin that is in use.

www.it-ebooks.info

In the following example, a `<tr:outputFormatted>` component is used to fill the box with formatted text; but any other component could have been used. Also, the number of child components of the `<tr:panelBox>` component is not limited to one. However, should you want to have more than one child component, you should be aware that you have to use some layout component to lay out the child components in the way that you want.

```
<tr:panelBox icon="../images/exclamation.png"
             text="Warning!" background="light"
             inlineStyle="width:500px;">
  <tr:outputFormatted value="You should be aware that a lot of
&lt;b&gt;monsters&lt;/b&gt; work at this scaring facility. It is their
job to scare kids in order to collect their screams. People with (a
history of) heart problems are advised not to get scared. Please read
our &lt;a href='#'&gt;disclaimer&lt;/a&gt; for information about
liability."/>
</tr:panelBox>
```

Note that some escaped HTML tags are used in the text that fills the `<tr:outputFormatted>` component. The following image shows the box that is produced by this code, where the `background` attribute is set to (from top to bottom) `light`, `medium`, `dark`, and `transparent`:



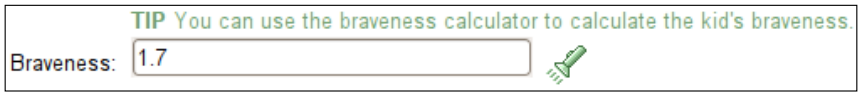
www.it-ebooks.info

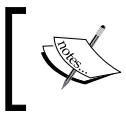
Displaying tips

The tip panel is a fairly simple component as it doesn't have any special attributes. It renders its children in a special style, and adds an indicator to identify it as a tip to the user. This may, for example, be useful if it is not entirely obvious what values may be entered in a form. Let's add a tip to the `EditKid.xhtml` page, to tell the user that he can use the braveness calculator to calculate the braveness of the kid. We add the following code just above the input field for braveness:

```
<tr:panelTip>
  <tr:outputText value="#{msg.bravenessCalcTip}"/>
</tr:panelTip>
```

The result of this is shown in the following image:

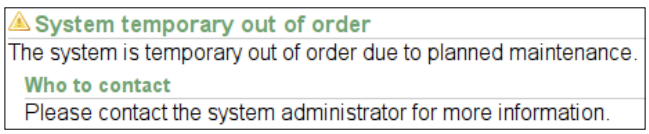




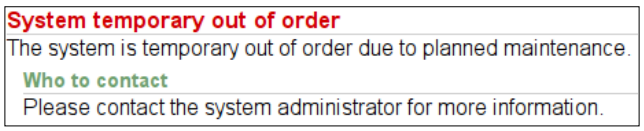
www.it-ebooks.info

```
      <tr:outputText value="Please contact the system
                            administrator for more information. "/>
    </tr:panelHeader>
  </tr:panelHeader>
  <tr:panelHeader messageType="error"
                  text="System temporary out of order" >
    <tr:outputText value="The system is temporary out of order
                          due to planned maintenance."/>
      <tr:panelHeader text="Who to contact">
      <tr:outputText value="Please contact the system
                            administrator for more information. "/>
    </tr:panelHeader>
  </tr:panelHeader>
```

The example that we just saw generates the same content twice. The first time, no `messageType` is set, and thus the default header formatting is used and a custom icon can be added, as can be seen in the following image:



If the `messageType` is set to `error` for the second time, the header will be displayed in another color, as can be seen in the following image:



Using pop ups

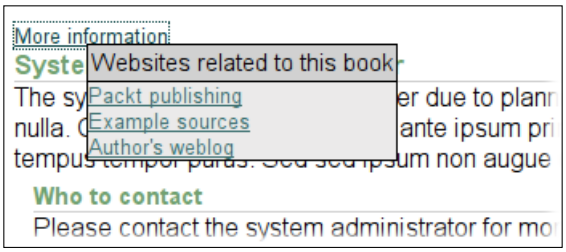
The `<tr:panelPopup>` component renders a clickable link that displays a pop up when clicked. The contents of the pop up can be anything. If more than one child components are used for the contents of the pop up, it is best to use a layout panel to position the children as desired. The `text` attribute sets the text of the link, whereas the `title` attribute sets the text for the title bar of the pop up. If the `title` attribute is omitted, no title bar is added to the pop up. An example of the usage is given next:

```
<tr:panelPopup text="More information"
               title="Websites related to this book">
  <tr:panelGroupLayout layout="vertical">
    <tr:goLink text="Packt publishing"
```


www.it-ebooks.info

```
                destination="http://www.packtpub.com/"/>
      <tr:goLink text="Example sources"
                destination="http://code.google.com/p/jee-examples/"/>
      <tr:goLink text="Author's weblog"
                destination="http://www.bartkummel.net"/>
    </tr:panelGroupLayout>
  </tr:panelPopup>
```

The result of this example is shown in the following image:



Creating button bars

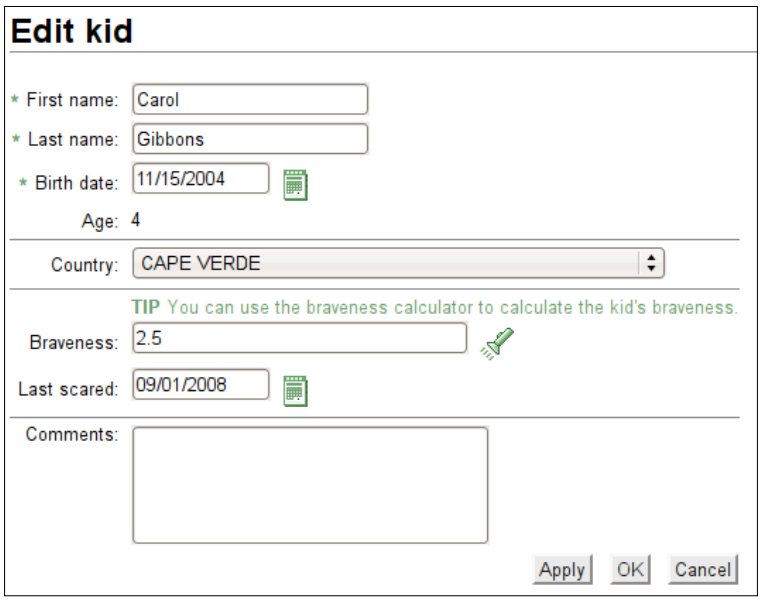
Button bars can be produced easily by using the `<tr:panelButtonBar>` component. It behaves more-or-less the same as a `<tr:panelGroupLayout>` component with `layout` set to `horizontal`. Unlike the group layout component, the button bar component does not have a `separator` facet, but applies some default spacing to buttons. Buttons that are grouped in a `<tr:group>` component are spaced a bit tighter than ungrouped buttons. Another difference is that `<tr:panelButtonBar>` has a semantical meaning, which gives skins the opportunity to apply special formatting to button bars, although the default skin does not take advantage of this opportunity. The `<tr:panelButtonBar>` component has a `halign` attribute that can take a value of `right`, `start`, `left`, `end`, or `center` in order to align the buttons. Depending on the reading direction of the language of the user's browser, `start` is equivalent to `left` or `right` and for `end` it's the other way around. An example of the usage can be found in the `EditKid.xhtml` page:

```
<tr:panelFormLayout>
...
  <f:facet name="footer">
    <tr:panelButtonBar halign="right">
      <tr:commandButton text="#{msg.apply}"
                        id="btnApply" partialSubmit="false"/>
      <tr:commandButton text="#{msg.ok}"
                        action="#{editKidForm.save}"/>
      <tr:commandButton text="#{msg.cancel}"
```


www.it-ebooks.info

```
                          action="cancel" immediate="true" />
      </tr:panelButtonBar>
    </f:facet>
  </tr:panelFormLayout>
```

This results in the button bar at the bottom of the "edit kid" form, which is shown in the following image:



Using caption groups

The caption group panel is meant to group a set of related controls within a form. As such, it can be seen as an alternative to grouping items in a form by using the `<tr:group>` component. Although the `<tr:group>` component doesn't have any means to influence the appearance of the grouping, the `<tr:panelCaptionGroup>` component has a facet called `caption`. This can be used to add a caption to the group, including all extra items that we would like, such as icons or formatted text. Should we want a text-only caption formatted in a default way, we could use the `captionText` attribute of the `<tr:panelCaptionGroup>` component instead, thus eliminating the need for a facet and extra components in it.

As an example, we could change the form in `EditKid.xhtml` to use the `<tr:panelCaptionGroup>` components for grouping:

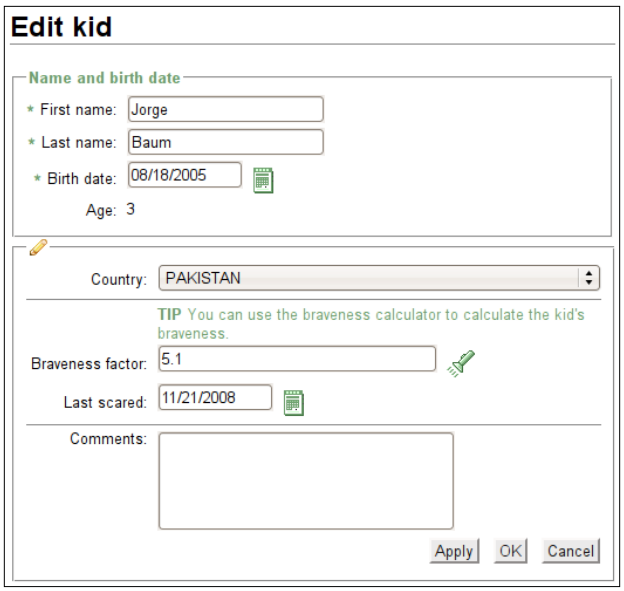
```
<tr:panelCaptionGroup captionText="Name and birth date">
  <tr:panelFormLayout>
```


www.it-ebooks.info

```
      <mias:field id="firstName" required="true"
                  bean="#{editKidForm.selectedKid}"
                  maximumLength="30"/>
...
    </tr:panelFormLayout>
  </tr:panelCaptionGroup>
  <tr:panelCaptionGroup>
    <f:facet "caption">
      <tr:image source="../images/pencil.png"/>
    </f:facet>
    <tr:panelFormLayout>
      <mias:selectField id="country" type="choice"
                        bean="#{editKidForm.selectedKid}"
                        items="#{mias:getCountries()}"
                        itemValue="name" itemLabel="name" />
...
      <f:facet name="footer">
        <tr:panelButtonBar halign="right">
...
        </tr:panelButtonBar>
      </f:facet>
    </tr:panelFormLayout>
  </tr:panelCaptionGroup>
```

In the previous example, we've used two `<tr:panelCaptionGroup>` components. The first one uses the `captionText` attribute to set the caption, whereas the second one uses the `caption` facet to set an icon as the caption. Note that the `<tr:panelFormLayout>` has to be inside the `<tr:panelCaptionGroup>` component, which requires an extra `<tr:panelFormLayout>` component. That's a problem, because the two `<tr:panelFormLayout>` components will not line up their labels nicely, as can be seen in the next image. This, combined with the fact that nowadays user interface specialists advise the use of simple separator lines over boxes like the caption group, the `<tr:panelCaptionGroup>` component does not seem to be a very usable component. Using the `<tr:group>` component within a single `<tr:panelFormLayout>` component seems to be a much more elegant way of grouping fields.

www.it-ebooks.info



Creating bulleted lists

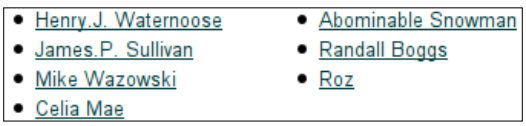
The list panel component is an easy way to make bullet lists. All child items of the `<tr:panelList>` component are stacked vertically, and are prefixed with a bullet. Longer lists can be split up and spread across multiple columns. This works more-or-less the same as with the `<tr:panelFormLayout>` component. The `rows` attribute sets the number of rows per columns, and the `maxColumns` attribute sets the maximum number of columns. The `<tr:panelList>` component tries to obey both settings, but the `maxColumns` attribute takes precedence. This means that the number of rows per column may be larger than the `rows` attribute if the maximum number of columns is reached. When the items are distributed over multiple columns, the `<tr:panelList>` component tries to distribute them evenly.

As an example, we could replace the table on the `Employees.xhtml` page with a simple bulleted list of employees, where each employee's name can be clicked on, to link to the edit page for that employee.

```
<tr:panelList rows="5">
  <c:forEach var="emp" items="#{empsList.employees}">
    <tr:commandLink action="edit">
      <tr:outputText value="#{emp.firstName} #{emp.lastName}" />
      <tr:setActionListener from="#{emp}"
                            to="#{pageFlowScope.selectedEmployee}" />
    </tr:commandLink>
  </c:forEach>
</tr:panelList>
```


www.it-ebooks.info

In this example, the rows attribute is set to `5`. In the following image, the list has seven employees. To keep the number of rows below six and distribute the items as evenly as possible, the first column has four employees and the second has three.



Lay out a page by using the panel page component

It seems that the `<tr:panelPage>` component was designed as a work-around for the lack of templates in JSF. It has a lot of facets with names such as `navigation1`, `appCopyright`, and so on. Each facet will be placed at a certain position on the page. All facets surround the direct children of the `<tr:panelPage>` component, which should be the main content of the page. As we have a good templating solution with Facelets, the use of this component no longer seems necessary. For completeness, a list of all facets is given in the following table:

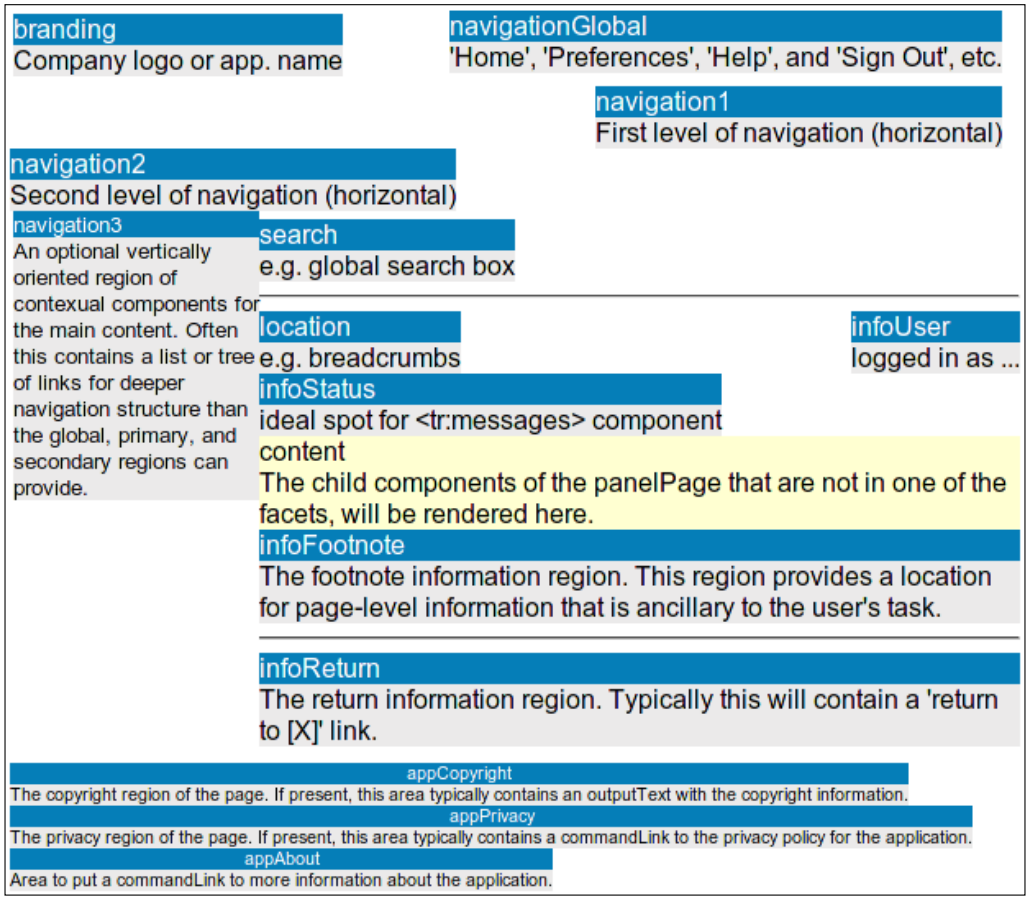
Facet name	Description
`appCopyright`	Area for copyright information. This will be rendered at the bottom of the page, just above the contents of `appPrivacy`. In the default skin, this area is rendered with a smaller font.
`appPrivacy`	Area for privacy disclaimers and the like, which is rendered at the bottom of the page, between the `appCopyright` and the `appAbout` contents. Also in a smaller font, just like `appAbout` and `appCopyright`.
`appAbout`	Area meant for information about the application, which is typically a link to the vendor or the IT department. This will be rendered at the bottom of the page. In the default skin, this area is rendered with a smaller font.
`branding`	This is the area where the (company) logo and the name of the application should go. It's rendered in the top-left corner of the page.
`infoFootnote`	Region reserved for footnotes specific to a certain page. This is rendered at the bottom of the page, above the copyright information.

www.it-ebooks.info

Facet name	Description
infoReturn	Area reserved for a "return to some page" link. This is rendered below the footnote, and above the copyright. In the default skin, a separator line is rendered above this area.
infoStatus	Area for status information. This seems the ideal place for a <tr:messages> component. It is rendered just above the main content.
infoUser	This area is reserved for user information, such as the username of the logged-in user, a logout link, and/or a link to the user's profile. This is rendered just above the infoStatus contents, and aligned to the right.
location	Reserved for location information. This is the place for breadcrumbs or process trains, and is rendered just above the infoStatus contents, and left aligned.
navigation1 navigation2 navigation3	The <tr:panelPage> component has room for three levels of navigation, in addition to the navigationGlobal area. navigation1 is rendered right aligned, just below the navigationGlobal content. navigation2 is rendered below navigation1, and is left aligned. navigation1 and navigation2 are meant for horizontal-oriented navigation components such as tab bars and the like. navigation3 is rendered below navigation2 and to the left of the main content, and is meant for vertical-oriented navigation components such as tree menus.
navigationGlobal	This section is meant for global navigation, such as "home", "sign out", "help", and so on. This is rendered in the top-right corner of the page.
search	The search region of the page. This is meant for application or system-wide searches, and is rendered below navigation2 and to the right of navigation3. In the default skin, a separator line is rendered below the search area.

www.it-ebooks.info

The following image shows the positioning of the facets:



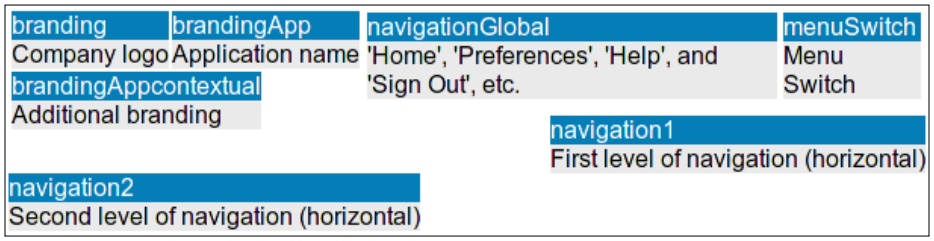
Using the page header panel

The `<tr:panelPageHeader>` component is much like the `<tr:panelPage>` component, except that it is only meant for the page header and not for the whole page. Child components that are not in one of the facets don't get rendered. The following table lists all of the facets, along with descriptions of them:

www.it-ebooks.info

Facet name	Description
`branding` `brandingApp` `brandingAppContextual`	This component defines three levels of branding. The first, `branding`, will be rendered in the top-left corner of the page. The second, `brandingApp`, will be rendered just to the right of the first-level branding. `brandingAppContextual` will be rendered below the first two levels.
`menuSwitch`	This is rendered in the top-right corner. This area is meant for some global pull-down menu.
`navigation1`	This is rendered right-aligned, just below the `menuSwitch` contents.
`navigation2`	This is rendered left-aligned, below the three branding areas.
`navigationGlobal`	This is meant for global navigation, as on the `<tr:panelPage>` component. It is rendered just to the left of the `menuSwitch` contents.

The following image gives an idea of the positioning of the facets:



Summary

In this chapter, we saw how to use the many components of Apache MyFaces Trinidad to build usable and consistent-looking pages. We also learned how to use the common features that most Trinidad components share. We learned how useful it is to have a well-designed set of components that are all designed around the same principles. The more advanced topics of Trinidad are covered in the next chapter.

www.it-ebooks.info

www.it-ebooks.info

6
Advanced Trinidad

Apache MyFaces Trinidad is an extensive JSF library that goes far beyond supplying fancy JSF components. This chapter continues where the previous chapter left off, covering the more advanced features of Trinidad.

After reading this chapter, you will be able to:

- Add charts to your user interface
- Pass data from one page to another using the `pageFlowScope`
- Make your user interface more responsive and interactive using Partial Page Rendering (PPR, aka AJAX)
- Create pop-up dialogs
- Perform validation and conversion on the client side; write, test, and debug JavaScript code for client-side validation and conversion

Data visualization

Trinidad has a data visualization component that is able to visualize numerical data in an appealing way. This component relies on a special data model that we have to implement. This section focuses on implementing that data model, and also gives an overview of the most important options of the visualization component itself.

www.it-ebooks.info

Creating the data model

The chart component expects a number of methods to be present in order to get the data to be visualized. One would expect these methods to be defined in a Java interface, as is common practice. However, the Trinidad project only supplies an abstract class that has to be extended, which is the `org.apache.myfaces.trinidad.model.ChartModel` class. This class defines three methods that have to be implemented regardless of which visualization type will be chosen. Apart from these three methods, the class has a number of methods that are not abstract, but that can be overridden in a subclass. Some of them are needed for some specific visualization types, whereas others are meant to provide optional data such as extra labels.

Understanding the terminology

Before we start implementing a data model, it's good to have a look at the terminology used in the Trinidad `ChartModel`. Here's a list of the most important terms used:

- **Series**: This is a list of data points that can be shown in a chart. Often, multiple series are shown in a single chart. For example, if the quarterly results over the past two years of two companies are compared, there's one series for each company.
- **Group**: This is a set of data points from multiple series that share a position on the horizontal axis. In the quarterly results example, there's a group for each quarter.
- **X axis**: This is the axis where the groups are displayed. Most of the time, this will be the horizontal axis. However, there are some chart types where the axes are rotated, for example with a "horizontal bar" graph. For rotated graphs, the X axis is the vertical axis.
- **Y axis**: This is the axis where the values are shown. Most of the time, this is the vertical axis, except for the rotated graphs.

Implementing a minimal data model

Let's start by implementing a minimal data model that is useful for most of the visualization types. Suppose we want to create a chart that displays the average braveness factor of kids per age. We could create a class like this:

```
public class AgeVsBraveness extends ChartModel {
  private List<Kid> kids;
  private List<List<Double>> data;

  private void calculate() {
    // do some math...
  }
```


www.it-ebooks.info

```
    @Override
    public List<String> getGroupLabels() {
      calculate();
      List<String> x = new ArrayList<String>();
      for(int i = 0; i < data.size(); i++) {
        x.add("" + i);
      }
      return x;
    }

    @Override
    public List<String> getSeriesLabels() {
      calculate();
      List<String> x = new ArrayList<String>();
      x.add("Braveness");
      return x;
    }

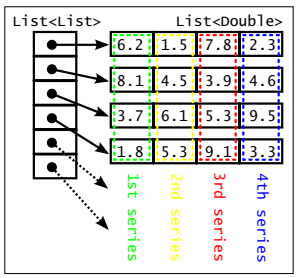
    @Override
    public List<List<Double>> getYValues() {
      calculate();
      return data;
    }
  }
```

We assume that `kids` references the main kids list of the application. The `calculate()` method will calculate the average braveness factor for each age. The result is put into a two-dimensional structure, a `List` of `Lists`. The inner `Lists` are `Lists` of `Doubles`. This data structure is the required return type of the `getYValues()` method. Therefore, this method can be fairly simple, as long as we know that the `calculate()` method will fill the structure.

The idea is that a single chart can visualize several series of data. Perhaps one would expect that every `List` of `Doubles` represents one series of data, but it is a bit more complicated than that. Every Nth value in every `List` of `Doubles` contains a value of series N. So if we have a single series of data, we need as many `Lists` of `Doubles` as we have values, and every `List` has only one `Double` value in it. If we have more than one series, this can be confusing sometimes.

www.it-ebooks.info

The following image tries to clarify it a bit by giving an example:



The `getSeriesLabels()` method should return a `List` of `Strings` containing a name for each series. So this list should contain exactly as many `Strings` as there are values in our `Lists` of `Doubles`. In our case, this list contains only one `String`. However, note that we cannot return a simple `String`; we have to wrap it in a `List`.

The third and last method that has to be implemented is the `getGroupLabels()` method. This method also has to return a `List` of `Strings`, but this list should contain the labels that will be displayed on the X axis. So the length of this list should be the same as the number of `Lists` of `Doubles` in the data set. In our example, we know that every element in a data series corresponds to an age. So, we can simply start counting at zero and add one for every label until we have as many labels as the `Double` values in the data set. This is exactly what the code in the example does. But because the group labels are returned as a `List` of `Strings`, we are not limited to numerical labels.

Calculating the values

It's nice to have labels and everything, but one of the most important things of a chart is the data. Let's see how we can calculate the values that we need for our "Age Vs. Braveness" chart. In the previous example, we created a `calculate()` method that did the math. Now let's see how we can implement this method:

```
private void calculate() {
  data = new ArrayList<List<Double>>();

  for(int i = 0; i < 12; i++) {
    data.add(new ArrayList<Double>());
    data.get(i).add(0.0);
  }

  for(Kid kid: getKids()) {
    int age = kid.getAge();
    double braveness = kid.getBraveness();
```


www.it-ebooks.info

```
        double oldValue = data.get(age).get(0);
        double newValue;

        if (0.0 == oldValue) {
          newValue = braveness;
        } else {
          newValue = (oldValue + braveness) / 2.0;
        }

        data.get(age).set(0, newValue);
      }
    }
```

The first `for` loop creates 12 `Lists` of `Doubles`, and fills each list with one double. The second loop iterates over all `Kid` objects. Depending on the age of the kid, the braveness factor of the kid is used to update the average braveness of all kids with the same age. Note that comparing `double` values with `==` is generally not a good idea. In this example, there's no problem because we compare with `0.0`.

Initializing the data model

Now that we've implemented a data model for our graph, we can start using it to create the graphs. Before adding a graph to a page, we have to make sure the `<tr:graph>` component that we're going to use can access our calculated data. The most elegant way of doing this is to use a backing bean. So let's create a simple backing bean:

```
package inc.monsters.mias.backing;

import inc.monsters.mias.data.Kid;
import inc.monsters.mias.data.statistics.AgeVsBraveness;

import java.util.List;

public class Statistics {
  private AgeVsBraveness ageVsBraveness;
  private List<Kid> kids;

  public List<Kid> getKids() {
    return kids;
  }

  public void setKids(List<Kid> kids) {
    this.kids = kids;
  }

  public AgeVsBraveness getAgeVsBraveness() {
    if(null == ageVsBraveness) {
      ageVsBraveness = new AgeVsBraveness();
```


www.it-ebooks.info

```
        ageVsBraveness.setKids(getKids());
      }
      return ageVsBraveness;
    }
  }
```

This backing bean has a `List` of `Kids` that is passed to the `AgeVsBraveness` object to provide it with data. In the highlighted `getAgeVsBraveness()` method, we use lazy initialization to create an `AgeVsBraveness` object only once when needed. Now we have to declare this backing bean in our `faces-config.xml` file and make sure the `kids` property of the bean gets a reference to the main list of kids. This can be done as follows:

```
<managed-bean>
  <managed-bean-name>statistics</managed-bean-name>
  <managed-bean-class>inc.monsters.mias.data.backing.Statistics
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>kids</property-name>
    <property-class>java.util.List</property-class>
    <value>#{kidsList.kids}</value>
  </managed-property>
</managed-bean>
```

We use expression language to refer to another managed bean that we use to initialize the `kids` property of our bean.



You may wonder why we don't instantiate an instance of `AgeVsBraveness` as a managed bean directly. While this can be done, it is not very elegant—especially when we want to use more than one `ChartModel` implementation in a single page. In this case, we can simply add another member to our `statistics` backing bean, instead of having to create an extra managed bean for every `ChartModel` that we add. The source code that can be downloaded from the author's website (`http://www.bartkummel.net`) has an example of a page that uses multiple `ChartModels`.

Adding a graph to a page

Everything is now set to add a graph to a page. We can either add it to an existing page, or create a new page for it. Whatever we do, adding the graph to a page is as simple as adding a single JSF component to that page:

```
<tr:chart value="#{statistics.ageVsBraveness}"
          type="verticalBar"
          inlineStyle="width:800px; height:600px;"/>
```


www.it-ebooks.info

The `value` attribute takes a reference to an object that subclasses the `ChartModel` class. In our case, this is the `ageVsBraveness` member of the `statistics` bean we created earlier. The `type` attribute sets the type of chart that will be generated. The *Chart types* section shows an overview of all available types. The `<tr:chart>` component itself does not have any size attributes. As shown in the next example, we can use `inlineStyle` instead, to set a size.

Changing data display

The `<tr:chart>` attribute has a lot of attributes that change the way the data is displayed in a chart. The following table gives an overview of these attributes:

Attribute	Description
`XMajorGridLineCount`	By default, for every "label" that is returned by the `getGroupLabels()` method, a grid line is drawn. By setting this attribute to a value other than `-1`, this behavior can be overridden. We can set any positive number of grid lines. The lines will be distributed evenly, regardless of the number of group labels.
`YMajorGridLineCount`	Controls the number of grid lines on the vertical axis. The default value is `3`.
`YMinorGridLineCount`	Meant to set the number of secondary (minor) grid lines between two primary (major) grid lines. Currently only works with "gauge" charts.
`maxPrecision`	Sets the number of decimals for the values that are displayed on the Y axis. The default is `0`, which means that no decimals are displayed.

Changing the looks

There are a lot of attributes that can be used to change the look of the generated graph. The following table lists them all:

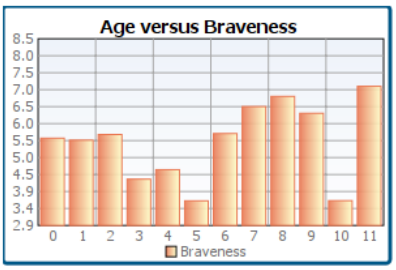
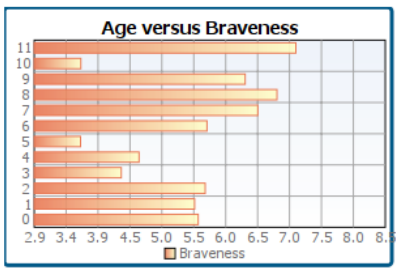
Attribute	Description
`animationDuration`	By default, the chart will be animated the first time it is displayed. This attribute sets the duration of the animation in milliseconds. Set it to `0` to disable animation. The default value is `1000`.
`gradientsUsed`	For chart types that have areas that are filled with colors, such as bar and pie charts. If set to `true`, which is the default, the areas are filled with a gradient. If set to `false`, the areas are filled with a solid color.

www.it-ebooks.info

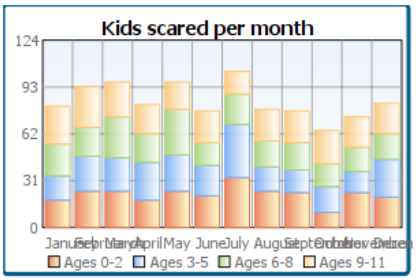
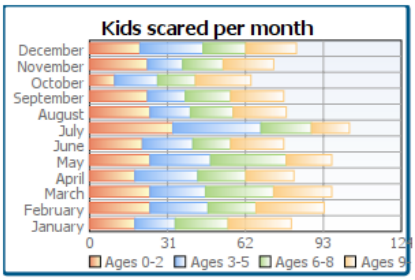
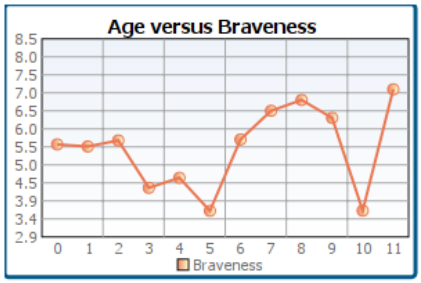
Attribute	Description
`legendPosition`	The position of the legend. The legend (or key) helps the user to determine which color is used to visualize each data series. Valid values are `none`, `bottom`, `end`, `top`, and `start`. `none` will hide the legend, `start` and `end` will put the legend on the left or right of the chart, depending on the reading direction of the current language. The default is `bottom`.
`perspective`	By default, the charts are rendered with a semi-3D look. This can be disabled by setting `perspective` to `false`.
`tooltipsVisible`	Specifies whether or not tool tips should be displayed if the mouse cursor hovers over the chart. If set to `true`, a tool tip with the name of the series and the value will be displayed if the mouse pointer is over a data point in the chart. The default value is `true`.

Chart types

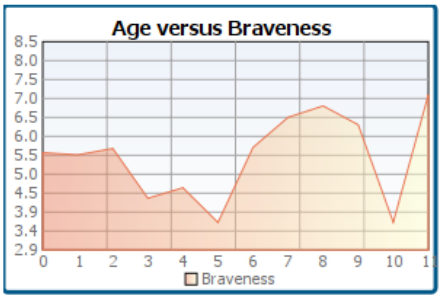
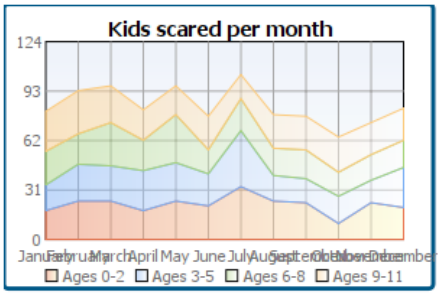
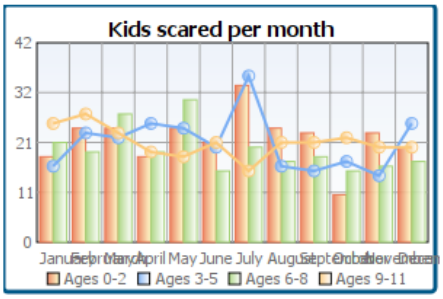
The `<tr:chart>` component can render different types of charts. The following table lists all possible types, along with an example of each type and some explanatory notes:

`type` and example	Notes
 `verticalBar`	A simple bar chart. If more than one series of data is available, each label on the X axis gets several bars in different colors.
 `horizontalBar`	This is basically a rotated version of the normal bar chart. Note that although the values are returned by a method called `getYValues()`, the values are actually displayed on the X axis in this chart.

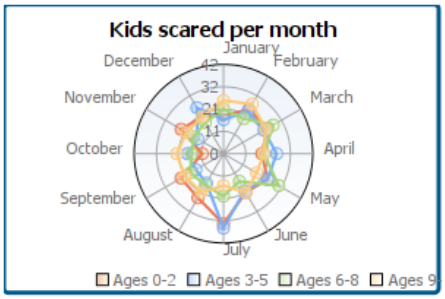
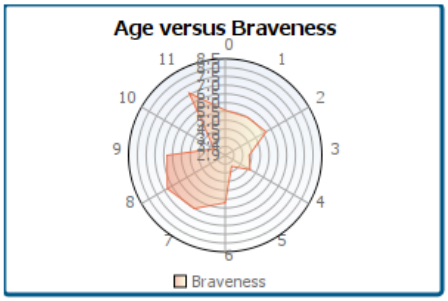
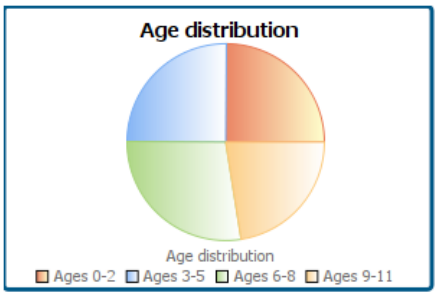
www.it-ebooks.info

`type` and example	Notes
 `stackedVerticalBar`	A stacked version of the standard bar chart. If only one series of data is available, the result is exactly the same as with the normal bar chart.
 `stackedHorizontalBar`	A rotated version of the `stackedVerticalBar` type.
 `line`	This chart type plots the data points on the grid and draws a line between them.

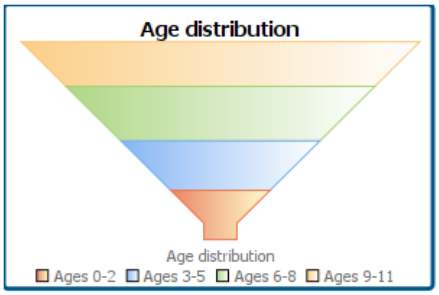
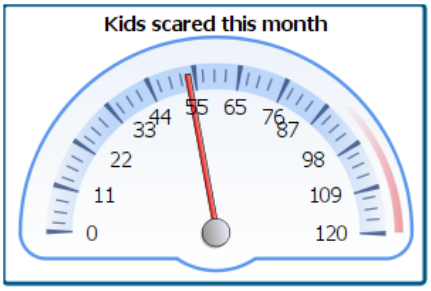
www.it-ebooks.info

`type` **and example**	**Notes**
	This is nearly the same as the line type, but now the area below the line is filled.
`area`	
	A stacked version of the area chart.
`stackedArea`	
	A combination of bars and lines. The even data series (0, 2, 4, 6, ...) are plotted as bars and the odd data series (1, 3, 5, 7, ...) are plotted as lines.
`barLine`	

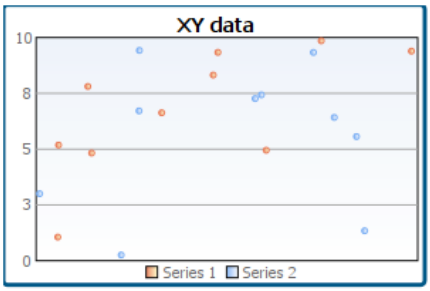
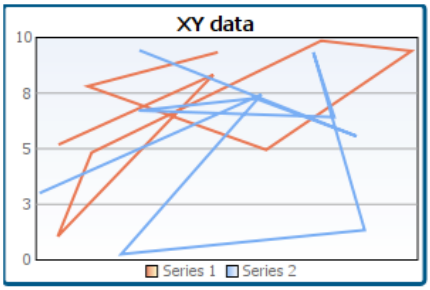
www.it-ebooks.info

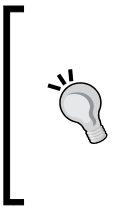
`type` and example	Notes
 `radar`	A radar plot.
 `radarArea`	A radar plot where the area within the line gets filled. It is possible to use multiple data series, but the result is hard to read as all colors blend together in the center.
 `pie`	The `pie` type renders as a normal pie chart. All data should be in the first `List` of `Doubles`. If the data model returns more than one `List` of `Doubles`, more pie charts will be rendered—one for each `List`. Note that no values are displayed on the chart; the value of a part will be shown if the mouse pointer is positioned above the part.

www.it-ebooks.info

`type` **and example**	**Notes**
 `funnel`	The funnel chart behaves the same as a pie chart, only the form factor is different.
 `circularGauge`	The `circularGauge` mode can be used to visualize a single value. An example usage could be to display a system status on a start page. As there is only a single value, it is not possible for this chart type to automatically guess a good value for the minimum and maximum values on the scale. Therefore we should make sure that the `getMaxYValue()` and `getMinYValue()` methods in our `ChartModel` subclass return the correct values.
 `semiCircularGauge`	The `semiCircularGauge` mode behaves exactly in the same way as the `circularGauge` mode, except for the fact that it renders the gauge in a semi-circle instead of a circle.

www.it-ebooks.info

`type` **and example**	**Notes**
 `scatterPlot`	The `scatterPlot` lets you define data points by their X and Y values. To use this type of chart, the `getXValues()` method should return exactly as many values as the `getYValues()` method does. The values at the same position in both data sets form a pair that together determine the location of a dot in the scatter plot. A scatter plot can have multiple data series.
 `XYLine`	A `XYLine` plot is more or less the same as a scatter plot, but with the dots connected by a line.



www.it-ebooks.info

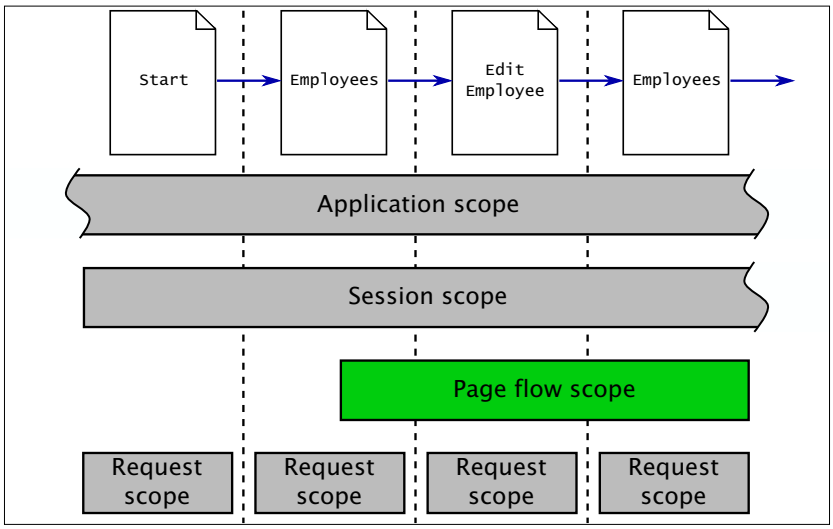
The Trinidad charting component was added to Trinidad after it was donated to Apache. For that reason, it is not (yet) as tightly integrated as the other Trinidad components. For example, the `<tr:chart>` component does not use Trinidad's skinning capabilities to change its look. On a more subjective note, it seems the chart component is not as well thought through and rock solid as other parts of Trinidad. The component could be improved, for example, by using Java interfaces where applicable, and by better hiding the internal implementation from the programmer. That said, `<tr:chart>` is still a useful component for adding charts to a user interface in a not too complicated way. And, of course, anyone is free to submit improvements to the Trinidad project.

Passing on data with page flows



www.it-ebooks.info

Imagine that we have an "Employees" page with a table, as shown in the previous image. If the user clicks on the pencil icon in one of the rows, he'll navigate to an "Edit Employee" page where the data for the employee that was on the row where he clicked can be edited. The described navigation is shown at the top of the next image. The user starts at the "Start" page, navigates to "Employees", goes to "Edit Employee" by selecting an employee from the table, and then goes back to "Employees" once he has finished editing the selected employee:



Below the navigation in the image, the bars indicate the lifetime of the various scopes. The **application scope** is already there if the user starts his navigation at the "Start" page. At that moment, the session scope is created and stays alive as long as the user keeps using the system. There's a new request scope created every time a new page is requested from the server, and the request scope only lives during the processing of this request. The page flow scope starts its life when the user selects an employee to edit. The page flow scope stays alive until the user navigates away from the page on which it was created, in this example this is the "Employees" page. To cut a long story short, the page flow scope does live exactly as long as needed.

www.it-ebooks.info

What is not reflected in the image is what happens if the user opens a link in a new window or browser tab. Imagine that the user opens the "Edit Employee" link in a new browser tab. He or she returns to the "Employees" page and uses the "edit employee" link of another employee to open the "Edit employee" page for that employee in another browser tab. Of course, the user expects to be able to edit both selected employees without any problem. If we had used the session scope to store the selected employee, the first selected employee gets overwritten by the second, as there is only one session scope for the user. Using the page flow scope can solve this problem. As we will see, a new page flow scope will be created every time an "edit employee" link is used. So in our example, both new browser tabs will get their own page flow scope, preventing the selected employee from being overwritten by any next selection. In this way the user can edit multiple employees simultaneously, in different browser tabs.

There's one thing that we'll have to keep in mind, though. As the page flow scope is not a standard part of JSF 1.2, the page flow scope does have some limitations:

- We cannot define managed beans in `pageFlowScope` from our `faces-config.xml` file.
- We cannot use variables that are stored in the page flow scope without prepending them with `pageFlowScope`. For example, if we have a variable `name` stored in the session scope, we can refer to it as `#{name}`. But if we store the same variable in the page flow scope, we have to refer to it as `#{pageFlowScope.name}`.

Now let's see how we can use the page flow scope in our MIAS application. Let's start with the table. The part where the column with the pencil icons gets defined could look as follows:

```
<mias:column columnName="edit" headerName="emptyTableHeader"
custom="true">
<tr:commandLink action="#{empsTable.edit}"
immediate="true">
<tr:image source="../images/pencil.png"
inlineStyle="border-width: 0px;" />
</tr:commandLink>
</mias:column>
```

Note the `<tr:commandLink>` component, where an `action` is set. This refers to a method in the `empsTable` backing bean. This method has to get the value of the `emp` variable for the row in the table where the user clicked. Then, that value has to be copied to a variable that can be used by the edit page:

```
public String edit() {
FacesContext context = FacesContext.getCurrentInstance();
ELResolver elr = context.getApplication().getELResolver();
```


www.it-ebooks.info

```
    Employee employee = (Employee)
                      elr.getValue(context.getELContext(), null, "emp");

    if (employee == null) {
      return null;
    }

    RequestContext requestContext = RequestContext.getCurrentInstance();

    requestContext.getPageFlowScope().put("selectedEmployee",
                                          employee);

    return "edit";
  }
```

While this code is not too complicated, it still involves a lot of steps just to copy a value. Luckily, Trinidad also offers us the `<tr:setActionListener>` component. This component is made for this task. We can just add it as a child of our `<tr:commandLink>` component:

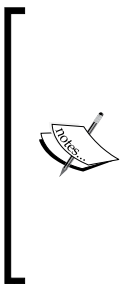
```
<mias:column columnName="edit" headerName="emptyTableHeader"
             custom="true">
  <tr:commandLink action="edit" immediate="true">
    <tr:image source="../images/pencil.png"
              inlineStyle="border-width: 0px;" />
    <tr:setActionListener from="#{emp}"
                          to="#{pageFlowScope.selectedEmployee}" />
  </tr:commandLink>
</mias:column>
```

This action listener component will take the value of the variable in the `from` attribute and copy it to the variable in the `to` attribute. Now we don't need Java code in our backing bean to do this copy operation. Regardless of the method that we use—Java code or `<tr:setActionListener>`—a new page flow scope gets created as soon as we put our selected employee into it. Once we've copied the selected employee into the `selectedEmployee` variable in the page flow scope, we can refer to it in our edit page. For example, to create an edit field for the last name of the employee, we could add the following to our edit page:

```
<mias:field id="lastName" required="true"
            bean="#{pageFlowScope.selectedEmployee}"/>
```


www.it-ebooks.info

Using AJAX and Partial Page Rendering



www.it-ebooks.info

With PPR, we can tell a `<tr:commandButton>` component (or a `<tr:commandLink>` component) to do a **partial submit** instead. In this case, a piece of JavaScript will post the form data to the server and wait for the response from the server while the browser keeps displaying the page. After the JavaScript code receives the answer from the server, it will update the page accordingly. You can see this happening when you push a button and the progress indicator of your browser doesn't start moving.

We can use network monitoring software to intercept the requests to the server and the answers from the server, in order to investigate the difference more quantitatively. Eclipse has a built-in TCP / IP monitor that can be used for this. With this tool, we can look at the data traffic generated by a single push on the **Apply** button on the `EditKid.xhtml` page of our example application. The following table lists the results of a single measurement with this tool:

	Partial submit	Full submit	Factor
Number of requests	1	3	3.0
Total bytes sent	900	1,884	2.1
Total bytes received	661	40,559	61.4

While this is by far not a scientific measurement, it is clear that we can save a lot of data traffic between the browser and the server by using partial submits. Enabling partial submits is as simple as setting the `partialSubmit` attribute of a `<tr:commandButton>` or `<tr:commandLink>` component to `true`.

While this can be a nice way to make web applications more responsive, this is not the "sexy" AJAX stuff that people are looking for. So let's explore some more PPR possibilities.

Using the autoSubmit and partialTriggers attributes

One of the typical AJAX-like tricks is to update a part of a page without the need for the user to press a button. This can be achieved by using the `autoSubmit` attribute on a Trinidad input component. By setting this attribute to `true`, the value will be submitted to the server every time the user edits the contents, without the need for a button to be pressed. Let's see how we can apply this in a simple example. In the `EditKid.xhtml` page in our example application, we have a date field where the birth date of a kid can be entered and we also have a read-only field that displays the age of the child. Until now, the age of the child was updated only when we clicked on the **Apply** button.

www.it-ebooks.info

To enable automatic submission of the value of the birth date field, we have to set the `autoSubmit` attribute on that component to `true`. But that's not the only thing we have to do. We also have to tell the age component to update itself whenever the birth date gets updated. This can be done by setting the `partialTriggers` attribute of the age field. This attribute should contain a list of IDs of components to "listen" to. So in our case, the `id` of the component to listen to is `birthDate`. Let's see how this sums up in our page:

```
<mias:dateField id="birthDate"
                bean="#{editKidForm.selectedKid}"
                required="true"
                popup="true"
                autoSubmit="true" />
<mias:field     id="age"
                bean="#{editKidForm.selectedKid}"
                readOnly="true"
                partialTriggers="birthDate"/>
```

The additions that are needed to let the age be automatically updated whenever the birth date changes are highlighted. Note that we have to make sure that our Facelets composition components "forward" the `autoSubmit` and `partialTriggers` values to the Trinidad components that they use internally. For example, the definition of the `<mias:dateField>` component should look like this:

```
<c:if test="#{empty autoSubmit}">
  <c:set var="autoSubmit" value="false" />
</c:if>
<tr:inputDate value="#{bean[id]}" id="#{id}"
              required="#{required}" readOnly="#{readOnly}"
              label="#{msg[id]}:"
              autoSubmit="#{autoSubmit}">
  <tr:convertDateTime pattern="#{msg.datePattern}"/>
  <ui:insert />
</tr:inputDate>
```

The first three highlighted lines make sure that a default value is set if the `autoSubmit` attribute was not set in the calling page. Then we "forward" the value of the `autoSubmit` variable to the `autoSubmit` attribute of the `<tr:inputDate>` component in the fourth highlighted line.

www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

Working with partialTriggers and naming containers

Some components are **naming containers**. This means that their `id` attribute is required. Appendix B contains a list of all of the Trinidad components, and whether or not they are naming containers. To refer to other components, for example the `partialTriggers` attribute, some special rules apply with regards to naming containers. We are going to apply these rules to an example. Assume that we have the following page structure:

```
<tr:form id="form">
<tr:subform id="subForm1">
<tr:subform id="subForm2">
<tr:inputText id="input1" autoSubmit="true" />
</tr:subform>
</tr:subform>
<tr:subform id="subForm3">
<tr:subform id="subForm4">
<tr:inputText id="input2" />
</tr:subform>
</tr:subform>
</tr:form>
```

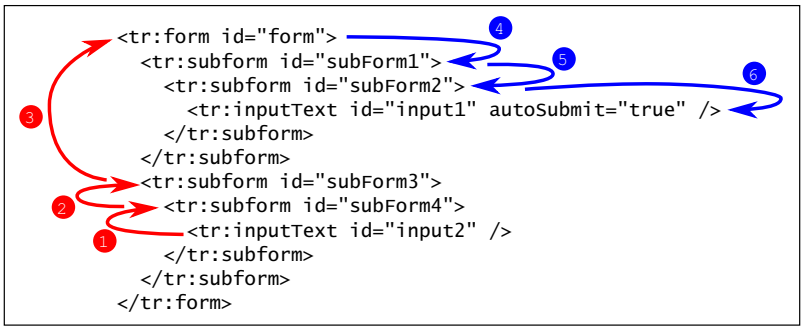
Now if we want to add a `partialTriggers` attribute to the second `<tr:inputText>` component, and we want that to be triggered by the first `<tr:inputText>` component, how should we refer to that component? We have to traverse the page hierarchy from where we are to the component that we want to refer to. We first go up the hierarchy until we reach a component that both components have as a parent; in our example this would be the `<tr:form>` component. Then we go down through the hierarchy again, until we reach the component that we want to refer to.

While traversing through the hierarchy, we assemble the "path" to our component as follows:

- To pop out the current naming container, two colons have to be added to the path
- For each additional naming container that we pass on our way up through the hierarchy of the page, we have to add an extra colon
- As we reach the highest level that is needed before we can go down through the hierarchy, we add the ID of that naming container, followed by a colon
- For each naming container that we pass on our way down the hierarchy, we have to add its ID, followed by a single colon
- We end, of course, with the ID of the component that we want to refer to

www.it-ebooks.info

Now let's traverse, step-by-step, through the hierarchy of our example, as shown in the following image:



The following table shows what is added to our "path" for each step:

Step	Add to path	Explanation
1	(nothing)	Going up without passing a naming container.
2	`::`	Popping out the first naming container, add a double colon.
3	`:`	Popping out another naming container, add a single colon.
4	(nothing)	`form` is the first element that has both components as child. As it is not a naming container, we don't have to add its ID to the path.
5	`subForm1:`	Going down in the hierarchy and passing a naming container, add its name followed by a colon.
6	`subForm2:`	Going down in the hierarchy and passing a naming container, add its name followed by a colon.
7	`input1`	Finally reaching the component that we want to refer to, add its ID to the path.

Having followed these steps, our complete path now is `:::subForm1:subForm2:input1`. So if we add it to the component, our code becomes:

```
<tr:form id="form">
  <tr:subform id="subForm1">
    <tr:subform id="subForm2">
      <tr:inputText id="input1" autoSubmit="true" />
    </tr:subform>
  </tr:subform>
  <tr:subform id="subForm3">
    <tr:subform id="subForm4">
      <tr:inputText id="input2"
                    partialTriggers=":::subForm1:subForm2:input1"/>
```


www.it-ebooks.info

```
    </tr:subform>
  </tr:subform>
</tr:form>
```

Creating a status indicator

As stated earlier, the status indicator of the browser will not move when data is transferred through partial requests. Should we want to show the user that something is happening in the background—for example, if we know that the retrieval of data will take a long time—there is a special status indicator component that can be used: `<tr:statusIndicator>`. We can just put one or more of these on our page and it will automatically show the user that there is something going on in the background. The indicator has only two states: "ready" and "busy". By default, the status indicator will show a round icon that starts to spin when the state is "busy". This appearance can be overridden by the use of Trinidad's skinning capabilities—as with every Trinidad component. (See the next chapter for an introduction to skinning.) If we would like to have a status indicator on every page, it would be a good idea to put the status indicator in our page template.

Using the addPartialTarget() method

Sometimes we may want to trigger a partial refresh of a component from within the Java code of one of our backing beans. For example, we may have a data table on our page that we want to refresh only if we know that the data has really changed. In this case, we can call the `addPartialTarget()` method on the current `RequestScope` object. We can also use this method to partially refresh a non-Trinidad component that doesn't have a `partialTriggers` attribute. Let's look at an example to see how to use this method.

On the `Kids.xhtml` page, we have a table showing all of the kids in the database. We can remove kids from the system by selecting them in the table and then clicking on the **Delete** button. Currently, the button performs a full submit of the page. We can change this into a partial submit, as follows:

```
<tr:table var="kid" value="#{kidsList.kids}" rows="20"
          id="kids" rowSelection="multiple"
          binding="#{kidsTable.table}">
...
  <f:facet name="actions">
    <tr:commandButton
                actionListener="#{kidsTable.deleteSelected}"
                text="#{msg.delete}"
                partialSubmit="true"/>
```


www.it-ebooks.info

```
    </f:facet>
  </tr:table>
```

But if we now click on the button, the table does not get refreshed. Of course, the easiest way to fix this is to give the table a `partialTriggers` attribute. But let's say we want to be super-efficient and want to prevent the table from being refreshed if no kid was deleted because of some error. In such a case, we have to enforce refreshing in the backing bean code. To do that, we have to add some lines to the action listener method in the `KidsTable.java` backing bean code:

```
public void deleteSelected(ActionEvent event) {
  Object oldRowKey = getTable().getRowKey();

  Iterator<Object> selectedKeys =
                           getTable().getSelectedRowKeys().iterator();

  Map<Integer, Kid> map = Util.getKidsMap();

  int deleted = 0;
  while(selectedKeys.hasNext()) {
    Object key = selectedKeys.next();
    getTable().setRowKey(key);
    Kid kid = (Kid) getTable().getRowData();
    map.remove(kid.getId());
    deleted++;
  }

  getTable().setSelectedRowKeys(null);
  getTable().setRowKey(oldRowKey);

  if(deleted > 0) {
    RequestContext rc = RequestContext.getCurrentInstance();
    rc.addPartialTarget(getTable());
  }
}
```

We add a counter (`deleted`) to see if any kids get deleted. Only if `deleted > 0` at the end of the method, we get the `RequestContext` instance and call the `addPartialTarget()` method. The argument is the `UIXTable` object that is returned by the `getTable()` accessor method. Note that we have to have access to the UI object via binding, otherwise we don't have an object to pass to the `addPartialTarget()` method. In this case, we already had access to the UI object by using the `binding="#{kidsTable.table}"` attribute of the table definition.

www.it-ebooks.info

Dynamically hiding or showing components

A common requirement of interactive pages is to dynamically show or hide certain sections of a page. This can be done easily with Trinidad's PPR framework, but some caution should be taken. Each JSF component has a `rendered` attribute that determines if the component will be rendered or not. We could dynamically show or hide a component by manipulating this rendered attribute by some expression.

Let's take the `Employees.xhtml` page as an example. We added a list of employees below the table, and we want to show or hide this list dynamically, depending on whether or not a checkbox is selected. To achieve this, we start by adding a `boolean` property `showAsList` to the backing bean of the page, `EmployeesList.java`:

```
public class EmployeesTable {
  private UIXTable table;
  private boolean showAsList = false;

  public boolean isShowAsList() {
    return showAsList;
  }

  public void setShowAsList(boolean showAsList) {
    this.showAsList = showAsList;
  }

  ...
}
```

Now we can edit the page as follows:

```
<tr:selectBooleanCheckbox id="checkBox" autoSubmit="true"
                          label="Show list"
                          value="#{empsTable.showAsList}"/>

<tr:panelGroupLayout partialTriggers="checkBox">
  <tr:panelHeader rendered="#{empsTable.showAsList}"
                  text="List of employees"
                  inlineStyle="margin-top: 20px;">
    <tr:panelList>
    <!-- generate the list of employees here... -->
    </tr:panelList>
  </tr:panelHeader>
</tr:panelGroupLayout>
```


www.it-ebooks.info

The `<tr:selectBooleanCheckbox>` component has an `id`, and has `autoSubmit` set to `true`. Note that we've surrounded the `<tr:panelHeader>` component by an extra `<tr:panelGroupLayout>` component that has its `partialTriggers` attribute set to `checkBox`, which is the `id` of the `<tr:selectBooleanCheckbox>` component. The `<tr:panelHeader>` component inside the `<tr:panelGroupLayout>` component has it's `rendered` attribute bound to the same property in the backing bean where the `<tr:selectBooleanCheckbox>` component stores its value.

It is important to understand that we couldn't leave out the `<tr:panelGroupLayout>` component and simply add `partialTriggers="checkBox"` to the `<tr:panelHeader>` component. This is because of the way things get processed on the server. If a component's `rendered` attribute is `false`, the component does not get added to the component tree that is built on the server every time that the page gets rendered. If the component is not in the component tree, it cannot be triggered by another component. That's why we always need a parent component to have its `partialTriggers` attribute set to the `id` of the triggering component.

By the way, there are much more elegant ways to achieve the same result. In the previous example, we could have surrounded the `<tr:panelHeader>` component by a `<tr:showDetail>` component. In this case, we wouldn't need to add a property to our backing bean and we wouldn't have to fiddle with `partialTriggers`. But there may be situations where the presented approach will be useful.

Polling

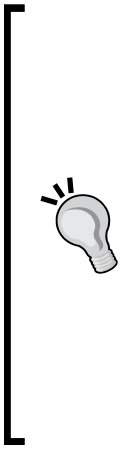
Sometimes we may want to refresh a part of a page on a timely basis—for example, because the data that is displayed can be changed by other users frequently. In this case, we can use the `<tr:poll>` component. The `<tr:poll>` component performs a partial submit at a given `interval`. We can refresh other elements by referring to the `<tr:poll>` element in the `partialTriggers` attribute. For a simple example, we could use this to display the current time on any page. Assume that we have a bean that returns the current time, as follows:

```
public Date getTime() {
  return new Date();
}
```

Now we can add the following fragment to our page to display the time:

```
<tr:poll interval="900" id="poll"/>
<tr:outputText value="#{bean.time}"
               partialTriggers="poll" >
  <tr:convertDateTime pattern="hh:mm:ss" />
</tr:outputText>
```


www.it-ebooks.info



www.it-ebooks.info

Apart from these ideas, it's good to realize that a lot of Trinidad components have some PPR usage embedded; we don't have to do anything to get PPR in our application if we use these components. The components with embedded PPR are:

- The `<tr:panelTabbed>` and `<tr:panelAccordion>` components show their children through PPR requests
- Expanding and collapsing content with `<tr:showDetail>` and `<tr:showDetailHeader>` components, and within `<tr:table>` and `<tr:treeTable>` components, as well as expanding and collapsing the `<tr:tree>` components will result in PPR requests instead of full submits
- Navigating through large data sets with the "paging" control of the `<tr:table>` and `<tr:treeTable>` components automatically uses PPR to refresh only the data in the table instead of the whole page if the next set of rows is retrieved
- When sorting is enabled in a `<tr:table>` or `<tr:treeTable>` component, the sorted rows are also retrieved from the server with PPR
- When selecting a date with a `<tr:chooseDate>` component, the calendar information is retrieved from the server via PPR
- The dialog framework (see the next section) also uses some built-in PPR functionality

Creating dialogs

Using dialogs in web applications is never a trivial thing to do. Trinidad doesn't offer us a solution that makes using dialogs trivial, but the Trinidad dialog framework does make the use of dialogs a lot easier. Before we dive into this dialog framework, let's briefly understand what the unique features of a dialog are. According to Wikipedia (`http://en.wikipedia.org/wiki/Dialog_box`), a dialog (box) is:

> *a special window, used in user interfaces to display information to the user, or to get a response if needed.*

From a software engineering perspective, we should also realize that the same dialog box can potentially be used from different screens in an application. This means that a dialog should not have to know anything about the page it is called from. Now let's see how the Trinidad dialog framework helps us to create dialogs that can return values to their calling pages without any knowledge of the calling page. We'll do this by building an example dialog. We'll build a "braveness calculator" that can calculate the braveness factor of a kid. We will do this by selecting one of the predefined reactions that comes closest to the reaction of the kid when the monster entered the kid's room to scare the kid. The age of the kid will also be taken into account.

www.it-ebooks.info

Building a dialog

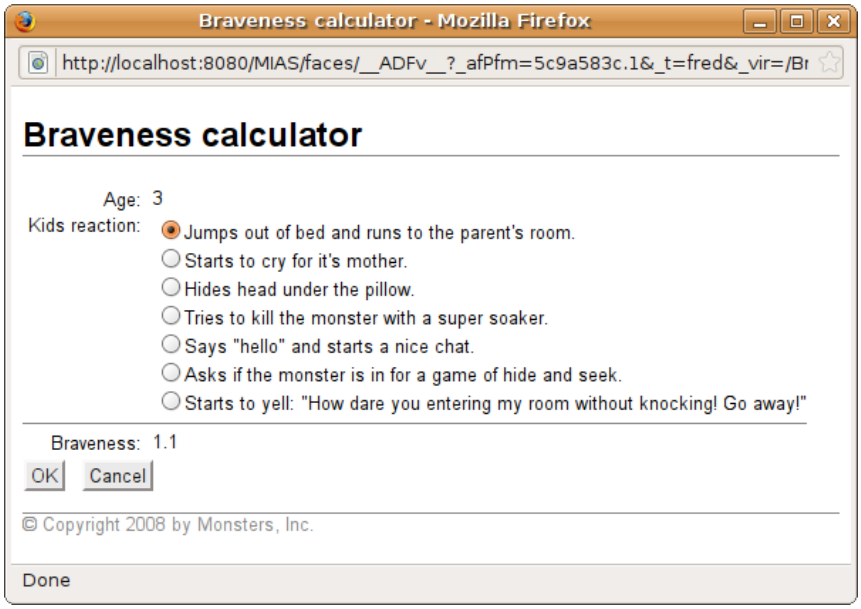
We can build the page that will render our dialog just like any other page we've built so far:

```
<ui:composition template="templates/dialog.xhtml">
  <ui:define name="title">#{msg.bravenessCalc}</ui:define>
  <ui:define name="content">
    <tr:panelFormLayout>
      <tr:group>
        <mias:field id="age" bean="#{bravenessCalc}"
                    readOnly="true" />
        <mias:selectField
                    id="selectedReaction"
                    bean="#{bravenessCalc}"
                    type="radio"
                    items="#{bravenessCalc.childReactions}"
                    itemValue="value"
                    itemLabel="reactionDescription"
                    partialSubmit="true" autoSubmit="true"/>
      </tr:group>
      <mias:field id="braveness" bean="#{bravenessCalc}"
                  partialTriggers="selectedReaction"/>
      <f:facet name="footer">
        <tr:panelButtonBar>
         <tr:commandButton value="#{msg.ok}"
                           action="#{bravenessCalc.done}"/>
         <tr:commandButton value="#{msg.cancel}"
                           immediate="true"
                           action="#{bravenessCalc.cancel}"/>
        </tr:panelButtonBar>
      </f:facet>
    </tr:panelFormLayout>
  </ui:define>
</ui:composition>
```

In the first line, we use a special dialog template instead of our default template. This dialog template is essentially not much more than the standard template without the header with the company logo. We use a backing bean called `bravenessCalc` for this page. Note that the first field is read only, and is filled by the `age` property from that backing bean. The second input control is a set of radio buttons, where the kid's reaction to the monster can be selected. The `autoSubmit` attribute is set to `true`, in order to force a submission whenever the selection is changed. The `partialSubmit` attribute is set to `true`, in order to prevent the `autoSubmit` attribute from submitting the entire page, but instead forcing a partial submission via AJAX requests. See the *Using AJAX and Partial Page Rendering* section for more details.

www.it-ebooks.info

The footer of the form contains two buttons: an **OK** button and a **Cancel** button. Both `<tr:commandButton>` components call a method from the backing bean via their `action` attribute. Note that apart from the different template, we don't have any dialog-specific code here.



Creating the backing bean for the dialog

In the backing bean, we do have dialog-specific stuff. Let's be complete this time and take a look at the backing bean class as a whole:

```
package inc.monsters.mias.backing;

import java.util.ArrayList;
import java.util.List;

import org.apache.myfaces.trinidad.context.RequestContext;

public class BravenessCalc {
private List<KidsReaction> childReactions;
private int selectedReaction;
private int age;
private double braveness;
public BravenessCalc() {
childReactions = new ArrayList<KidsReaction>();
// fill the list with KidsReaction objects
}
```


www.it-ebooks.info

```
  public List<KidsReaction> getChildReactions() {
    return childReactions;
  }

  public void setChildReactions(List<KidsReaction>
                                          childReactions) {
    this.childReactions = childReactions;
  }

  public int getSelectedReaction() {
    return selectedReaction;
  }

  public void setSelectedReaction(int selectedReaction) {
    this.selectedReaction = selectedReaction;
  }

  public int getAge() {
    RequestContext rc = RequestContext.getCurrentInstance();
    age = (Integer)rc.getPageFlowScope().get("kidsAge");
    return age;
  }

  public double getBraveness() {
    braveness = ( (12.0 - (double)getAge())
                  * (10.0/12.0)
                  * ((double)getSelectedReaction() + 1.0)
                  * (10.0 / (double)childReactions.size()))
                / 10.0;
    return braveness;
  }

  public String done() {
    RequestContext rc = RequestContext.getCurrentInstance();
    rc.returnFromDialog(getBraveness(), null);
    return null;
  }

  public String cancel() {
    RequestContext rc = RequestContext.getCurrentInstance();
    rc.returnFromDialog(null, null);
    return null;
  }
}
```


www.it-ebooks.info

The lines in code that are specific to dialogs are highlighted in the class declaration that we just saw. In the `getAge()` method, we use Trinidad's `RequestContext` class to get access to the page flow scope and get the value of the `kidsAge` property from that scope. This assumes that the calling page has put a value there before calling the dialog. The `done()` and `cancel()` methods are nearly identical. They both call the `returnFromDialog()` method on the `RequestContext`. This is needed to close the dialog. The difference is that the `done()` method calls the `getBraveness()` method to pass the calculated braveness to the `returnFromDialog()` method. This method then passes this value to the calling page. Now we have input via the page flow scope and output via the `returnFromDialog()` method. Both ensure that we don't have to know anything about the calling page.

Using an alternative way of returning values

In case we don't want to write special methods in our backing bean, we can alternatively use the `<tr:returnActionListener>` component to return values. This component will call the `returnFromDialog()` method for us with a value that we define in the `value` attribute. The `<tr:returnActionListener>` component has to be a child of a component that would otherwise have called an action, such as a `<tr:commandButton>` or `<tr:commandLink>` component. So we could remove the `done()` and `cancel()` methods from our bean and instead change the button definitions as follows:

```
<tr:commandButton text="#{msg.ok}">
  <tr:returnActionListener value="#{bravenessCalc.braveness}"/>
</tr:commandButton>
<tr:commandButton text="#{msg.cancel}" immediate="true" >
  <tr:returnActionListener />
</tr:commandButton>
```

Calling the dialog

Now that our dialog is finished, we want to call it from one of our pages. To be able to do so, we first have to create a navigation rule for our dialog. This is done in the `faces-config.xml` file:

```
<navigation-rule>
  <display-name>*</display-name>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>dialog:bravenessCalc</from-outcome>
    <to-view-id>/Braveness.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```


www.it-ebooks.info

This is just an ordinary navigation rule except from the `<from-outcome>` part. Note that we have prepended the name of our outcome with `dialog:`. This tells Trinidad to treat this navigation rule as navigation to a dialog. Without the `dialog:` prefix, the dialog wouldn't work as expected. Now let's see how we can call the dialog from our `EditKid.xhtml` page:

```
<tr:panelLabelAndMessage for="braveness"
                         labelAndAccessKey="#{msg.braveness}:">
  <tr:inputText id="braveness"
                value="#{editKidForm.braveness}"
                simple="true"
                binding="#{editKidForm.bravenessInput}" />
  <tr:commandButton text="#{msg.calculate}"
                    partialSubmit="true"
                    useWindow="true"
                    action="dialog:bravenessCalc"
                    id="calcButton">
    <tr:setActionListener to="#{pageFlowScope.kidsAge}"
                          from="#{editKidForm.age}"/>
  </tr:commandButton>
</tr:panelLabelAndMessage>
```

In comparison with the original page, we've added a button to the right of the braveness input field that will open the braveness calculator dialog. To help the `<tr:panelFormLayout>` component keep the labels and components lined up, we've put the `<tr:inputText>` and the `<tr:commandButton>` components inside a `<tr:panelLabelAndMessage>` component. Note that the `<tr:commandButton>` component has `dialog:bravenessCalc` as an action. This tells Trinidad to open our dialog whenever the button is clicked. But before the dialog opens, we have to put the input parameter into the `pageFlowScope`. This is done by the `<tr:setActionListener>` component.

Receiving the dialog's output

Once the dialog closes, we want to receive the output value(s) from the dialog, of course. We need some extra attributes for the `<tr:inputText>` and `<tr:commandButton>` components to achieve that. The extra attributes are highlighted in the following code snippet:

```
<tr:panelLabelAndMessage for="braveness"
                         labelAndAccessKey="#{msg.braveness}:">
  <tr:inputText id="braveness"
                value="#{editKidForm.braveness}"
                simple="true"
                binding="#{editKidForm.bravenessInput}"
                partialTriggers="calcButton"/>
  <tr:commandButton text="#{msg.calculate}"
```


www.it-ebooks.info

```
                        partialSubmit="true"
                        useWindow="true"
                        action="dialog:bravenessCalc"
                        returnListener="#{editKidForm.bravenessCalcReturn}"
                        id="calcButton">
      <tr:setActionListener to="#{pageFlowScope.kidsAge}"
                            from="#{editKidForm.age}"/>
    </tr:commandButton>
  </tr:panelLabelAndMessage>
```

The `returnListener` attribute will tell Trinidad which method to call in order to process the output values of the dialog. The `partialTriggers` attribute is needed to make sure that the `<tr:inputText>` component gets updated when we return from the dialog. To make the whole thing works, we have to implement the `bravenessCalcReturn` method in our `editKidForm` bean. That method could look like this:

```
public void bravenessCalcReturn(ReturnEvent event) {
  if (event.getReturnValue() != null) {
    getBravenessInput().setSubmittedValue(null);
    getBravenessInput().setValue(event.getReturnValue());
  }
}
```

The `getBravenessInput()` method is just a getter method that accesses the input component that is made accessible by using its `bindings` attribute to bind it to this bean. The `setSubmittedValue(null)` call will reset the component and, of course, the call to `setValue()` will set a new value in it. We used the `getReturnValue()` method of the `ReturnEvent` to get the return value of the dialog.

Using inputListOfValues as an easier alternative

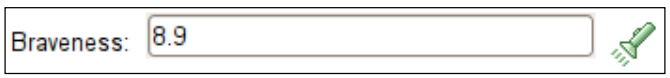
While it's not rocket science to create a dialog as described in the previous section, it isn't trivial either. Some custom methods in backing beans are needed, although we might not want to do other things besides just passing the values. Therefore, a simpler but less flexible solution is offered as an alternative: the `<tr:inputListOfValues>` component. This component replaces the separate input field and button, and handles the receipt of output values from the dialog. It doesn't change the definition of the dialog itself. We could rewrite the part of the `EditKid.xhtml` page where we call the dialog, as follows:

```
<tr:inputListOfValues id="braveness"
                      value="#{editKidForm.braveness}"
                      action="dialog:bravenessCalc"
                      labelAndAccessKey="#{msg.braveness}:">
```


www.it-ebooks.info

```
    <tr:setActionListener to="#{pageFlowScope.kidsAge}"
                          from="#{editKidForm.age}"/>
</tr:inputListOfValues>
```

Note that we still use the `dialog:bravenessCalc` navigation case to navigate to the dialog. So, there won't be any changes in the `faces-config.xml` file, compared to the earlier situation. We also need the `<tr:setActionListener>` component to put the input value(s) for the dialog in the `pageFlowScope`. But we don't need a return listener to receive the output value from the dialog. The `<tr:inputListOfValues>` component puts the output value of the dialog into its integrated input field automatically, so we can remove the `bravenessCalcReturn()` method from our bean. The `<tr:inputListOfValues>` component does not use a button to launch the dialog, but uses a clickable icon instead. By default, a flashlight icon is used, as displayed in the next image. This icon can be overridden by pointing the `icon` attribute to a custom image.



Using lightweight dialogs

By default, all Trinidad dialogs are created as new browser windows with the standard browser menu and button bar hidden. This way of creating dialogs has some drawbacks. Most browsers have a built-in pop-up blocker that will block this kind of dialogs. And creating a new browser window is also a relatively heavy operation, so it might take a while before the dialog appears.

To overcome these drawbacks, Trinidad has the option to use "lightweight dialogs" instead. By setting an application-wide configuration property, all dialogs will be created inside the current page by using some JavaScript. The only thing we have to do to enable this feature is set the value of the `org.apache.myfaces.trinidad.ENABLE_LIGHTWEIGHT_DIALOGS` context variable to `true` in the `web.xml` file. The *Tuning Trinidad* section in the next chapter has detailed instructions on how to set the context variables.

Client-side validation and conversion

Apart from refreshing only a portion of a page with Partial Page Rendering, another important property of "AJAX" or "Web 2.0" applications is that the validation of input takes place at the client—in other words, in the browser. From a user's perspective, the advantage is that this type of validation is often faster. Another advantage for the user is that the validation can be triggered per field, so he or she gets an error message immediately if invalid data is entered.

www.it-ebooks.info

From a developer's perspective, an extra bonus of client-side validation is that it can save a lot of traffic to the server because a form will only be submitted when it is valid. On the other hand, a downside of client-side validation is that validation code gets spread over the project even more. And as a browser cannot run Java code locally, client-side validation code has to be written in JavaScript, the web browser's language.

The good news is that we get a lot of client-side validation for free with Trinidad. Whenever we use one of Trinidad's `<tr:validateXXX>` components, we will get client-side validation for free. This also goes for Trinidad's converters, the `<tr:convertXXX>` components. These converters will also be executed at the client side. The set of validators is pretty versatile. In particular, the `<tr:validateRegExp>` component is very flexible.

Nevertheless, there may always be a situation where the standard validators can't do the job. In this case, we can use the standard JSF API to create our own custom validator. Trinidad offers us an elegant way of extending that custom validator with client-side capabilities. Unfortunately, Trinidad cannot magically write JavaScript code for us, so we'll still need to write some JavaScript. Although we focus on the client-side stuff in this section, we will work through a complete example, as creating custom validators is not an everyday task—not even for experienced JSF developers.

A good reason why standard validators may be inadequate can be a validation that has to be done on a custom data structure. In this case, the input value (always a `String`) must be converted first, so we also need a custom converter. In the rest of this section we're going to build a data structure, a converter, and a validator. The data structure in our example is a list of foods that can be used to store a list of favorite foods for each kid in the system. We want the user to be able to enter the list in a simple input box, where the items are separated by some separator character.

Defining the data structure

Let's define a simple data structure for our food list. We create a new class called `FoodList` in the `inc.monsters.mias.data` package. It's not much more than a thin wrapper around a `java.util.List`:

```
package inc.monsters.mias.data;

import java.util.ArrayList;
import java.util.List;

public class FoodList {
  List<String> list;

  public FoodList() {
```


www.it-ebooks.info

```
    this.list = new ArrayList<String>();
  }

  public void add(String food) {
    list.add(food);
  }

  public List<String> getAsList() {
    return list;
  }

  public String toString() {
    return toStringWithSeparator(" ");
  }

  public String toStringWithSeparator(String separator) {
    StringBuilder sb = new StringBuilder();
    for(String food : list) {
      sb.append(separator);
      sb.append(food);
    }

    if (sb.length() > 0) {
      // Use substring to remove leading separator
      return sb.toString().substring(separator.length());
    } else {
      return "";
    }
  }
}
```

And, of course, we extend our `Kid` class with getters and setters for the favorite food list:

```
  public FoodList getFavouriteFood() {
    return favouriteFood;
  }

  public void setFavouriteFood(FoodList favouriteFood) {
    this.favouriteFood = favouriteFood;
  }
```


www.it-ebooks.info

Creating the converter

Creating a converter for this data type is pretty straightforward. As the separator character for the list of food is not fixed, we have to make it configurable. For now, we start with a private property and getter and setter methods. Later on, we'll see how we can pass values from our converter tag to this property. Of course, we have to implement the `javax.faces.convert.Converter` interface and the two methods that are in that interface. This is how our converter class would look without client-conversion capabilities:

```
package inc.monsters.mias.conversion;

import inc.monsters.mias.data.FoodList;

import java.util.Collection;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;

public class FoodListConverter implements Converter {
  private String separator = " ";

  public Object getAsObject(FacesContext context,
                            UIComponent component,
                            String string) {

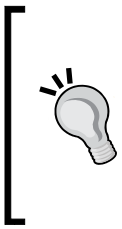
    String[] list = string.split(getSeparator());
    FoodList foodList = new FoodList();
    for(String food : list) {
      foodList.add(food);
    }
    return foodList;
  }

  public String getAsString(FacesContext context,
                            UIComponent component,
                            Object foodList) {

    if(foodList != null) {
      return ((FoodList)foodList)
                      .toStringWithSeparator(getSeparator());
    } else {
      return "";
```


www.it-ebooks.info

```
        }
    }
```



www.it-ebooks.info

Let's see how this changes our previously-created `FoodListConverter` class:

```
package inc.monsters.mias.conversion;

import inc.monsters.mias.data.FoodList;

import java.util.Collection;
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;

import org.apache.myfaces.trinidad.convert.ClientConverter;

public class FoodListConverter implements Converter, ClientConverter {

// Left out the unchanged server side conversion code...

public String getClientLibrarySource(FacesContext context){
ExternalContext ec = context.getExternalContext();
return ec.getRequestContextPath() +
"/scripts/FoodListConverter.js";
}

public String getClientScript(FacesContext context,
UIComponent arg1component) {
return null;
}

public String getClientConversion(FacesContext context,
UIComponent component) {
return "new FoodListConverter('"+getSeparator()+"');";
}

public Collection<String> getClientImportNames() {
return null;
} }
```

The `getClientLibrarySource()` method uses the `ExternalContext` to get the request context path, and appends the location of the JavaScript file that contains the JavaScript implementation of the converter. (We're going to write that one later on.) The `getClientScript()` method just returns `null` because we do not use inline JavaScript here. The most interesting method here is the `getClientConversion()` method. This returns a string that will be evaluated as JavaScript to instantiate a new `FoodListConverter` object on the client side.

www.it-ebooks.info

We should remember that we have to implement a constructor that accepts a separator in the JavaScript implementation. Note that the separator is put between single quotes. (In JavaScript, strings may be either between single or double quotes. This comes in handy now; otherwise we would have had to escape the double-quote characters, making the line even more unreadable than it already is.) The `getClientImportNames()` method also returns `null` because we don't use any of the built-in Trinidad conversion scripts.

Implementing the client-side code

Now comes the tricky part—implementing the client-side converter in JavaScript. Trinidad gives us some JavaScript APIs that are very similar to the JSF server-side conversion APIs. Although JavaScript doesn't have something that is similar to interfaces in Java, Trinidad has defined a virtual interface that we have to implement for the client-side conversion to work. Let's see how we can implement this "interface":

```
function FoodListConverter(separator) {
this.separator = separator; }

FoodListConverter.prototype = new TrConverter();

FoodListConverter.prototype.getAsString = function (value, label) {
return value.toString(); }

FoodListConverter.prototype.getAsObject = function (value, label) {
return new FoodList(value, this.separator); }
```

The first three lines are the JavaScript equivalent of a constructor. Note that this constructor takes a single argument, which is the separator. This is stored in a member variable called `this.separator`. The line `FoodListConverter.prototype = new TrConverter();` lets the `FoodListConverter` class inherit from the `TrConverter` class. The `prototype` keyword is the JavaScript way to access a class definition. The JavaScript functions `getAsString()` and `getAsObject()` are the JavaScript equivalents of the methods with the same name in the Java implementation. Note that a `FoodList` object is used to do the real conversion work. Let's see how we can implement such a `FoodList` class in JavaScript:

```
function FoodList(value, separator) {
this.separator = separator;
this.foodlist = value.split(separator); }

FoodList.prototype.toString = function() {
```


www.it-ebooks.info

```
    var result = ""
    for (food in this.foodlist) {
      result += this.separator + food;
    }
    return result.substring(this.separator.length);
  }
```

Again, we start with a constructor. In this case, the constructor takes two arguments: a `value` that should be an input string that can be split into separate food names, and a `separator`. The separator is stored "as is" in the object. The value is split into an array of strings that is stored in the `foodlist` variable in the object. Apart from the constructor, we also have a `toString()` function that concatenates the elements of the `foodlist` array to a single string, where the elements are separated by `this.separator`. As we can only return a single JavaScript file in the `getClientLibrarySource()` method, the easiest way to make sure that the client-side `FoodListConverter` has access to the `FoodList` class is to put both in the same `FoodListConverter.js` file. This is possible because JavaScript doesn't put any restrictions on how many classes can be in a file.

Creating the validator

For the validator, we also start with a basic server-side validator class that implements the `javax.faces.validator.Validator` interface. We've decided that a valid food name may only contain letters. Let's see how we can implement this:

```
package inc.monsters.mias.validators;

import java.util.Collection;

import inc.monsters.mias.data.FoodList;

import javax.faces.application.FacesMessage; im-
port javax.faces.component.UIComponent; import
javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class FoodListValidator implements Validator {

public void validate(FacesContext context,
UIComponent component,
Object value)
throws ValidatorException {

int errors = 0;
StringBuilder wrong = new StringBuilder();
for (String food : ((FoodList) value).getAsList()) {
```


www.it-ebooks.info

```
        if (!food.matches("[A-Za-z]*")) {
          if (errors > 0) {
            wrong.append(" ");
          }
          wrong.append(food);
          errors++;
        }
      }

      if (errors > 0) {
        FacesMessage msg = new FacesMessage();
        msg.setSeverity(FacesMessage.SEVERITY_ERROR);
        msg.setSummary("Validation Error");
        msg.setDetail(wrong.toString() +
                      (errors == 1
                            ? " is not a valid food name"
                            : " are not valid food names"));
        throw new ValidatorException(msg);
      }
    }
  }
```

Most of the code is only used for generating a nice error message. The two highlighted lines are essential for the validation itself. The first defines a loop over all of the items in the incoming `FoodList` object. The second highlighted line validates a single item in the food list against a regular expression that allows only letters.

Enabling client-side capabilities

To enable client-side validation for our validator, we have to start by implementing the `org.apache.myfaces.trinidad.validator.ClientValidator` interface, which is very similar to the `ClientConverter` interface. Our `FoodListValidator.java` class will change as follows:

```
package inc.monsters.mias.validators;

import java.util.Collection;

import inc.monsters.mias.data.FoodList;
import javax.faces.application.FacesMessage; im-
port javax.faces.component.UIComponent; import
javax.faces.context.ExternalContext; import
javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import org.apache.myfaces.trinidad.validator.ClientValidator;

public class FoodListValidator implements Validator, ClientValidator {
```


www.it-ebooks.info

```
  // Unchanged validate() method is left out here...

  public Collection<String> getClientImportNames() {
    return null;
  }

  public String getClientLibrarySource(FacesContext context){
    ExternalContext ec = context.getExternalContext();
    return ec.getRequestContextPath() +
                            "/scripts/FoodListValidator.js";
  }

  public String getClientScript(FacesContext context,
                                UIComponent component) {
    return null;
  }

  public String getClientValidation(FacesContext context,
                                    UIComponent component) {
    return "new FoodListValidator();";
  }
}
```

As you can see, the four methods that are prescribed by the `ClientValidator` interface are nearly identical to the ones we implemented in our converter.

Implementing the client-side code

As with the converter, implementing the client-side code is the trickiest part, especially when you don't have much JavaScript experience, which is true for a lot of Java EE developers. Again, Trinidad offers us a JavaScript API that is very similar to the server-side API for validation in standard JSF. Let's see how we can build a client-side validator using this API:

```
function FoodListValidator() {

}

FoodListValidator.prototype = new TrValidator();
FoodListValidator.prototype.validate =
                        function(value, label, converter) {
  if(value != null) {
    var expr = /^[A-Za-z]*$/
    for(var i = 0; i < value.foodlist.length; i++) {
      if(! expr.test(value.foodlist[i])) {
        var msg = new TrFacesMessage(
```


www.it-ebooks.info

```
                "Validation Error",
                value.foodlist[i] + " is not a valid food name.",
                TrFacesMessage.SEVERITY_ERROR);
            throw new TrValidatorException(msg);
          }
        }
      }
    }
```

As with the converter, we start with a constructor for the `FoodListValidator` object. We don't have any arguments, so the constructor is empty. We cannot remove the empty constructor, though, as JavaScript doesn't create a default constructor for us. Then we have the `FoodListValidator.prototype = new TrValidator();` line, which makes our class inherit from the `TrValidator` class. The most important part here is the `validate()` function. Note that the same regular expression is used for validation, except that in JavaScript, regular expressions are surrounded by the `/^` and `$/` characters instead of quotes. A `TrFacesMessage` object is used for the error message in case an invalid food name is detected. This is similar to the `FacesMessage` object on the server side. The same goes for the `TrValidatorException` that is thrown, similarly to `ValidatorException` in Java.

Wiring everything together

Now that we have implemented our data structure, converter, and validator on both the server side and the client side, we have to do some "wiring" to make it work. This subsection outlines the steps we have to take.

Declaring the converter and validator in faces-config.xml

First, we have to declare our converter and validator in the `faces-config.xml` file of our project. This is pretty straightforward. Just add the following to the file:

```
<converter>
  <converter-id>foodListConverter</converter-id>
  <converter-class>
              inc.monsters.mias.conversion.FoodListConverter
  </converter-class>
</converter>
<validator>
  <validator-id>foodListValidator</validator-id>
  <validator-class>
              inc.monsters.mias.validators.FoodListValidator
```


www.it-ebooks.info

```
    </validator-class>
  </validator>
```

Creating custom tags

Once we've declared our converter and validator in the `faces-config.xml` file, we can use them with the general `<f:validator>` and `<f:converter>` tags. Our validator, for example, could be used as follows:

```
<f:validator validatorId="foodListValidator" />
```

However, we cannot pass attributes this way, which is needed for our converter. To be able to pass parameters to our converter, we have to define a custom tag using Facelets. This can be done by adding the following lines to our `mias.taglib.xml` file, where a lot of other custom tags are already defined:

```
<tag>
  <tag-name>convertFoodList</tag-name>
  <converter>
    <converter-id>foodListConverter</converter-id>
  </converter>
</tag>
<tag>
  <tag-name>validateFoodList</tag-name>
  <validator>
    <validator-id>foodListValidator</validator-id>
  </validator>
</tag>
```

We've added our validator too, just for the convenience of having our own tag. Note that we didn't define which arguments our tag should have. Facelets automatically calls a setter method in the converter class if it finds an attribute on the custom tag. Although this may sound easy, it has the disadvantage that an incorrectly-spelled attribute name in a page won't generate an error, and thus can easily be overlooked. So if you ever notice a custom validator that seems not to work, start by checking if the attributes used in the tag correspond exactly to the properties in the class that implements the validator.

Using the converter and validator in a page

Now let's see how we can use our custom converter and validator in a page. Let's add a "favorite food" field to our `EditKid.xhtml` page, and add the converter and validator to it:

```
<mias:field id="favouriteFood"
            bean="#{editKidForm.selectedKid}"
            partialTriggers="btnApply">
  <mias:convertFoodList separator=" " />
```


www.it-ebooks.info

```
    <mias:validateFoodList />
  </mias:field>
```

Note that the `partialTriggers` attribute of the input field is set to the `id` of the apply button. We have to make sure that the apply button has its `partialSubmit` property set to `true` in order to enable client-side validation. In this case, Trinidad will validate on the client side before a partial submit to the server is executed. If validation fails, no data is submitted to the server.

Internationalization of messages

Until now, we have added hardcoded error messages to our JavaScript files. Of course, we want our error messages to be taken from our message bundle, as for all other user interface elements, which will enable easy internationalization of our application. To achieve this, we can use the constructor of our client-side validation object to pass an error message that we look up from the message bundle. We have to change our JavaScript code a bit to make use of these messages. That's a nice opportunity to also have a look at an extra JavaScript API that Trinidad offers us—`TrFastMessageFormatUtils`.

Changing getClientValidator()

First things first. Let's start by changing the `getClientValidation()` method in our `FoodListValidator.java` class that returns the JavaScript code that is needed to instantiate our JavaScript `FoodListValidator` object:

```
public String getClientValidation(FacesContext context,
                                  UIComponent component) {
  String bndl = context.getApplication().getMessageBundle();
  Locale locale = context.getViewRoot().getLocale();
  ResourceBundle msg = ResourceBundle.getBundle(bndl, locale);
  return "new FoodListValidator('"
            + msg.getString("NoValidFoodErrorTitle")
            + "', '"
            + msg.getString("NoValidFoodError")
            + "');";
}
```

The `getMessageBundle()` method returns the base name of the message bundle that is configured for the application. The `getLocale()` method of the `ViewRoot` object returns the current locale. With both the base name and the current locale, we can get a `ResourceBundle` object where we can look up the error messages. In the `return` statement, we compose a string that calls a `FoodListValidator` constructor with an error title and an error message. Note that we have to define values for `NoValidFoodErrorTitle` and `NoValidFoodError` in our message bundle.

www.it-ebooks.info

Changing the JavaScript constructor

Of course, we also have to change the JavaScript constructor to make sure that the error title and error message get stored inside the `FoodListValidator` object. That's easy:

```
function FoodListValidator(errorTitle, errorMessage) {
  this.errorTitle = errorTitle;
  this.errorMessage = errorMessage;
}
```

Formatting the error message

At the client side, we don't have the luxury of the Java API with its numerous text manipulation methods. Luckily, Trinidad has the `TrFastMessageFormatUtils` API, which at least gives us some formatting options. There's a `TrFastMessageFormatUtils.format()` method that takes a format string and an unlimited list of arguments. The format string can contain only indexed placeholders, marked by braces. For example:

```
{0} is not valid food
```

The indexes start at `0`. Assume that we add to our message bundle:

```
NoValidFoodError={0} is not valid food
```

We can adapt our JavaScript `validate()` function to format the error message as follows:

```
FoodListValidator.prototype.validate =
                          function(value, label, converter) {
  if(value != null) {
    var expr = /^[A-Za-z]*$/
    for(var i = 0; i < value.foodlist.length; i++) {
      if(! expr.test(value.foodlist[i])) {
        var msg = TrFastMessageFormatUtils
                          .format(this.errorMessage,
                                  value.foodlist[i]);
        var fmsg = new TrFacesMessage(
                        this.errorTitle,
                        msg,
                        TrFacesMessage.SEVERITY_ERROR);

        throw new TrValidatorException(fmsg);
      }
    }
  }
}
```


www.it-ebooks.info

First, we format the error message by making a call to `TrFastMessageFormatUtils.format()`. The first parameter is the format string (`this.errorMessage`), and the second parameter is the value that will be substituted at the position of the placeholder `{0}`. Then we call the `TrFacesMessage` constructor as we did before, except that we now use our formatted error message and the title that was set by the constructor.

Using Trinidad's JavaScript API

While implementing our client-side converter and validator, we used some methods from the Trinidad JavaScript API. This API has more methods than the ones we used so far. For example, there are methods to easily submit a form via an AJAX request or to create custom AJAX requests from scratch. The main benefit of using Trinidad's API functions is that we don't have to care about the differences between various browsers, as this is taken care of by the JavaScript library of Trinidad.

The *AJAX and Partial Page Rendering (PPR)* section in the *Developer Guide* on the Trinidad website (`http://myfaces.apache.org/trinidad/devguide/ppr.html`) has some examples of the usage of the Trinidad JavaScript API. But apart from these examples, no further documentation on this API is available. This limits the usability of the API drastically. We should also ask ourselves if it is desirable to use custom JavaScript in our application. Isn't it one of the goals of JSF to eliminate the need to write this sort of code ourselves? But if we end up writing custom JavaScript anyway, we now know there is an API that we could use.

Writing, testing, and debugging JavaScript

As writing JavaScript code is not a common task for many Java EE developers, let's have a quick look at some tooling for testing and debugging JavaScript code. Once you get started with JavaScript, you'll soon find out why these are very useful.

Writing JavaScript code

Most programmer-oriented text editors have code highlighting and code completion for JavaScript. Most IDEs should also have these options too. However, we should realize that most of the time those editors are not as sophisticated as the Java editor of a modern Java IDE. One of the difficulties with JavaScript is that it is loosely typed. This highly limits the possibilities for auto completion and error detection. The fact that JavaScript code doesn't get compiled removes an extra opportunity to detect errors.

www.it-ebooks.info

Debugging

As JavaScript code mostly runs within a browser, it makes perfect sense to integrate a JavaScript debugger in a browser. This is exactly what the Firebug extension does for Firefox—it lets you look at the source code of any page that loads in Firefox. It is also possible to see referenced files such as CSS stylesheets and JavaScript source files. Breakpoints can be set to see if a certain line of JavaScript code is executed, and from a breakpoint one can execute the code step-by-step just as with a Java debugger. Firebug can be downloaded from the website `http://getfirebug.com/`.

It may also be necessary to debug the interaction of custom JavaScript code with Trinidad's built-in JavaScript. So when debugging client-side code, it is a good idea to put Trinidad in the debug mode to prevent JavaScript code from being obfuscated. The *Debugging* subsection of the *Tuning Trinidad* section in Chapter 7 discusses the various configuration options for debugging with Trinidad.

Logging

Logging can be very useful when testing and debugging. There's no standard way of logging in JavaScript (yet), but the same Firebug plugin for Firefox offers an API that can be used to send log messages to the JavaScript console of the Firebug plugin. While it goes beyond the scope of this book to cover all of the features of Firebug, let's use a small example to see some nice possibilities of this plugin. With a call to `console.log()`, we can write any message to the Firebug console from our JavaScript code. We could extend our `validate()` JavaScript function with some logging:

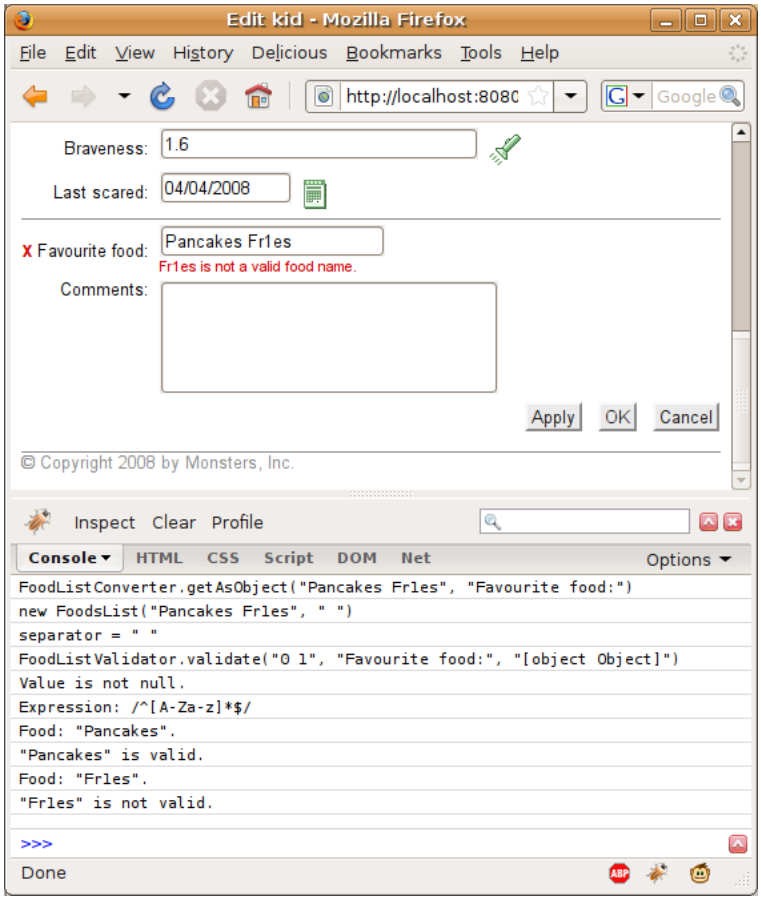
```
FoodListValidator.prototype.validate =
                    function(value, label, converter) {
  console.log('FoodListValidator.validate("'
              + value + '", "' + label + '", "'
              + converter + '")');
  if(value != null) {
    console.log('Value is not null.');
    var expr = /^[A-Za-z]*$/
    for(var i = 0; i < value.foodlist.length; i++) {
      console.log('Food: "' + value.foodlist[i] + '".');
      if(! expr.test(value.foodlist[i])) {
      console.log('"' + value.foodlist[i]
                  + '" is not valid.');
      var msg = new TrFacesMessage(
                  "Validation Error",
                  value.foodlist[i]
                    + " is not a valid food name.",
                  TrFacesMessage.SEVERITY_ERROR);

      throw new TrValidatorException(msg);
    }
```


www.it-ebooks.info

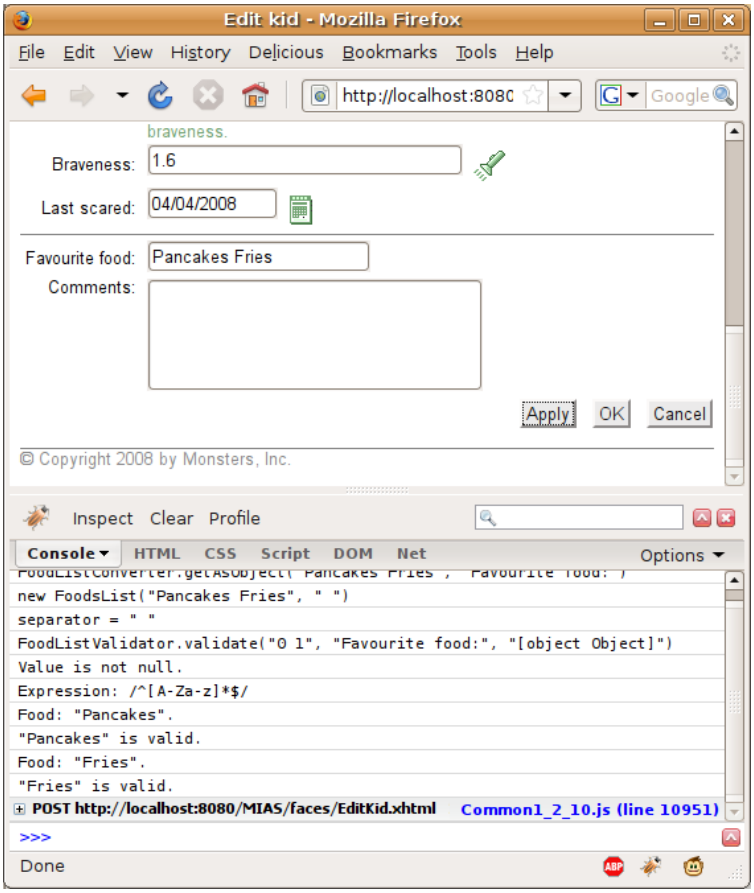
```
            console.log('"' + value.foodlist[i]
                            + '" is valid.');
        }
    }
```

The log lines are highlighted, and the other lines are left unchanged. In the following two images, the first shows the log entries created when an invalid food name is entered, and the second shows the log entries created when valid data is entered.



www.it-ebooks.info

We can see that in the second case, the last line in the console shows an HTTP POST request, meaning that the form data is being submitted to the server. In the first case, there is no HTTP POST because the data was invalid, and so no data is submitted to the server.



More information about Firebug's logging possibilities can be found on the **Firebug and Logging** page on Firebug's website—`http://getfirebug.com/logging.html`.

www.it-ebooks.info

Summary

This chapter covered the more advanced features of Trinidad. We saw that apart from the advanced components, Trinidad also has data visualization capabilities, as well as AJAX functionality. We also saw how we can simplify data flow with Trinidad's page flow scope. This chapter also introduced the dialog framework, and client-side validation and conversion.

In the next chapter, we'll explore Trinidad's skinning capabilities, and take a look at the various ways of tuning Trinidad for optimal performance, accessibility, and appearance.

www.it-ebooks.info

www.it-ebooks.info

7
Trinidad Skinning and Tuning

Trinidad is a highly-configurable framework. The look and feel of Trinidad's components can be changed in every little detail by using Trinidad's skinning capabilities. The behavior of the Trinidad framework as a whole can be changed by using the tuning parameters in configuration files. This chapter gives an introduction to Trinidad skinning, and an overview of the most important tuning options.

After reading this chapter, you will:

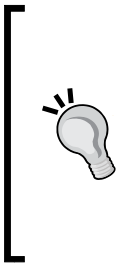
- Understand the basics of Trinidad skinning
- Be able to create a custom skin with the help of a web designer
- Know where to set the tuning parameters of Trinidad
- Have an overview of the most important tuning parameters of Trinidad

Skinning

Trinidad has the possibility for **skinning**. This means that you can customize the look and feel of every Trinidad component to fit your taste or company style. Skinning is done by means of **Cascading Style Sheets** (**CSS**). So, depending on how far you want to diverge from the default skin, a level of experience with CSS is needed. Given the large number of components that Trinidad has, you can imagine that skinning every little detail of all available components can be an awful lot of work.

The good news is that the Trinidad skinning mechanism makes good use of the cascading part of CSS, and even takes it a level higher. For example, if we only want to change some colors, we can do just that and inherit all of the defaults from the default skin. In this section, we will focus on the Trinidad-specific stuff. Diving into CSS and explaining how to use it is beyond the scope of this book. Should we want to create an advanced skin, probably the best thing to do is to ask a professional web designer to help us with the CSS. The professional designer probably should read this section as well, because Trinidad has its own special CSS extensions that get processed at the server side to generate a standard CSS file for the client.

www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

Letting the user choose the skin

Sometimes you may want to let the user choose which skin he wants to use. This can be useful if you have a skin with extra accessibility features for specific users, or if you create a hosted application that is used by different clients, and each client has its own skin. It is not that hard to make the skin configurable. We can make use of the fact that it is possible to use JSF expression language within the `trinidadconfig.xml` file. Of course, we need a bean to hold the user's preference. Let's create a `Preferences.java` file:

```
package inc.monsters.mias;

import java.util.List;

public class Preferences {
private String skinFamily;
private List<String> availableSkinFamilies;

public String getSkinFamily() {
return skinFamily;
}
public void setSkinFamily(String skinFamily) {
this.skinFamily = skinFamily;
}

public List<String> getAvailableSkinFamilies() {
return availableSkinFamilies;
}

public void setAvailableSkinFamilies(
List<String> availableSkinFamilies) {
this.availableSkinFamilies = availableSkinFamilies;
} }
```

That's pretty straightforward—just a bean with two properties and the accessors for those properties. We can add this class to the session scope as a managed bean, so that every user has his or her own preferences bean, including the initialization of the bean. This means that we have to add the following to our `faces-config.xml` file:

```
<managed-bean>
<managed-bean-name>preferences</managed-bean-name>
<managed-bean-class>inc.monsters.mias.Preferences
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
<managed-property>
<property-name>skinFamily</property-name>
```


www.it-ebooks.info

```
      <property-class>java.lang.String</property-class>
      <value>mias</value>
    </managed-property>
    <managed-property>
      <property-name>availableSkinFamilies</property-name>
      <property-class>java.util.List</property-class>
      <list-entries>
        <value-class>java.lang.String</value-class>
        <value>minimal</value>
        <value>mias</value>
      </list-entries>
    </managed-property>
  </managed-bean>
```

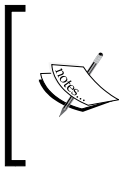
Note that we add two values to the list of available skins:

- `mias` is the name of the skin family that we are going to create
- `minimal` is the name of the default skin that is part of the Trinidad package

Now, we have to create a page that lets the user change the value of the `skinFamily` property in the `preferences` bean. Let's call that page `Preferences.xhtml`:

```
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Preferences</ui:define>
  <ui:define name="content">
    <tr:panelFormLayout>
      <mias:selectField type="radio"
                        items="#{preferences.availableSkinFamilies}"
                        bean="#{preferences}" id="skinFamily" />
      <f:facet name="footer">
        <tr:panelButtonBar>
          <tr:commandButton text="#{msg.apply}"
                            action="apply" />
          <tr:commandButton text="#{msg.ok}"
                            action="ok"/>
          <tr:commandButton text="#{msg.cancel}"
                            action="cancel"
                            immediate="true" />
        </tr:panelButtonBar>
      </f:facet>
    </tr:panelFormLayout>
  </ui:define>
</ui:composition>
```


www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

It's not relevant if `disabled` is a valid pseudo class in CSS. Trinidad will take care of rendering a valid CSS. Of course, the Trinidad component must have a disabled state; otherwise this would not make sense.

Using component piece selectors

Most Trinidad components consist of multiple **pieces** that can be styled separately. To do so, we have to identify each piece in the skinning file. There's a special syntax for this. Pieces of components are identified by the component name and the name of the piece, separated by two colons (`::`). So the syntax is much like the syntax for styling component states, except that we now use a double colon instead of a single colon. For example, if we want to style the tab piece of a `<tr:panelTabbed>` component, we could write:

```
af|panelTabbed::tab {
    background-color: gray;
    border-top: 1pt solid navy;
    border-bottom: 1pt solid navy;
}
```

It's also possible to combine the state and piece syntax. For example, we could write:

```
af|inputText:readOnly:disabled::content {
    color: gray;
}
```

This will give the `content` piece of the `<af:inputText>` component a gray color if it is in the `disabled` or `readOnly` state.

A list of all Trinidad components, and their pieces and states can be found on the Trinidad website at `http://myfaces.apache.org/trinidad/skin-selectors.html#ComponentLevelSelectors`.

Setting global styles using alias selectors

Given the large number of components, it is very undesirable to set the same colors and font names for every component that you want to style. In that case, should you ever want to change the overall color or font, you'd have to change it all over the CSS file. In normal CSS, you can use the fact that there's a fixed hierarchy. For example, it is a common practice to set the base colors and fonts in the style for the `<body>` elements, and derive the other elements' styles from that one. Trinidad skinning uses another approach.

www.it-ebooks.info

Trinidad skinning uses **alias selectors** to define global styles that can be referred to by the specific component styles. An alias selector always starts with a period (.) and ends with `:alias`. Most alias selectors have a rather narrow definition of what they style. Most of them only set a color or a font. In this way, combinations of several alias selectors can be made to achieve the desired style for a component. For example, to set the default font family for the entire skin, we could write:

```
.AFDefaultFontFamily:alias {
  font-family: "Times New Roman", times, serif;
}
```

This will only set the font family. If we want to change the font size too, we should add another alias selector—`.AFDefaultFont:alias`. This one is meant to set the font family, the default size, and the default weight. However, we don't want to repeat the font family, as we have already defined it. So we need a way to refer to an already defined style. For that reason the `-tr-rule-ref:selector()` syntax exists. We can use it as shown here:

```
.AFDefaultFont:alias {
  -tr-rule-ref:selector(".AFDefaultFontFamily:alias");
  font-size: 12pt;
  font-weight: normal;
}
```

This will set the font size to 12 points and the default weight to normal. The font family is the same as the one that is defined in the `.AFDefaultFontFamily:alias` selector.

An overview of all available alias selectors is available on the Trinidad website at `http://myfaces.apache.org/trinidad/skin-selectors.html#GlobalSelectors`. You should be aware that a lot of aliases will have calculated values if we don't set them explicitly. For example, the background colors for the `.AFVeryDarkBackground:alias`, `.AFMediumBackground:alias`, and `.AFLightBackground:alias` selectors will be calculated based on the background color value of the `.AFDarkBackground:alias` selector.

Skinning icons

Some components use icons. These can be skinned too, which is useful for making a consistent skin with matching colors. Icons are set with the `content` keyword, as shown here:

```
af|inputDate::launch-icon {
    content:url(/skins/suede/images/dp.gif);
    width: 19; height: 24;
}
```


www.it-ebooks.info

Note that the URL of the image starts with a slash (/). This means that the URL is relative to the context root of the application. There are four ways of specifying a URL:

- **Relative to the CSS file**: A URL should not start with a slash. In this example, it would be `images/dp.gif`, as the images directory is in the same directory as the skin's CSS file. To go up a level in the directory structure, `../` can be used.
- **Relative to the context root**: A URL should start with a single slash, as in the previous example.
- **Relative to the server**: This can be used to link to an image in another application on the same server. Of course, we need to be sure that the other application will always be available on the same server. Server-relative URLs start with a double slash, such as `//OtherContextRoot/images/dp.gif`.
- **Absolute**: Absolute URLs are not very flexible and should be used with caution. They start with `http://`. In the previous example, the corresponding absolute URL could be something like `http://mias.monsters.inc:8080/MIAS/skins/suede/images/dp.gif`. This example shows the main problem with using absolute URLs in this context—if an absolute URL is used in a skin CSS file, the CSS has to be adapted whenever the application gets deployed to another application server or even if the port of the server changes.

In some cases, you might want to use a text indicator instead of an image. In this case, the `url()` part can be replaced by the desired text in quotes, like the following example:

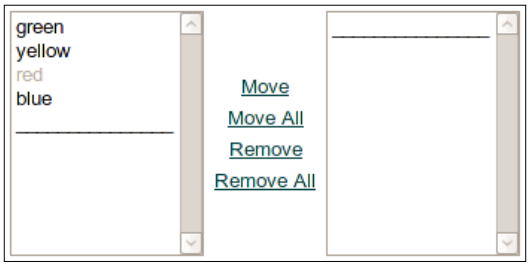
```
.AFErrorIcon:alias {
 content: '!'; }
```

The example we just saw shows a global icon that is set by using an alias, and can be used by multiple Trinidad components. An overview of alias selectors for global icons can be found at `http://myfaces.apache.org/trinidad/skin-selectors.html#Global_Icon_Selectors`. There are also some component-specific icons that can be set by using part selectors of the components, as shown by the example at the start of this section. Refer to the list of component selectors at `http://myfaces.apache.org/trinidad/skin-selectors.html#ComponentLevelSelectors` to find out which icon parts are defined for a certain component.

www.it-ebooks.info

Skinning text

Trinidad skinning also allows us to change text elements that are part of the Trinidad components. This can be useful if we're not satisfied with the default texts of the components. Let's take the `<tr:selectManyShuttle>` component as an example. As we saw in Chapter 5, by default this component renders text links that can be used to move items from the unselected list to the selected list, and the other way around. The texts are **Move**, **Move All**, **Remove**, and **Remove All**, as shown in the following image:



We already discussed that this isn't very intuitive. This can, of course, be solved by adding arrow-shaped icons to the links. An alternative solution is to apply a minimalist style and use the **<** and **>** characters. We can use the **>** character instead of **Move**, **>>** instead of **Move All**, and so on. Let's see how we can implement this.

First, we have to add a line to our `trinidad-skins.xml` file to let Trinidad know that we use custom texts for our skin:

```
<?xml version="1.0" encoding="UTF-8"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
  <skin>
    <id>mias.desktop</id>
    <family>mias</family>
    <render-kit-id>org.apache.myfaces.trinidad.desktop
    </render-kit-id>
    <style-sheet-name>skins/mias/mias-skin.css
    </style-sheet-name>
    <extends>suede</extends>
    <bundle-name>inc.monsters.mias.Messages
    </bundle-name>
  </skin>
</skins>
```


www.it-ebooks.info

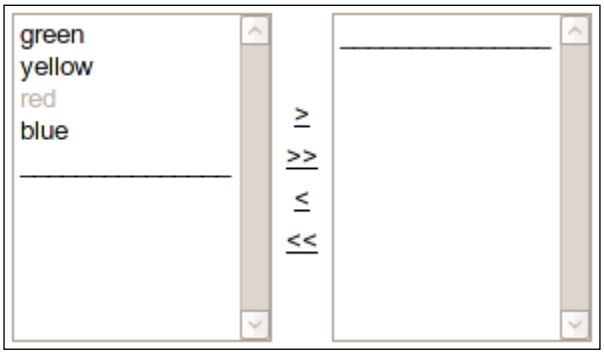
The changed lines are highlighted. We should always point to a message bundle that is in the classpath. In this example, we chose to use our application-wide message bundle. This way, we only have to translate a single file if we want to internationalize our application. But if we want to create a "self-contained" skin that can be reused in different projects, we can also refer to a message bundle that is inside the skin's JAR file, as long as it is on the classpath.

In our message bundle, we should use the keys that refer to the text part of the component that we want to change. Appendix E has a list of all of the keys and the default texts for the Trinidad components in Trinidad 1.2.12. Unfortunately, the Trinidad project doesn't provide any official list of these keys. However, they can be found in the `CoreBundle.xrts` file in the Trinidad source repository.

For the proposed changes to the `<tr:selectManyShuttle>` component, we should add the following lines to our `Messages.properties` file:

```
af_selectManyShuttle.MOVE_ALL = >>
af_selectManyShuttle.MOVE = >
af_selectManyShuttle.REMOVE_ALL = <<
af_selectManyShuttle.REMOVE = <
```

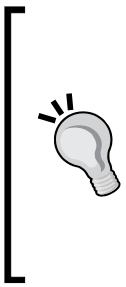
If we have multiple `Messages.properties` files for different languages, we shouldn't forget to add these lines to each of these files. Otherwise, the changed labels will only appear in the default language. The following image shows how the default texts are now replaced by the **<** and **>** signs in the shuttle component:



We shouldn't forget that the resource bundle keys are applied to one specific component type only. For example, there is also a `<tr:selectOrderShuttle>` component. This component still uses the **Move** and **Remove** texts, unless we also add four key / value pairs starting with `af_selectOrderShuttle` to our message bundle.

www.it-ebooks.info

Extending skins



www.it-ebooks.info

trinidad-config.xml file

The `trinidad-config.xml` is a configuration file specific to Trinidad. It should be placed in the `WEB-INF` directory, under the root of a deployment file (JAR, WAR, or EAR). A minimal `trinidad-config.xml` looks like this:

```
<?xml version="1.0"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
</trinidad-config>
```

Inside the `<trinidad-config>` element, several configuration elements can be added in a random order. They are covered in the rest of this section.

Although there is one single `trinidad-config.xml` file for a single application, it is possible to have different settings for (groups of) users, as JSF Expression Language can be used inside `trinidad-config.xml`. An example of the use of this feature is given in the *Letting the user choose the skin* section at the beginning of this chapter.

web.xml file

The `web.xml` file is the top-level configuration file for any Java EE web application. It also resides in the `WEB-INF` directory. Some tuning parameters for Trinidad are set as context parameters in the `web.xml` file. Context parameters can be added to the `web.xml` file as follows:

```
<context-param>
  <param-name> <!-- name goes here --> </param-name>
  <param-value> <!-- value goes here --> </param-value>
</context-param>
```

In the rest of this section, only the name and possible values of context parameters are mentioned. They should always be added to the `web.xml` file inside a `<context-param>` element, as shown in the previous code snippet.

In the following subsections, various related tuning options are grouped together, regardless of wether they are a `web.xml` options or `trinidad-config.xml` options. The file where each setting can be changed is indicated in the subsection heading, with **(T)** for `trinidad-config.xml` and **(W)** for `web.xml`.

www.it-ebooks.info

Accessibility

Trinidad has a couple of ways of configuring **accessibility** options at the application level. Remember that the settings made in `trinidad-config.xml` can use Expression Language to make them differ for different users.

Accessibility mode (T)

The `<accessibility-mode>` element in the `trinidad-config.xml` file can be used to choose between three different modes:

- `default`: Trinidad output supports most accessibility options by default.
- `inaccessible`: To minimize the output size, accessibility features can be turned off entirely, resulting in smaller pages that are not accessible for people with disabilities.
- `screenReader`: If you know beforehand that users with screen readers are going to use your application, you could use this setting. Output is optimized for screen reading, but other users may be negatively affected by this setting.

Accessibility profile (T)

The `<accessibility-profile>` element can be used to enable some specific accessibility options. You can configure more than one option in the same `<accessibility-profile>` element by separating them by whitespace characters. The following options exist:

- `high-contrast`: Output is optimized for use with high contrast settings on the client side
- `large-fonts`: Output is optimized for use with large font settings on the client side

These settings are not intended to change the appearance for all users, but rather to take some precautions in the output to prevent accessibility settings on the client side from breaking the functionality of the application. This feature depends on support by the skin that is used. At the moment, Trinidad's default skin does not take advantage of this setting.

www.it-ebooks.info

Lightweight dialogs (W)

By default, the Trinidad dialog framework uses browser window pop ups to render dialogs. However, these may be blocked by a pop-up blocker. To circumvent this, you can tell Trinidad to use "lightweight dialogs" instead. Lightweight dialogs are inline frames that are rendered atop the main content, like a pop up. Lightweight dialogs can be enabled by setting the `org.apache.myfaces.trinidad.ENABLE_LIGHTWEIGHT_DIALOGS` context parameter in the `web.xml` file to `true`. The default is `false`.

Performance

The first thing you have to check as far as **performance** is concerned is if the various debugging options are disabled, as they all add some overhead that will decrease the performance of the application. Apart from that, several specific options exist to fine-tune the performance of the application.

Page flow scope lifetime (T)

Perhaps the name of this setting is not entirely right. The `<page-flow-scope-lifetime>` element in the `trinidad-config.xml` file sets the number of `pageFlowScope` instances that are kept in memory. (There may be a relation with the lifetime of `pageFlowScope` instances, but still the name seems a little odd.) As this is a true application-wide setting, no Expression Language can be used inside this element. The default value is `15`. Increasing this value may cause the application to use more memory.

Uploaded file processor (T)

A default implementation of the `org.apache.myfaces.trinidad.webapp.UploadedFileProcessor` interface is provided by Trinidad. This should be sufficient for most file uploads. However, when file uploading performance is critical, you might want to implement your own `UploadedFileProcessor` implementation. In this case, you should use the `<uploaded-file-processor>` element in the `trinidad-config.xml` file to configure the fully-classified name of your implementation class.

www.it-ebooks.info

State saving (W)

State saving is an important process in JSF applications in general. It's all about saving the state of the application between requests for a particular client. The standard JSF parameter `javax.faces.STATE_SAVING_METHOD` in `web.xml` can be used to choose between state saving on the `client` or the `server`. The latter is JSF's default setting. In case client-side state saving is chosen, the method of state saving on the client can be further refined by some Trinidad-specific settings in the `web.xml` file:

- The standard JSF way to save the state on the client is by putting the entire state in a single, hidden form field on the client. However, Trinidad uses another tactic. It saves the state on the session and saves only a token on the client to uniquely identify the state. This behavior can be controlled by the `org.apache.myfaces.trinidad.CLIENT_STATE_METHOD` context parameter in `web.xml`. Set it to `all` to get the standard JSF method, or set it to `token` to get the Trinidad behavior. The latter is the default.
- If client-side token state saving is configured, the maximum number of tokens can be set with the `org.apache.myfaces.trinidad.CLIENT_STATE_MAX_TOKENS` context parameter. It defaults to `15`. This means that if the user navigates to the 16th page, the state of the first page will be forgotten. You might want to increase the value if users use the back button a lot, or have multiple windows or tabs open at the same time.
- Another optimization for client-side token state saving that Trinidad does is caching the view root for every token. This can increase efficiency for applications that make a lot of AJAX requests to the server, as the view root now doesn't have to be built from scratch for each AJAX request that has to be handled. However, there are some known issues with Tomahawk's `<t:saveState>` component and Facelets template texts appearing twice. This behavior can be turned off by setting the `org.apache.myfaces.trinidad.CACHE_VIEW_ROOT` context parameter to `false`. The default is `true`.

The Trinidad developers recommend using client-side state saving with the token method for best performance. If your application server is low on memory, the `all` option for client-side state saving will save you memory at the expense of lower performance.

www.it-ebooks.info

Application view caching (W)

For improved scalability, Trinidad has a built-in caching mechanism for the view state at the application level. It caches the initial view state of every page at the application level, as the initial view state is identical for every user. This saves the server from building an initial view state tree from the page definition every time a page is viewed. Only when a user posts data back to the server, an individual view state of that page is created for the unique user.

However, there are some downsides to this caching mechanism. First, changes to pages are only detected upon application server restart, making it less practical to use this caching during development. Second, there are ways to break the assertion that the initial view state of a page is identical for every user:

- If you use non-JSF conditionals (such as `<c:if>` or `<c:choose>`) to dynamically add or remove JSF components from the component tree. This can be worked around by using the `rendered` attribute of the component instead of surrounding it by a conditional element.
- If you use an iterator component (such as `<c:forEach>` or `<tr:forEach>`) to add components to the tree. This may sometimes be worked around by using a single component instead. For example, you could use the `<f:selectItems>` component instead of the `<f:selectItem>` component surrounded by the `<c:forEach>` component.

If you can live with the downsides, the application view caching mechanism can be enabled by the `org.apache.myfaces.trinidad.USE_APPLICATION_VIEW_CACHE` context parameter in the `web.xml` file. Set it to `true` to enable caching, or `false` to disable it.

Debugging

Trinidad has some options for easy **debugging**. They all come with a performance penalty, so don't forget to turn them off in your production environment!

www.it-ebooks.info

Enabling debug output (T)

Trinidad can generate a lot of useful debug output in both the generated XHTML code and the server logs. Of course, this comes with a performance penalty, so you should make sure that debug output is turned off in production environments. For debugging, a lot of useful information is generated, such as:

- Extra comments in the generated XHTML, such as markers that indicate by which JSF component a piece of code was generated.
- Extra warnings in the server log if invalid XHTML code is generated. For example, nesting errors can be detected easily this way. These errors may have stayed unattended otherwise, as most browsers try to always render a page, even if the XHTML is not valid.
- Indented XHTML code. Normally, the generated XHTML is not indented to save processing time and bandwidth. But for debugging, it is nice to have more readable code.

Debug output can be turned on by adding the `<debug-output>` element to the `trinidad-config.xml` file. Values can be `true` (debug output on) or `false` (debug output off). The default value is `false`.

Turning off compression and obfuscation (W)

By default, Trinidad uses compression and obfuscation on JavaScript and CSS to optimize the size of the separate JavaScript and CSS files. For debugging and / or skin development, it may be useful to disable these optimizations during the development stage. Remember to turn the optimizations on again in your production environment!

- Obfuscation of JavaScript can be turned off by setting the `org.apache.myfaces.trinidad.DEBUG_JAVASCRIPT` context parameter in `web.xml` to `true`
- Compression of CSS class names can be turned off by setting the `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION` context parameter in `web.xml` to `true`

www.it-ebooks.info

Changing deployed files (W)

When developing, it may be useful to be able to apply changes directly to deployed files, instead of redeploying the entire application on every change. Some combinations of IDEs and application servers can do this transparently. For this to work, Trinidad can be configured to monitor file changes. This comes at the expense of extra overhead, so it is a good idea to not use this in production environments. File monitoring can be turned on by setting the `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION` context parameter in the `web.xml` file to `true`.

Appearance

Several global-tuning options regarding the **appearance** of the application exist. The following subsections cover them.

Client validation (T)

Client validation can be configured in the `trinidad-config.xml` file by adding the `<client-validation>` element. Three different values can be set:

- `inline`: Client-side validation results are presented the same way as server-side validation results—via the message areas in the pages. This is the default.
- `setalert`: Results of client-side validation are shown to the user by means of a JavaScript alert pop up. This way the user can notice a difference between validation that is executed on the client side and validation that is executed on the server side. Users may find these pop ups annoying, though.
- `disabled`: No client-side validation will be performed.

Output mode (T)

Through the `<output-mode>` element in the `trinidad-config.xml` file, Trinidad can be configured to produce output that is optimized for printing or e-mailing. The following values can be set:

- `default`: Output optimized for on-screen display; this is the default setting
- `printable`: Output optimized for printing
- `email`: Output pages are optimized to be able to send them by e-mail

www.it-ebooks.info

Skin family (T)

As described in the *Skinning* section, the `<skin-family>` element can be used to select which skin is used. The value should be the skin family name of the desired skin. The default value is `minimal`, which is the name of the default skin family.

Localization

Localization often goes automatically by configuring the correct locale. However, some localization settings can be overridden in the application configuration. This may be useful for overriding locale settings with static defaults for all users, or to enable the user to configure his or her own settings. Each of the following subsections discusses a localization-related setting.

Time zone (T)

A time zone can be configured in the `trinidad-config.xml` file to display times in the correct time zone. By default, Trinidad's `DateTimeConverter` tries to use the time zone setting of the user's browser to determine the time zone. However, you might want to have a time zone setting in your application instead. In this case, you could bind the `<time-zone>` element in the `trinidad-config.xml` file to a `java.util.TimeZone` object.

Two-digit year start (T)

Another setting for the `DateTimeConverter` is the `<two-digit-year-start>` element in `trinidad-config.xml`. This helps the converter to interpret two-digit year values when parsing date strings. The default setting is `1950`, which means that values below `50` are interpreted as years past `2000`; while values from `50` and up are interpreted as years before `2000`. This setting also allows Expression Language, enabling you to make a user setting out of it if desired.

Reading direction (T)

In some languages, such as Arabic and Hebrew, text is written and read from right to left instead of left to right. Normally, Trinidad will automatically choose the correct text direction based on the configured locale. Should you want to override the setting of the locale, then you can use this setting. It is set by adding a `<right-to-left>` element to the `trinidad-config.xml` file and setting it to `true` or `false`, either by a literal value or by a JSF expression that evaluates to a `Boolean`.

www.it-ebooks.info

Number notation (T)

Normally, Trinidad's `NumberConverter` derives its notation from the current locale. However, should you want to override this with a fixed value or a value that can be configured from within your application, the `<number-grouping-separator>` component, the `<decimal-separator>` component, and the `<currency-code>` component settings in the `trinidad-config.xml` file can be used. All three accept either a fixed character or an expression that evaluates to a `Character`.

Summary

This last chapter of the three chapters on Trinidad introduced Trinidad's skinning capabilities. We also had a look at the various ways to tune Trinidad for optimal performance, accessibility, and appearance. In the next chapter, we will see how we can integrate our web application with a backend system.

www.it-ebooks.info

8
Integrating with the Backend

As the MyFaces project is largely about view technologies, we have focussed on the view layer until now. For the sake of simplicity, we did not care about persisting our data in a database or something similar. However, in a real-world application, we do need a persistence solution most of the time. That's why we'll have a quick look at integrating our JSF web application with a backend solution. We will only look at some basics here, as lots of books could be written about backend technologies. (And luckily, many good books on the topic are already available.) This chapter focuses on using Java persistence, as it is included within the Java EE standard without the use of any additional libraries.

In this chapter we will learn about the following topics:

- Basic knowledge about the Model-View-Controller design pattern
- Basic knowledge about Enterprise JavaBeans (3.0)
- Basic knowledge about the Java Persistence API (1.0)
- Using Enterprise JavaBeans (EJB) facades in your web applications
- Limitations and problems of using EJB without additional frameworks

The Model-View-Controller architecture

It is a common practice to implement the **Model-View-Controller** (MVC) design pattern in a Java EE application. In fact, parts of the Java EE standard are designed around this pattern. The goal of the MVC pattern is to separate business logic from the user interface. The MVC pattern splits an application into three separate parts, each with its own responsibilities, as follows:

- **Model**: This part is responsible for manipulating application data. In other words, the Model implements the business logic.

www.it-ebooks.info

- **View**: This part is responsible for presenting the contents of the Model to the user, and providing ways for the user to send data or commands to the application.
- **Controller:** This part defines the behavior of the application, and is responsible for receiving the user's input.

So far in this book, we have only created a view and a controller for an application. In a JSF-based application, the View consists of all of the JSP or Facelets pages. The JSF Controller and all of the backing beans together form the Controller of the application. We bundled these parts into a **Web Archive**, a WAR file, in order to deploy it to an application server.

As we just discussed, the Model layer of the application implements the business logic, and is responsible for persisting application data. In a typical application, this layer is bundled into its own JAR file. In this chapter we will look at the EJB 3.0 and JPA 1.0 frameworks that are part of the Java EE standard. These can help us to implement a Model layer for our application.

Setting up the Java EE application structure

In a typical Java EE application, persistence services reside in their own archive. They are referred to as an **EJB JAR** (Enterprise JavaBean JAR). In order to let our web application (the View and Controller) and the persistence services (the Model) easily work together, both the EJB JAR and our previously-generated WAR are wrapped within a single **Enterprise Archive** (**EAR**). So let's first create a skeleton EJB JAR and EAR.

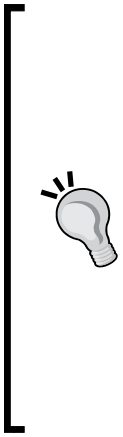
Creating a skeleton EJB JAR

An EJB JAR is nothing more than a simple JAR that contains Java classes in a package structure. What distinguishes an EJB JAR from a normal JAR is the fact that there are some extra files in the `META-INF` directory inside the JAR file. In a normal JAR file, this directory only contains the `MANIFEST.MF` file. In an EJB JAR, the `META-INF` directory also contains some extra files. See the following table for an overview:

Filename	Mandatory	Description
`MANIFEST.MF`	mandatory	Nothing special compared to a normal JAR's `MANIFEST.MF`.
`ejb-jar.xml`	mandatory	Deployment descriptor for the EJB JAR. An example of a minimal `ejb-jar.xml` file is given after this table.

www.it-ebooks.info

Filename	Mandatory	Description
`persistence.xml`	optional	Configuration file for the Java Persistence API (JPA).
`orm.xml`	optional	Optional configuration file for configuring object-relational mapping without using annotations. Because the use of annotations is the default, this file can be omitted most of the time.
application server specific	optional	Some application servers define specific configuration files for EJB JARs.



www.it-ebooks.info

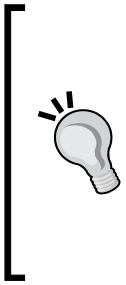
Filename	Mandatory	Description
`MANIFEST.MF`	mandatory	Nothing special compared to a normal JAR's `MANIFEST.MF`.
`application.xml`	mandatory	Deployment descriptor for the EAR that describes the structure of the application. An example of an `application.xml` file is given after this table.

In our case, we could have a `MIAS-EAR.ear` file that contains our previously-created `MIAS.war` file and our newly created `MIAS-EJB.jar` file. This structure should be reflected in the `application.xml` file, as follows:

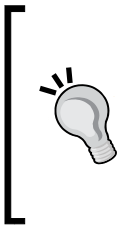
```
<?xml version="1.0" encoding="UTF-8"?>
<application
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:application="http://java.sun.com/xml/ns/
                                                javaee/application_5.xsd"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/application_5.xsd"
       version="5">
  <display-name>MIAS-EAR</display-name>
  <module>
    <web>
      <web-uri>MIAS.war</web-uri>
      <context-root>MIAS</context-root>
    </web>
  </module>
  <module>
    <ejb>MIAS-EJB.jar</ejb>
  </module>
</application>
```

As you can see, this is nothing more than a reflection of the internal directory structure of the EAR, along with a little bit of extra information.

www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

(=Java DB) database.

- `//localhost:1527`: The database is installed on the localhost machine. `1527` is the default port number on which Java DB listens for new connections.
- `/test`: The name of the database that we wish to connect to is `test`.
- `;create=true`: If no database with the name `test` exists, Java DB will create one.

Don't forget the trailing `;`. ij will do nothing if a command does not end with `;`. To verify that we're really connected, we can use the following command as an example:

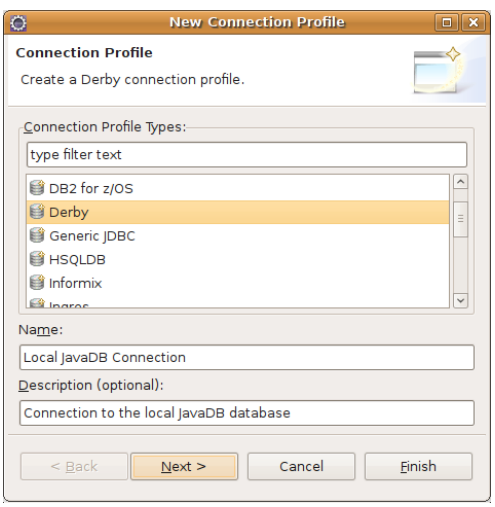
```
show schemas;
```

This will cause a list of database schemas to be displayed.

Managing the database

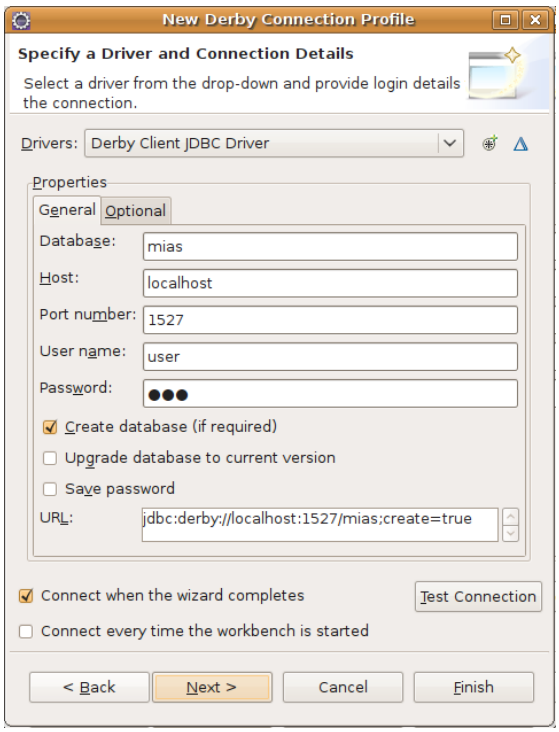
Using a command-line tool such as ij is perhaps not the easiest way to manage a database. Luckily, there are easier alternatives, of which we will look into one briefly, here. Recent versions of the Eclipse IDE have a **Data Source Explorer**, which is a part of the **Data Tools Platform**. This can be made visible by selecting the **Database Development perspective** via **Window | Open Perspective | Other... | Database Development | OK**. Now, let's see how we can connect to our Java DB database from the Data Source Explorer.

1. In the **Data Source Explorer** pane is a folder called **Database Connections**. To create a new database connection, right-click on that folder and select **New...**:



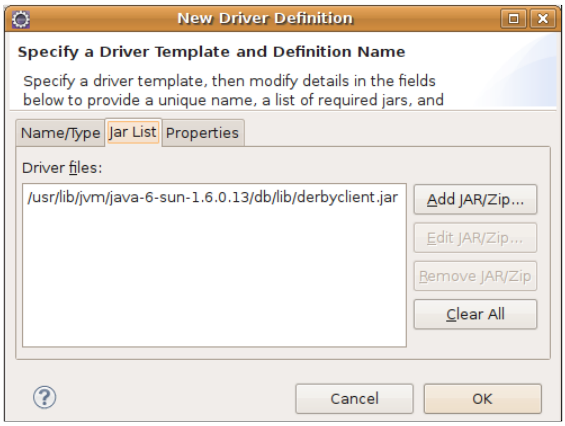
www.it-ebooks.info

2. The **New Connection Profile** window appears. Select **Derby** as the **Connection Profile Type** and give the connection a sensible name.
3. Click on **Next >** to go to the following screen:



4. Make sure that **Derby Client JDBC Driver** is selected in the **Drivers** combobox. If this is not available, then you can create a new driver definition via the **New Driver Definition** button ( ), which is located to the right of the combobox. If Derby Client JDBC Driver is already available in the list, you can skip to step 7.
5. In the **New Driver Definition** window on the **Name/Type** tab, you can just select the latest version of the **Derby JDBC Client driver**. You can keep the default name.
6. On the **Jar List** tab, make sure that the selected JAR is in the same directory where your Java DB database is located. If this is not the case, remove the JAR and add a new one via the **Remove JAR/Zip** and **Add JAR/Zip...** buttons.

www.it-ebooks.info



7. Back in the **New Derby Connection Profile** window, enter a database name, and make sure that the **Create database (if required)** option is selected. All other options can be left at the default settings.
8. For **User name** and **Password**, any values can be entered, as long as they are at least three characters long. By default, no authentication is required to connect to the Java DB database.

Once the connection has been defined and opened, the Data Source Explorer can be used to browse the Java DB database. By default, several schemas exist in the database—one of them being `APP`. This is the schema that can be used for application data. In a newly-created database, this schema is empty.

Creating a table for employees

As an example, we will create a table for employees. This table will later be used to persist `Employee` objects. We can use some simple SQL commands to create the table. They can either be typed into Java DB's ij tool or in the SQL Scrapbook of Eclipse's Data Source Explorer. (The latter is available by right-clicking on the database connection in the Data Source Explorer and choosing **Open SQL Scrapbook**.) To create a table for employees, the following SQL command can be issued:

```
CREATE TABLE app.employees (
    ID INTEGER GENERATED ALWAYS AS IDENTITY CONSTRAINT
                                    employees_pk PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    job_title VARCHAR(50) NOT NULL,
    hire_date DATE NOT NULL,
    birth_date DATE NOT NULL,
    salary INTEGER
);
```


www.it-ebooks.info

Populating the table with data

For testing our application later on, it is handy if the table already contains some data. We can manually insert some data via either ij or the SQL Scrapbook by issuing a series of insert statements, like this:

```
INSERT INTO app.employees (first_name, last_name, birth_date,
                           hire_date, job_title, salary)
VALUES ('Henry.J.', 'Waternoose', DATE('1953-05-11'),
        DATE('1990-04-01'), 'Chief Executive Officer', 5000);

INSERT INTO app.employees (first_name, last_name, birth_date,
                           hire_date, job_title, salary)
VALUES ('James.P.', 'Sullivan', DATE('1978-04-06'),
        DATE('1999-07-15'), 'Senior Scarer', 2500);

INSERT INTO app.employees (first_name, last_name, birth_date,
                           hire_date, job_title, salary)
VALUES ('Mike', 'Wazowski', DATE('1984-12-03'),
        DATE('2003-11-01'), 'Junior Scarer', 2000);
```

The data can be verified by issuing the following select statement:

```
SELECT * FROM app.employees;
```

Now that we have a database up and running, and it has been populated with some data, we're ready to start implementing the Model layer of our application.

Implementing the Model

Now that we have prepared the structure for our Java EE application, it's time to focus on the implementation of the Model layer—in other words, creating the contents for our EJB JAR.

Creating an entity

As stated in the introduction of this chapter, we are going to use Enterprise JavaBeans (EJB) 3.0 and Java Persistence API (JPA) 1.0, both of which are part of the Java EE 5 standard. In the JPA, persistent data is represented by special objects called entities. Therefore, to be able to use persistent `Employee` data in our application, we need an `Employee` entity definition. In JPA, an entity definition is nothing more than a Java class with some extra annotations.

www.it-ebooks.info

We will base our entity on the table that we created earlier. This means that for every column in the table we need a bean property in our class. Where needed, JPA annotations should be added. This will lead to an entity class that looks like this:

```
package inc.monsters.mias.data;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;

@Entity
@Table(name="EMPLOYEES", schema="APP")
public class Employee implements Serializable {
private static final long serialVersionUID = 1L;
private int id;
private Date birthDate;
private String firstName;
private Date hireDate;
private String jobTitle;
private String lastName;
private int salary;
public Employee() {

}
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
public int getId() {
return this.id;
}

public void setId(int id) {
this.id = id;
}

@Temporal(TemporalType.DATE)
@Column(name=""BIRTH_DATE")
public Date getBirthDate() {
return this.birthDate;
}

public void setBirthDate(Date birthDate) {
this.birthDate = birthDate;
}
@Column(name="FIRST_NAME")
public String getFirstName() {
return this.firstName;
}
```


www.it-ebooks.info

```
    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    @Temporal(TemporalType.DATE)
    @Column(name="HIRE_DATE")
    public Date getHireDate() {
      return this.hireDate;
    }

    public void setHireDate(Date hireDate) {
      this.hireDate = hireDate;
    }

    @Column(name="JOB_TITLE")
    public String getJobTitle() {
      return this.jobTitle;
    }

    public void setJobTitle(String jobTitle) {
      this.jobTitle = jobTitle;
    }

    @Column(name="LAST_NAME")
    public String getLastName() {
      return this.lastName;
    }

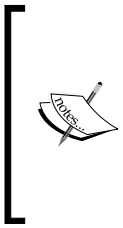
    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public int getSalary() {
      return this.salary;
    }

    public void setSalary(int salary) {
      this.salary = salary;
    }
  }
```

The `@Entity` annotation at the top marks this class as being an entity class. The `@Table` annotation is added because the name of the database table is different from the name of the entity. It is also added to include information on which database schema the table is in. On the `getId()` method, two annotations are set. The `@Id` annotation marks the `id` property as being the primary key in the database table. This means that the JPA framework knows that this property can be used to uniquely identify an entity object. The `@GeneratedValue` annotation tells JPA that the database will generate a value for this field every time that a new record is inserted.

www.it-ebooks.info



www.it-ebooks.info

As per the EJB 3.0 standard, such a service interface is called a **Session Bean**. For each Session bean, EJB requires that we define an interface and create an implementation. Let's focus on the interface first. Suppose we need only two functionalities: to be able to get a list of all employees, and to update an `Employee` object with new data. Our interface could then look like this:

```
package inc.monsters.mias.data.facade;

import inc.monsters.mias.data.Employee;
import java.util.List;

public interface EmployeeService {
public List<Employee> getEmployees();
public void updateEmployee(Employee emp); }
```

That's pretty simple, isn't it? But now we have to implement this interface. Let's see how this can be done:

```
package inc.monsters.mias.data.facade;

import inc.monsters.mias.data.Employee;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.*

@Stateless
public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="MIAS-EJB")
private EntityManager em;

public EmployeeServiceBean() {
}
@Override
public List<Employee> getEmployees() {
Query q = em.createNamedQuery("Employee.all");
return q.getResultList();
}

@Override
public void updateEmployee(Employee emp) {
if (null == em.find(Employee.class, emp.getId())) {
throw new IllegalArgumentException(
```


www.it-ebooks.info

```
                        "Unknown employee id: " + emp.getId());
        }

        em.merge(emp);
      }
    }
```

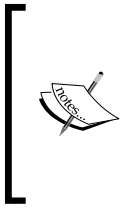
A lot of interesting things can be noted about this implementation. Let's start at the top. The class is preceded by a `@Stateless` annotation. This tells the EJB framework that this is a Stateless Session Bean that does not have a state to preserve. This means that a call to any method of this bean does not depend on values that might be set by a previous call to any method of the class. Now look at the name of the class. As per convention, the name of the implementation class is the same as the name of the interface that it implements, with `Bean` appended to it.

Next, we have a private member variable called `em`, which is of the `EntityManager` type. `EntityManager` is a class provided by the JPA framework. It has a lot of methods that are needed for managing entities. The `@PersistenceContext` annotation ensures that the EJB framework will inject a reference to an `EntityManager` object whenever an instance of the `EmployeeServiceBean` class is created. Of course, this `EntityManager` does need a database connection. This is why the annotation has the `unitName="MIAS-EJB"` attribute. This tells the JPA framework to look for a persistence unit with the name `MIAS-EJB`. We'll see how to create this, later on.

This brings us to the `getEmployees()` method, which is a pretty short method. First, a `Query` object is created, and then the `getResultList()` method is called on that `Query` object. In order for the `createNamedQuery()` method to work, a named query should be defined in the `Employee` entity, which we will look at in the next section. As an alternative, the `createQuery()` method could be used. This method accepts a `String` containing the query directly. However, it might be a good idea to keep all queries with the entity object that they are related to. This also helps to reuse queries instead of writing the same query several times.

The `updateEmployee()` method is a bit more complicated. It takes an `Employee` object as an argument. It is expected that this `Employee` object has updated data that is not in the database. In the first line, the `find()` method is called on the `EntityManager` object. This method takes a class and a value. The class must be an entity class corresponding to some database table. The value should be such that it can be a valid primary key value in that table. In this case, we are using the `find()` method to verify that the entity already exists in the database, otherwise we wouldn't be able to update it. If `find` returns `null`, then the entity does not exist, and thus we throw an exception.

www.it-ebooks.info



www.it-ebooks.info

Although this is simple, it is more elegant to group all queries together as named queries. This also has the benefit that we don't have to repeat ourselves if we need the same query more than once. Named queries can be defined by adding some annotations just before the definition of our entity class. In this case, the first lines of our `Employee` class become:

```
@Entity
@Table(name="EMPLOYEES", schema="APP")
@NamedQueries({
  @NamedQuery(name="Employee.all",
              query="SELECT emp FROM Employee emp"),
  @NamedQuery(name="Employee.byId",
              query="SELECT emp FROM Employee emp
                     WHERE emp.id = :id")
})
public class Employee implements Serializable {
  ...
}
```

The name of the named query can be used to refer to a call to `createNamedQuery()`, as is shown in the example in the *Creating a service facade* section of this chapter.

Defining persistence units

To define a persistence unit, we need a `persistence.xml` file. This file should be created in the `META-INF` directory of our EJB JAR. In our case, the file could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
        xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/
                            persistence
                            http://java.sun.com/xml/ns/
                            persistence/persistence_1_0.xsd">
  <persistence-unit name="MIAS-EJB">
     <jta-data-source>jdbc/miasDataSource</jta-data-source>
  </persistence-unit>
</persistence>
```

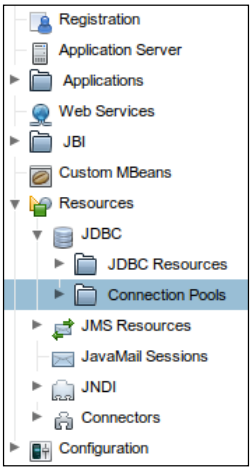
The name of the unit is important here. This is the name that is referred to in the `@PersistenceContext(unitName="MIAS-EJB")` annotation in our service bean. Another important thing is the `<jta-data-source>` element. This binds the persistence unit to a data source. The data source has to be provided by the Java EE container. We will discuss this in more detail later.

www.it-ebooks.info

Defining a data source

To define a data source, we have to configure the application server. We cannot cover the steps needed for every application server, here. In this section, we see how we can configure a data source on a GlassFish application server. As the steps to be taken are similar for most application servers, it should be possible to figure out how it can be done for any other application server. The steps are as follows:

1. Log in to the administration interface of the application server. For a typical GlassFish installation, the login page can be reached via `http://localhost:4848`. The default username is **admin** and the password is **adminadmin**.
2. Navigate to **Resources** | **JDBC** | **Connection Pools**. In the right half of the screen, click on the **New...** button.



3. In the **New JDBC Connection Pool (Step 1 of 2)** screen, enter the following data:
   - Name: `JavaDBConnectionPool`
   - Resource Type: `javax.sql.DataSource`
   - Database vendor: `Derby`
4. Click on **Next**.
5. In **Step 2 of 2**, make sure that the **Datasource Classname** is set to `org.apache.derby.jdbc.ClientDataSource`. Other settings can be left with their default values. Also make sure that, under **Additional Properties**, at least the following properties are set:
   - PortNumber: `1527`

www.it-ebooks.info

- ServerName: `localhost`
- DatabaseName: `mias`

6. Click on **Finish**.
7. After creating a connection pool, we still have to create a data source based on that pool. To do so, navigate to **Resources** | **JDBC** | **JDBC Resources**.
8. Click on the **New...** button at the top of the table, in the right half of the screen.
9. In the **New JDBC Resource** screen, make sure that the following data is set:
   - JNDI name: `jdbc/miasDataSource`
   - Pool Name: `JavaDBConnectionPool`
   - Status: `enabled`

   The JNDI name is the name that is referred to in the `<jta-data-source>` element in the `persistence.xml` file. The Pool Name is the name of the Connection Pool that was created in steps 2 to 5.

Using the service facade in the View layer

Now that we've created our service facade, it's time to adapt our web application to use this service facade instead of the dummy objects we used so far. For the Employees, we had a managed bean that contained a list of employees. To get access to our persisted `Employee` objects, we have to call the methods in our service facade from the backing bean of our page. We already have the `EmployeesTable` backing bean, so let's add some methods there to access our facade:

```
private List<Employee> empList;

public List<Employee> getEmployees() {
  if (null == empList) {
    empList = service.getEmployees();
  }
  return empList;
}

public void saveSelected(ActionEvent event) {
  RequestContext rc = RequestContext.getCurrentInstance();
  Employee emp = (Employee) rc.getPageFlowScope()
                                    .get("selectedEmployee");
  service.updateEmployee(emp);
}
```


www.it-ebooks.info

The `getEmployee()` method simply delegates the call to the service. The result is stored in a member variable. Combined with the `null` check, this guarantees that the `getEmployees()` method on the facade will only be called once during the lifetime of the `EmployeesTable` bean. This is important for good performance of the application, as getter methods in JSF managed beans can be called more than once during the lifecycle.

The `saveSelected()` method has to get the `selectedEmployee` object from the page flow scope first, and then call the `updateEmployee()` method with the `selectedEmployee`. But the interesting question here is: How is the `service` variable declared, and how did it get a reference to our service bean in the EJB JAR? The answer: is we should declare `service` as being of the `EmployeeService` type. To get a reference to our service bean, we just have to put an `@EJB` annotation before it. So the declaration of this member variable will look like this:

```
@EJB
private EmployeeService service;
```

The `@EJB` annotation will cause the EJB framework to look for an Enterprise Java Bean of the `EmployeeService` type. As our `EmployeeServiceBean` class implements the `EmployeeService` interface, and is an EJB, thanks to the `@Stateless` annotation, it satisfies both. So at runtime, a reference to our service bean will be injected.

Updating the pages

Now, we have to adapt our pages to use the new methods, instead of the managed bean. In our `Employees.xhtml` page, we only have to change the value property of the `<tr:table>` element, as follows:

```
<tr:table var="emp"
          value="#{empsTable.employees}"
          rows="20"
          id="kids"
          rowBandingInterval="1"
          horizontalGridVisible="false"
          rowSelection="multiple"
          binding="#{empsTable.table}">
```


www.it-ebooks.info

In the `EditEmployee.xhtml` page, we have to add action listeners to the `OK` and `Apply` buttons. This will make the button definitions look like:

```
<tr:panelButtonBar>
  <tr:commandButton text="#{msg.apply}"
                    action="apply"
                    partialSubmit="true" id="btnApply"
                    actionListener="#{empsTable.saveSelected}"/>
  <tr:commandButton text="#{msg.ok}"
                    action="ok"
                    actionListener="#{empsTable.saveSelected}"/>
  <tr:commandButton text="#{msg.cancel}"
                    action="cancel"
                    immediate="true" />
</tr:panelButtonBar>
```

The highlighted lines are added; the rest is unchanged. It is important to note that very little changes are needed in the pages. As the properties of our new entity objects are the same as the properties that our dummy objects had, we can still refer to them via the JSF Expression Language.

Limitations and problems

Although EJB 3.0 is a very elegant framework, it has some limitations. These limitations will not show up in simple applications like the one we used as an example in this book, but in real-life applications, they will.

Transactions

One of the limitations that you can come across when using JSF for the View and Controller, and EJB for the Model, is the lack of coordination between JSF and EJB when it comes to transactions. In databases, all changes to data happen within a transaction. In the example in this chapter, we updated a single row in the `EMPLOYEES` table. The container will simply start a transaction just before the update takes place, and close the transaction immediately after the update. But in more complex applications, it is often necessary to perform multiple actions within a single transaction. This is the case when related records are updated, added, or removed. A database transaction can only be closed if all of the constraints are satisfied.

www.it-ebooks.info

For example, there might be a foreign key constraint saying that a record in a `Departments` table can only be removed if no employees are working in that department. Imagine a reorganization of the company where a certain department will be abolished. All employees of that department will be transferred to other departments. The transfer of those employees, as well as the removal of the department, must take place in a single transaction. This is because, in case the transfer of a single employee doesn't succeed due to some other constraint, the whole transaction can then be canceled.

Now back to our Java EE application. The problem is that the JSF framework has no idea what a transaction is. As long as transactions are opened and closed within a single screen, there's no real problem. However, when we need to keep a transaction open, spanning over multiple screens, things can get very complicated.

Validation of data

According to the Model-View-Controller pattern, validation of data should take place in the Model layer of the application because validation can be seen as business logic. In most real-life applications, data validation is implemented in the Model layer. But as we want to give our users fast feedback on entered data, a lot of validation is often duplicated in the View layer. This can create all sorts of problems, especially when validation rules change and the Model layer is updated, but the View layer is forgotten.

Having no validation in the Model and completely depending on the View for validation is not an option. Validation belongs in the Model layer. There could be a second interface (for example, a Web Service) that is built on the same model. Having no validation in the model could allow the second interface to enter invalid data.

Summary

In this chapter, we learned the basics of EJB and JPA by implementing a simple persistent Model layer for our web application. We also discussed some of the limitations and problems that arise when using JSF in combination with "plain" EJBs.

In the next chapters, we will have a look at the MyFaces subprojects that can help us overcome the limitations and problems discussed in the last section of this chapter. **MyFaces Orchestra**, discussed in the next chapter, can help us solve the problems with transactions, and **MyFaces Extensions Validator** (ExtVal) can help us define our validations in a single place. MyFaces ExtVal will be covered in Chapter 10.

www.it-ebooks.info

9
MyFaces Orchestra

The MyFaces Orchestra project contains various tools and goodies that are focussed on making Java EE application development easier and more productive. In particular, the conversations feature is very powerful as it gives us an elegant way to handle persistence transactions that have to span multiple screens. By using this feature, we can solve one of the shortcomings mentioned at the end of the previous chapter. This chapter guides you through the basic setup of Orchestra, which is not trivial as it also involves adding the Spring framework to the project. After the setup, we will look at Orchestra's conversations feature. We will also have a quick look at some other useful Orchestra features.

It should be noted here that some of the shortcomings that Orchestra fixes are also fixed in the JSF 2.0 specifications. But as JSF is still a View-oriented standard, it cannot offer the tight integration with a persistence layer that Orchestra uses to automatically manage transactions for us.

After reading this chapter, you will be able to:

- Add the Spring framework to a web application project
- Configure a basic Spring application context
- Set up a web project to use MyFaces Orchestra
- Use Orchestra's ViewController concept
- Configure and use Orchestra conversations
- Generate a simple form with Orchestra's DynaForm component

www.it-ebooks.info

Setting up Orchestra

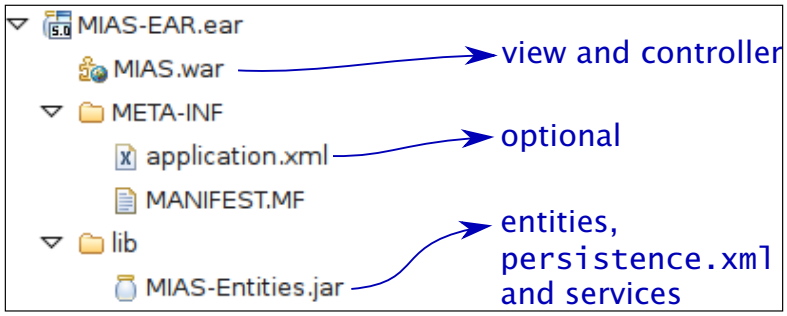
This section outlines the steps needed to prepare our project to use MyFaces Orchestra. This involves changing the structure of our application, adding and configuring the Spring framework, and adding the Orchestra libraries.

Adapting the application structure

Orchestra runs in the web application container (the WAR file). To be able to manage **transactions** in the persistence framework, Orchestra must have access to the persistence classes. (Obviously, this isn't necessary if we don't want to use Orchestra's persistence support.) This means that we can't put our persistence classes into a separate EJB container. Fortunately, as Orchestra uses Spring, we don't need an EJB container because Spring is capable of performing the tasks that are typically done by an EJB container. Also, using both Spring and a separate EJB container would make things overly complex.

Of course, putting the persistence and presentation layer in the same container has some down-sides too. For very large applications, we no longer have the option to run both layers on different servers in order to spread the load. As an alternative, clustering of the Web Container can be an option. And for small and medium-sized applications, this shouldn't be an issue at all.

It is also a good idea to have presentation logic and persistence in different projects during development. However, using Orchestra doesn't prevent us from doing so. We have to change the application structure a bit to be able to use different projects. Instead of putting our Entities, `persistence.xml`, and Service Beans into a special EJB JAR, we put them into an "ordinary" JAR. We put this JAR in the `lib` directory of the enclosing EAR. In this way, the WAR, which is also a part of the EAR, has access to it. The following image gives a schematic overview of this structure:



www.it-ebooks.info

The most important contents of each of the contained archives are listed in the following table:

Archive	Contents
`MIAS-Entities.jar`	Entities: `Empoyee.java`, `Kid.java`.
	Configuration file: `persistence.xml` (in `META-INF`).
	Service Facade: `EmployeeService.java`, `KidService.java`, `EmployeeServiceBean.java`, `KidServiceBean.java`.
`MIAS.war`	All other stuff, such as `*.xhtml` pages, `faces-config.xml`, backing beans, and so on.

Downloading the Spring framework

As Orchestra makes use of the Spring framework, we have to add the Spring framework to our web application file (WAR file) as a dependency. Orchestra requires at least Spring 2.0. The current production version at the time of writing this book is 2.5.6, which works fine with Orchestra, so let's take that one. It can be downloaded from `http://www.springsource.com/products/spring-community-download`. On this page, we're asked about our personal details. However, this can easily be skipped by clicking on the **I'd rather not fill in the form. Just take me to the download page** link below the form. This brings us to the actual download page. Although the current development release is at the top of the list, we're probably better off with the current GA release of the Spring Framework, which is just below the development build. Choose the version "with dependencies", to make sure that we have all of the JARs on which Spring depends, readily available when we need them later on.

Now we have to make sure that at least all JARs in the `dist` directory of the downloaded archive are added to the `WEB-INF/lib` directory of our WAR file. (In Eclipse, this can be done by selecting **Properties** from the context menu of the project and then going to **Java Build Path | Libraries**, and adding a "user library" via the **Add Library...** button.)



Extra dependency

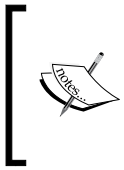
It turns out that at least in our example application, we also need the `cglib-nodep-2.1_3.jar` file to be in the classpath. This JAR can also be added to the `WEB-INF/lib` directory of our WAR. If you downloaded the "with dependencies" version of the Spring Framework, this JAR can be found in the `lib/cglib` directory of the downloaded archive. (Of course, this dependency is automatically resolved if you use Maven.)

www.it-ebooks.info

Configuring Spring

Although Spring configuration is a bit out of scope for this book, in the next sections some basic information is provided to get you started should you have no experience with Spring. If you want to read more on Spring, the Spring project itself provides some excellent reference documentation on its website: `http://www.springsource.org/documentation`.

Letting Spring manage the beans



www.it-ebooks.info

```
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

In Spring's *applicationContext.xml* file, this will become:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
                 http://www.springframework.org/schema/beans
                 http://www.springframework.org/schema/beans/
                                        spring-beans-2.0.xsd>
 <bean id="bravenessCalc"
       class="inc.monsters.mias.backing.BravenessCalc"
       scope="request"/>
</beans>
```

Note that the subelements of the `<managed-bean>` element are replaced by attributes of the `<bean>` element. It is also good to realize that Spring also has a request scope, just like JSF. Now let's take a slightly more complicated example from the `faces-config.xml` file:

```
<managed-bean>
  <managed-bean-name>miasMenu</managed-bean-name>
  <managed-bean-class>
    org.apache.myfaces.trinidad.model.XMLMenuModel
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>source</property-name>
    <value>/WEB-INF/menu.xml</value>
  </managed-property>
</managed-bean>
```


www.it-ebooks.info

This definition of the bean that holds our menu structure also has managed properties. In Spring syntax, this will become:

```
<bean id="miasMenu"
      class="org.apache.myfaces.trinidad.model.XMLMenuModel"
      scope="request">
  <property name="source" value="/WEB-INF/menu.xml"/>
</bean>
```

Notice how the subelements of the `<managed-property>` element are again replaced by attributes in the Spring configuration file.

Spring can also handle more complicated properties, such as lists and maps. Take, for example, the bean that we use to store the user's preferences, which is defined in the `faces-config.xml` file as follows:

```
<managed-bean>
  <managed-bean-name>preferences</managed-bean-name>
  <managed-bean-class>inc.monsters.mias.Preferences
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>skinFamily</property-name>
    <property-class>java.lang.String</property-class>
    <value>mias</value>
  </managed-property>
  <managed-property>
    <property-name>availableSkinFamilies
    </property-name>
    <property-class>java.util.List</property-class>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>minimal</value>
      <value>suede</value>
      <value>mias</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

In Spring's *applicationContext.xml* file, this will become:

```
<bean id="preferences"
      class="inc.monsters.mias.Preferences"
      scope="session">
  <property name="skinFamily" value="mias"/>
  <property name="availableSkinFamilies">
```


www.it-ebooks.info

```
      <list>
        <value>minimal</value>
        <value>suede</value>
        <value>mias</value>
      </list>
    </property>
  </bean>
```

The injection of lists and list items is similar in both frameworks. It should be noted that in this case the session scope is used, as Spring has such a scope (just like JSF). On a side note, all of these examples show that Spring's bean definition syntax is generally more compact than the managed bean definition syntax of JSF.

One last example shows us some more differences between the two frameworks. In our `faces-config.xml`, we configured a `java.util.HashMap` as a managed bean containing the login names and passwords for the application as follows:

```
<managed-bean>
  <managed-bean-name>loginsmap</managed-bean-name>
  <managed-bean-class>java.util.HashMap
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <map-entries>
    <map-entry>
      <key>James.P.Sullivan</key>
      <value>n@viLLus</value>
    </map-entry>
    <map-entry>
      <key>Mike.Wazowski</key>
      <value>iksW0z@w</value>
    </map-entry>
  </map-entries>
  ...
</managed-bean>
<managed-bean>
  <managed-bean-name>loginBean</managed-bean-name>
  <managed-bean-class>inc.monsters.mias.LoginBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>logins</property-name>
    <value>#{loginsmap}</value>
  </managed-property>
</managed-bean>
```


www.it-ebooks.info

The `HashMap` is defined at application scope. It is then injected into the `loginBean` by the use of JSF Expression Language—`#{loginsmap}`. In the Spring XML file, we cannot use Expression Language. Spring does not have an application scope either, and the way we define a `Map` as a Spring bean is also slightly different. Let's see how we do these things in the *`applicationContext.xml`* file:

```
<bean id="loginsmap"
      class="org.springframework.beans.factory.config.MapFactoryBean"
      scope="singleton">
  <property name="sourceMap">
    <map>
      <entry key="James.P.Sullivan" value="n@viLLus"/>
      <entry key="Mike.Wazowski" value="iksW0z@w"/>
      <entry key="Celia.Mae" value="e@M"/>
      <entry key="Henry.J.Waternoose" value="es00nRet@w"/>
      <entry key="Abominable.Snowman" value="n@mw0nZ"/>
      <entry key="Randall.Boggs" value="sGG0b"/>
      <entry key="Roz" value="Z0r"/>
    </map>
  </property>
</bean>
<bean id="loginBean"
      class="inc.monsters.mias.LoginBean"
      scope="request">
  <property name="logins" ref="loginsmap"/>
</bean>
```

We have replaced the application scope by Spring's singleton scope. A singleton-scoped bean in Spring is a bean of which one and only one instance will be created in the whole application. That's pretty much like JSF's application scope. It is important to note that we didn't use `java.util.HashMap` as the class of our bean, but instead we used a `MapFactoryBean` from the Spring framework. That's because Spring doesn't have a `<map-entries>` element, like JSF. (Of course, JSF uses some kind of factory too, under the hood.) The last important thing in this example is how we inject the `loginsmap` bean into the `loginBean`. We use a `<property>` element. However, instead of a `value` attribute, we use a `ref` attribute this time that lets us refer to another bean in the Spring context.

www.it-ebooks.info

Configuring the faces-config.xml file for Spring

After creating all of the bean definitions in the *applicationContext.xml* file, we shouldn't forget to remove all managed bean definitions from the `faces-config.xml` file. Otherwise, we will end up with double bean definitions.

You might wonder how we can reference our beans in the JSF Expression Language, now that they're managed by Spring. JSF doesn't have any Spring support embedded, so we need to add that. For this reason, the Spring framework provides a special Expression Language resolver. The only thing we have to do is add that resolver to our `faces-config.xml` file, as shown in the following example:

```
<el-resolver>
     org.springframework.web.jsf.el.SpringBeanFacesELResolver
</el-resolver>
```

Configuring the web.xml file for Spring

To bootstrap Spring, we need to add some listeners to our `web.xml` configuration file. This is fairly straightforward:

```
<listener>
  <listener-class>
      org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<listener>
  <listener-class>
      org.springframework.web.context.request.RequestContextListener
  </listener-class>
</listener>
```

The first listener is responsible for the initial context loading. The second listener enables Spring to correctly initialize scopes, such as the request scope and the session scope.

www.it-ebooks.info

Configuring Spring and persistence



www.it-ebooks.info

Note that the last line uses a namespace that is not already in our `applicationContext.xml` file. This means that we have to add it at the top of the file. The first lines of the file thus become:

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema
                                                    /context"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans
                                    /spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx
                                    /spring-tx-2.0.xsd
...
```

Accessing the services

In the standard EJB approach that we used in the previous chapter, the services were injected into our backing beans by means of the `@EJB` annotation. As we're using Spring instead of EJB now, we cannot use this approach anymore. But because Spring is a **dependency injection** framework, it shouldn't be too difficult to have the services injected. We can use Spring's **auto wire** feature, which makes this very easy. Let's have a look at the `kidsTable` bean, which backs the `Kids.xhtml` page. We can declare that bean in the `applicationContext.xml` file, as follows:

```
<bean name="kidsTable"
      class="inc.monsters.mias.backing.KidsTable"
      scope="request"
      autowire="byName"/>
```

The `autowire` attribute is set to `byName`. This means that Spring will look for beans that can be injected based on the name of the property and the bean. So in case of the injection of a service bean into our `KidsTable` bean, we have to make sure that the name of the service bean corresponds to the name of the property in the `KidsTable` bean. So let's declare our kids service bean as follows, in the `applicationContext.xml` file:

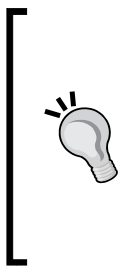
```
<bean name="kidService"
      class="inc.monsters.mias.data.facade.KidServiceBean"/>
```


www.it-ebooks.info

Now we have to make sure that the `KidsTable` backing bean has a `setKidService()` method, and Spring will do the rest for us:

```
public class KidsTable {
  private KidService kidService;
  ...
  public KidService getKidService() {
    return kidService;
  }
  public void setKidService(KidService kidService) {
    this.kidService = kidService;
  }
  ...
}
```

Downloading and installing Orchestra



www.it-ebooks.info

Configuring Orchestra

Now we can finally start configuring Orchestra. First, we have to add one extra listener to our `web.xml` file:

```
<listener>
  <listener-class>
        org.apache.myfaces.orchestra.conversation.servlet
                          .ConversationManagerSessionListener
  </listener-class>
</listener>
```

The rest of the Orchestra configuration is done in the Spring `applicationContext.xml` file. The first thing we have to do there is to include the default Orchestra initialization, as follows:

```
<import resource="classpath*:
                  /META-INF/spring-orchestra-init.xml" />
```

Note the `classpath*:` prefix—this causes Spring to search the whole classpath for the given file, so that it can be found inside the Orchestra JAR. One of the most important things to do is to configure at least one custom Orchestra scope. We'll define two scopes here, which are more-or-less standard. One scope is called `conversation.manual`, and the other is called `conversation.access`. The former can be used for conversations that are to be started and ended manually. The latter scope can be used for conversations that are ended automatically by Orchestra. (We will discuss these in more detail, later.) Let's see how we configure these scopes in the `applicationContext.xml` file:

```
<bean class="org.springframework.beans.factory.config
                                    .CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="conversation.manual">
        <bean class="org.apache.myfaces.orchestra
                              .conversation.spring
                              .SpringConversationScope">
          <property name="timeout" value="30" />
          <property name="advices">
            <list>
               <ref bean=
               "persistentContextConversationInterceptor"/>
            </list>
          </property>
        </bean>
      </entry>
      <entry key="conversation.access">
```


www.it-ebooks.info

```
            <bean class="org.apache.myfaces.orchestra
                                .conversation.spring
                                .SpringConversationScope">
              <property name="timeout" value="30" />
              <property name="advices">
                <list>
                  <ref bean=
                   "persistentContextConversationInterceptor"/>
                </list>
              </property>
              <property name="lifetime" value="access"/>
            </bean>
          </entry>
        </map>
      </property>
    </bean>
```

As you can see, this is, in fact, just another Spring bean definition. In this case, we have a `CustomScopeConfigurer` bean that gets a `Map` of scopes injected into it. That `Map`, in turn, contains two Spring beans that implement the scopes. Note that the only difference between the two scopes (apart from their name, of course) is that the `conversation.access` has an extra property of `lifetime`, which is set to `access`. Also notice the `persistentContextConversationInterceptor` bean that gets injected into both beans. You might have noticed that there is no bean with that name already, and that is correct. We have to define that bean:

```
<bean id="persistentContextConversationInterceptor"
      class="org.apache.myfaces.orchestra.conversation
        .spring.PersistenceContextConversationInterceptor">
  <property name="persistenceContextFactory"
            ref="persistentContextFactory"/>
</bean>
```

This leaves us with another bean, `persistentContextFactory`, that yet has to be defined. We define it as follows:

```
<bean id="persistentContextFactory"
      class="org.apache.myfaces.orchestra.conversation
                      .spring.JpaPersistenceContextFactory">
  <property name="entityManagerFactory"
            ref="entityManagerFactory"/>
</bean>
```


www.it-ebooks.info

As you can see, the `persistentContextFactory` needs a bean called `entityManagerFactory`. We already created that one when we were configuring Spring to work with our persistence unit.

Using the Orchestra ViewController

Orchestra promotes the use of the "one bean per page" paradigm. This often-used pattern means that every view (page) has only one bean associated with it. All of the server-side code that is specific to a certain page is put into the bean that is associated with that page. The Orchestra ViewController makes the association between the page and the bean a bit tighter, and has some convenient features. To define a bean as a ViewController for a specific page, the `@ViewController` annotation can be used. The next example shows how the `KidsTable` bean that is associated with the `Kids.xhtml` page is defined as being the ViewController of that `Kids.xhtml` page.

```
@ViewController(viewIds={"Kids.xhtml"})
public class KidsTable {
  ...
}
```

Note that the `viewIds` parameter is in plural form—a comma-separated list can be used to make this bean the ViewController of multiple views.

Using event methods

The Orchestra ViewController framework has the possibility to call a method on a ViewController bean for certain JSF events. Annotations can be used to mark a method in a ViewController bean as a handler for a certain event. The following table gives an overview of the available events:

Event	Annotation	Description
initView	`@InitView`	This event will be fired after the JSF `RESTORE_VIEW` phase. This is just after the HTTP request reaches the server. The initView event handler can be used to perform initialization in a ViewController bean.
preProcess	`@PreProcess`	This event will be fired before the JSF `INVOKE_APPLICATION` phase. At this point, all validation and conversion has finished, and any values submitted in the request are applied to the model. So in this event handler, the application's model can be used.

www.it-ebooks.info

Event	Annotation	Description



www.it-ebooks.info



www.it-ebooks.info

The controller class has a `kidService` and `employeeService` because both `Kid` and `Employee` objects have to be updated. The `selectedKid` member variable will contain the `Kid` object that the user has selected to edit. Now let's declare this class in the `applicationContext.xml` file, as follows:

```
<bean name="editKidController"
      class="inc.monsters.mias.controller.EditKidController"
      scope="conversation.manual"
      orchestra:conversationName="editKidConversation"
      autowire="byName"/>
```

The most important line here is `scope="conversation.manual"`. This will cause the bean to be put in the manual conversation scope. If the bean is referenced and no conversation exists, Orchestra will automatically create a new conversation. The `orchestra:conversationName` attribute gives the conversation a name, which is needed if we want to refer to this conversation later on. Also note the `autowire` setting. This will cause the `kidService` and `employeeService` member variables to be populated with the correct beans automatically, by Spring.

Now we have to edit our `EditKidForm` backing bean. The `selectedKid`, which is now contained in the `EditKidController`, was previously in this bean and should be removed. We also have to add a member variable and accessor methods for the `EditKidController`.

```
package inc.monsters.mias.backing;

// Imports hidden for brevity.

@ViewController(viewIds={"EditKid.xhtml"})
public class EditKidForm {

  private EditKidController editKidController;

  public EditKidController getEditKidController() {
    return editKidController;
  }

  public void setEditKidController(EditKidController ctrl) {
    this.editKidController = ctrl;
  }
}
```

Note that we have added a `@ViewController` annotation, just as we did before, for another backing bean. Some methods for specials functions, such as the braveness calculator, have been left out here for simplicity. Now let's verify the bean declaration of this bean in the `applicationContext.xml` file:

www.it-ebooks.info

```
<bean name="editKidForm"
      class="inc.monsters.mias.backing.EditKidForm"
      scope="request"
      autowire="byName"/>
```

The bean can now be put in the request scope without problems, as everything that has to live longer than a single request can be put in the `EditKidController`, which is in the conversation scope. The `autowire` setting will ensure that the `editKidController` member variable of `EditKidForm` will always have a valid reference to an `EditKidController`.

As the `selectedKid` variable is moved to another bean, we have to adapt all of the pages where this variable is referenced. Let's start with the `EditKid.xhtml` page, which is pretty straightforward. We simply have to replace every reference to `#{editKidForm.selectedKid}` by `#{editKidForm.editKidController.selectedKid}`. So, for example, the "first name" field becomes:

```
<mias:field id="firstName"
            bean="#{editKidForm.editKidController.selectedKid}"
            required="true"
            maximumLength="30"/>
```

In the `Kids.xhtml` page, the selected kid was previously put in the page flow scope. But now that we have a conversation scope, we do not need to use the page flow scope. Instead, let's just put the selected kid on the request scope. We'll add some code to the `EditKidController` later on in order to pull the selected kid off the request scope. In the `Kids.xhtml` page, we only need to change the `<tr:setActionListener>` component that is under the pencil icon:

```
<mias:column columnName="edit"
             headerName="emptyTableHeader"
             custom="true">
  <tr:commandLink action="edit"
                  immediate="true">
    <tr:image source="../images/pencil.png"
              inlineStyle="border-width: 0px;" />
    <tr:setActionListener from="#{kid}"
                          to="#{requestScope.selectedKid}" />
  </tr:commandLink>
</mias:column>
```


www.it-ebooks.info

We just changed the scope in the `to` attribute from `pageFlowScope` to `requestScope`, which is the only change necessary here. Now we have to add some code to our `EditKidController` to retrieve the selected kid from the request scope. We can use a **lazy initialization** pattern here, that is, we just change the `getSelectedKid()` method to this:

```
public Kid getSelectedKid() {
  if(null == selectedKid) {
    FacesContext ctx = FacesContext.getCurrentInstance();
    Kid k = (Kid) ctx.getExternalContext()
                        .getRequestMap().get("selectedKid");
    selectedKid = kidService.getKidById(k.getId());
  }
  return selectedKid;
}
```

This code ensures that the `Kid` object will only be retrieved from the request scope if the `selectedKid` member variable is empty. This will only be the case if the `EditKidController` object has just been created, which happens at the start of a new conversation.

Note that after getting the `Kid` object from the request scope, we re-retrieve the object from the `kidService` by its ID. The object that we get from the request scope is retrieved in another transaction that was open when the table with kids was filled. As that transaction is now closed, the `Kid` object that is in the request scope is now **detached**. This means that it is not managed by an EntityManager any more.

By re-retrieving the object from our `kidService` at the start of the conversation, we ensure that the `Kid` object we use in our conversation is managed by the conversation's EntityManager. (Orchestra guarantees that we're using a single EntityManager throughout the conversation.) In this way, Orchestra can monitor changes to the `Kid` object throughout the conversation, and can commit any changes to the database at the end of the conversation, if needed.

We've now finished the basic setup of our conversation. Note that the conversation will start automatically when any bean that is placed into the scope of the conversation is instantiated. In our case, there's only one bean in the scope of `editKidConversation`, and that is our `EditKidController`. This bean will be instantiated by Spring's `autowire` function if the `EditKidForm` bean is instantiated. That happens whenever the user navigates to the `EditKid.xhtml` page, which can be done by clicking on an edit link in the `Kids.xhtml` page.

www.it-ebooks.info

Extending the conversation

Now let's make the conversation more interesting by adding an extra screen to edit the "last scared" date of a kid, as described before. We start by creating a simple page with a couple of fields and call it `EditScared.xhtml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC ......>
<tr:document xmlns="http://www.w3.org/1999/xhtml" .....>
<ui:composition template="templates/template.xhtml">
<ui:define name="title”>Edit last scared</ui:define>
<ui:define name="content">
<tr:panelFormLayout>
<tr:group>
<mias:field id="firstName"
bean="#{editScaredForm.editKidController
.selectedKid}"
required="true"
readOnly="true"/>
<mias:field id="lastName"
bean="#{editScaredForm.editKidController
.selectedKid}"
required="true"
readOnly="true"/>
</tr:group>
<tr:group>
<mias:dateField id="lastScared"
bean="#{editScaredForm
.editKidController.selectedKid}"/>
<mias:selectField id="employee"
bean="#{editScaredForm
.editKidController.selectedKid}"
type="choice"
items="#{empsTable.employees}"
itemValue="id"
itemLabel="name" />
</tr:group>
<f:facet name="footer">
<tr:panelButtonBar halign="right">
<tr:commandButton text="#{msg.ok}"
actionListener=
"#{editScaredForm.editKidController.updateScareData}"
action="ok"/>
<tr:commandButton text="#{msg.cancel}"
action="cancel"
immediate="true" />
</tr:panelButtonBar>
```


www.it-ebooks.info

```
</f:facet>
</tr:panelFormLayout>
</ui:define></
ui:composition>
</tr:document>
```

Note that we use the same `selectedKid` object from the `editKidController` bean as is used in the `EditKid.xhtml` page. (The internal working of the components in the `mias` namespace, such as the `<mias:field>` component, is described in previous chapters and does not change.) The **OK** button has its `actionListener` attribute set to `#{editScaredForm.editKidController.updateScareData}`. Let's see how the `updateScareData()` method in the `EditKidController` class can be implemented:

```
@Transactional(propagation=Propagation.REQUIRED,
readOnly=false)
public void updateScareData(ActionEvent event) {
Kid k = getSelectedKid();
Employee e = employeeService
.getEmployeeById(k.getEmployee().getId());
e.increaseKidsScared(); }
```

Two important things can be noted about this method:

- It is prepended by a `@Transactional` annotation. This ensures that the Spring framework (and thus Orchestra) knows that this method can change data that might have to be committed to the database at the end of the transaction. The `propagation` and `readOnly` arguments are optional, and could have been left out in this case, as `Propagation.REQUIRED` and `false` are the respective default values.
- The `Employee` object is re-retrieved from the database for the same reason we re-retrieved the `Kid` object at the start of the conversation.

We also need to create a backing bean for the `EditScared.xhtml` page. Let's call it `EditScaredForm` to fit in with the naming scheme we used so far. The class only needs a member variable and accessors for the `EditKidController`:

```
package inc.monsters.mias.backing;

// Imports omitted for simplicity

@ViewController(viewIds={"EditScared.xhtml"})
@ConversationRequire(redirect="EditKid.xhtml",
conversationNames = { "editKidController" })

public class EditScaredForm {
private EditKidController editKidController;
```


www.it-ebooks.info

```
    public EditKidController getEditKidController() {

      return editKidController;
    }

    public void setEditKidController(EditKidController ctrl) {
      this.editKidController = ctrl;
    }
  }
```

The interesting thing about this class is the `@ConversationRequire` annotation. This Orchestra annotation ensures that a user can't use this page without a conversation. So if a user ever tries to navigate to this page directly, he or she will be redirected to the first page of the conversation. This first page is set by the `redirect` parameter of the annotation. The `conversationNames` parameter is a list of conversation names that are required for this page. In this case, there's only one conversation required, `"editKidConverstation"`, which is the name that we gave to the conversation in the `applicationContext.xml` file.

Ending the conversation

Of course, the final step is ending a conversation. If we had put our conversation in the `conversation.access` scope, instead of the `conversation.manual` scope, no special configuration was needed to end the conversation. In that case, Orchestra monitors the usage of the beans that are in the conversation scope. (In our case, this is the `editKidController` bean.) Whenever a page is loaded without referencing any bean in the scope of the conversation, Orchestra automatically ends the conversation. Although this may sound attractive, it also introduces a risk. You can think of different scenarios where a page does not reference the controller, but is in the middle of a conversation. As it is relatively simple to end a conversation manually, it might be worth the effort to do it manually. Ending a conversation manually also makes the end of the conversation clear in the code, which can be a benefit if someone else has to perform maintenance on the code later.

So let's see how we can end our "edit kid" conversation. First, let's see at which points we want to end the conversation:

- When the user clicks on the **OK** button in the `EditKid.xhtml` page, we want to end the conversation and commit the changed data to the database.
- When the user clicks on the **Cancel** button in the `EditKid.xhtml` page, the conversation must be ended without committing any changes.
- When the user clicks on the **Apply** button in the `EditKid.xhtml` page, the data should be saved, but the conversation should not be ended.

www.it-ebooks.info



www.it-ebooks.info

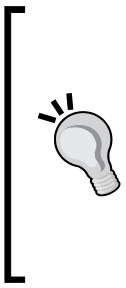


www.it-ebooks.info

Generating forms with DynaForm

Apart from the conversation stuff, Orchestra also offers an extra goodie: the `<ox:dynaForm>` component. As the name implies, this component dynamically generates a form. The component expects a JavaBean and creates a field for every property of the bean. It uses metadata from the bean, such as the type of properties and JPA annotations to decide what field is needed. The length of the field can be set, and read-only properties result in read-only fields in the form. Let's see how we can use this component.

Installing DynaForm



www.it-ebooks.info

Using DynaForm

To use the DynaForm component, not much extra work needs to be done. We can just add it to a page. The only thing we have to make sure is that we surround it with the correct layout component. If we want a simple data-entry style form, a `<h:panelGrid>` component is the appropriate choice. So let's create a new page, `EditEmployeeDyna.xhtml`, to duplicate the "edit employee" form in a dynamic way, as follows:

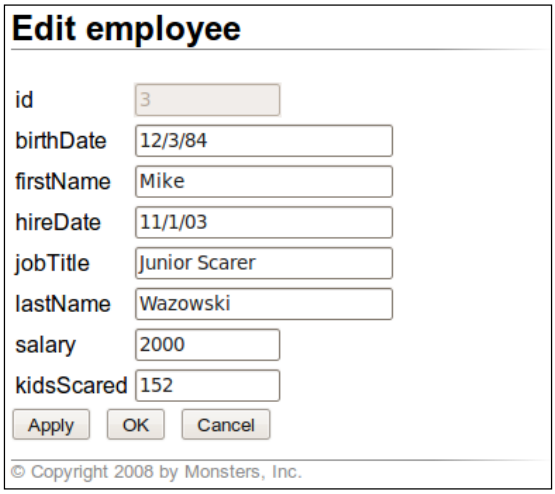
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC
 "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<tr:document xmlns="http://www.w3.org/1999/xhtml"
             xmlns:ui="http://java.sun.com/jsf/facelets"
             xmlns:h="http://java.sun.com/jsf/html"
             xmlns:f="http://java.sun.com/jsf/core"
             xmlns:tr="http://myfaces.apache.org/trinidad"
             xmlns:mias="http://www.monsters.inc/mias"
             xmlns:c="http://java.sun.com/jstl/core"
             xmlns:ox="http://myfaces.apache.org/orchestra/dynaForm">
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Edit employee - the dynamic way
  </ui:define>
  <ui:define name="content">
    <h:panelGrid id="employee-layout" columns="2">
      <ox:dynaForm id="employee"
                   uri="inc.monsters.mias.data.Employee"
                   valueBindingPrefix=
                                "pageFlowScope.selectedEmployee"/>
    </h:panelGrid>
    <tr:panelButtonBar>
    <!-- nothing changed compared to the original
         EditEmployee.xhtml page -->
    </tr:panelButtonBar>
  </ui:define>
</ui:composition>
</tr:document>
```


www.it-ebooks.info

Note the extra XML namespace that has been added at the top of the file. The `<ox:dynaForm>` component has a couple of attributes:

- `id`: This is nothing special. Just give the component a unique ID within the page.
- `uri`: The fully-classified name of the JavaBean that is going to be edited with the generated form.
- `valueBindingPrefix`: This is a JSF EL string (without the curly braces) that resolves to an instance of the Java class on which the form is based. In this example, we use an `Employee` object that is put on the `pageFlowScope` by the `Employees.xhtml` page. (We just left in place the mechanism that we used for our manually-created `EditEmployee.xhtml` page.)

The following image shows the form that is generated by the `<ox:dynaForm>` component, based on the properties of the `Employee` JPA bean:



Note that **id** field is read-only because the ID property of the `Employee` object is annotated with `@Id` and `@GeneratedValue`. This means that it is an immutable unique key, which will be generated by the database. Also note that the **salary** and **kidsScared** fields are smaller than the other fields. The buttons are not generated; we just copied them from the original page. You should realize that the Java code from the original backing bean is reused here for updating the persistent `Employee` object. In a typical use case where we have created a new page, we still have to write this backing bean code ourselves, as DynaForm doesn't generate it for us.

www.it-ebooks.info

Of course, much more can be said about the DynaForm component. But as long as it is not a part of the official Orchestra release, a lot of details can change, so it is simply too early to write a lot about the details of it in a book. Although the DynaForm component is a rather powerful component, it has its limitations. Apart from the unstable status, it has the same limitations that any automatic form generation tool has. Automatically-generated forms are probably very usable for prototypes and perhaps for maintenance screens in production applications. But for end-user screens, generally much more control is needed over the various details of the form. Most of the time, it is either not possible to control those details, or it requires just as much work as coding the form manually. Nevertheless, there are situations where a tool such as DynaForm can save a lot of work.

Summary

In this chapter, we learned how to add the Spring framework to a web project and how to change the project structure to enable Spring to replace an EJB container. We also learned how to set up Orchestra within a web project. We saw how we can configure conversations and how we can use this powerful concept in a web application. We took a quick look at Orchestra's DynaForm component, which looks promising. Throughout the chapter we saw that some problems that Orchestra solves are also solved in the specifications of the JSF 2.0 standard, but for the rest of the things we will still need Orchestra once we start using JSF 2.0.

In the next chapter, we'll explore the possibilities of MyFaces Extensions Validator—another framework that solves a common problem in Java EE 5 applications.

www.it-ebooks.info

www.it-ebooks.info

10
Extensions Validator

A common problem with the **Model-View-Controller pattern** (**MVC**) is that often the **Don't Repeat Yourself** (**DRY**) principle is violated when it comes to validation of data. The "single source of truth" with regards to validation is often either the Model layer or the underlying database. But to be able to give the user usable, easy-to-understand error messages, and to give those in a timely manner, we often need to repeat a lot of validation in the View layer.

This often leads to inconsistencies in applications when the validation code in the Model and View layers gets out of sync. This can happen because of a changed business rule that is implemented in the Model, but the View is not updated accordingly. Or if the View is redesigned, unintentional changes in the validation can occur. Even when Model and View are created at the same time but by different engineers, crippled communication between those engineers can lead to validation code that is out of sync.

Repeating validation logic in the View layer also breaks the DRY principle at another level. Often, information from a certain bean can be edited in different pages in the user interface. This means that the validation has to be repeated in all of those UI pages. So we can end up repeating the same validation logic in different pages.

Wouldn't it be better if we didn't have to repeat our validation code in the View layer while keeping usable error messages, and having the validation still taking place on the client side? This is the main reason that the "**Extensions Validator**" project was added to MyFaces. The word "Extensions" refers to the fact that this project is not about JSF components, but rather has to be seen as an extension to the JSF Framework. The idea is that more projects can be added in the future, as further extensions. However, for now, Validator is the only project under the "MyFaces Extensions" umbrella that has released any software yet. As "Extensions Validator" is a long name, the project is most of the time referred to as "**ExtVal**". We'll use this short name throughout this chapter.

www.it-ebooks.info

After reading this chapter, you will be able to:

- Set up a project to use ExtVal
- Use ExtVal to generate validation based on JPA annotations
- Use ExtVal's added annotations for additional validation
- Implement cross validation using ExtVal's annotations
- Use ExtVal with custom JSF validators
- Create custom error messages for ExtVal validations
- Override and extend ExtVal's default behavior
- Use Bean Validation (JSR 303) annotations in combination with ExtVal
- Use metadata to set the severity level of constraints

Setting up ExtVal

As with all other libraries, we start by downloading ExtVal and installing it in our project. As with many other JSF libraries, the ExtVal project has different branches for JSF 1.1 and 1.2. The first two digits of ExtVal's version number are the JSF version they are made for. So ExtVal 1.1.x is the xth version of ExtVal for JSF 1.1, whereas ExtVal 1.2.x is the xth version for JSF 1.2. Versions of ExtVal are not released very often. At the time of writing this book, only two official releases have been published for each branch. According to the lead developer of ExtVal, a third release (1.1.3 and 1.2.3) is in the works for both branches, as well as a first release from the new JSF 2.0 branch.

Apart from stable releases, ExtVal offers snapshot builds that are created on a regular basis. The snapshots are created manually, which gives some guarantees about the quality compared to automatically-created daily releases. No snapshots with major bugs will be created. According to the lead developer of ExtVal, the snapshot builds have "milestone quality".

Because of some issues and limitations in ExtVal 1.2.2, a snapshot build of ExtVal 1.2.3 was used while writing this chapter. A stable release of ExtVal 1.2.3 is expected to be available soon after the publishing date of this book. Stable releases can be downloaded from the ExtVal download site at `http://myfaces.apache.org/extensions/validator/download.html`. The downloaded ZIP file will contain all of the ExtVal modules, as listed in the next table. Note that more modules may be added to ExtVal in future releases. It is also possible that additional support modules will be provided by others. For example, a JSF component project may create a support module to get the most out of its components with ExtVal.

www.it-ebooks.info

Regarding component support modules, it is also worth mentioning the "Sandbox 890" project, which provides proof of concept implementations of support modules for some non-MyFaces component libraries. Currently, proofs of concept are available for IceFaces, PrimeFaces, RichFaces, and OpenFaces. The source code for the proofs of concept can be found at `http://code.google.com/p/sandbox890/source/browse/#svn/trunk/component-support`. Ready-to-use JARs can be downloaded from `http://code.google.com/p/os890-m2-repository/source/browse/#svn/trunk/sandbox890/sandbox890/extensions/validator/component-support-modules`.

Library	Description
`myfaces-extval-core-1.2.x.jar`	The core of ExtVal. This library should be added to the project in all cases.
`myfaces-extval-property-validation-1.2.x.jar`	Extension module that adds several custom ExtVal annotations that we can use in our Model layer.
`myfaces-extval-generic-support-1.2.x.jar`	Extension module for generic JSF component support. This library should be added to the project in almost all cases. There are two cases where we don't need this generic support library, which are as follows: • If we're using a support library for a specific component library, such as the Trinidad support module mentioned in the following row in this table • If the component library we're using is 100% compliant with the JSF specification, which is almost never the case If no specific support module is in use, and it is unclear if the generic module is needed, it is safe to add it anyway. It is also a good idea to take a look at the *Tested Compatibility* section on the ExtVal wiki, at `http://wiki.apache.org/myfaces/Extensions/Validator/`.
`myfaces-extval-trinidad-support-1.2.x.jar`	Extension module that supports the MyFaces Trinidad JSF components. If we use this one, we don't need the "generic support" module. The Trinidad support module will make use of Trinidad's client-side validation options where possible. So we get client-side validation based on annotations in our Model with no extra effort.

www.it-ebooks.info

Library	Description
`myfaces-extval-bean-validation-1.2.x.jar`	Extension module that adds support for Bean Validation (JSR 303) annotations. This module will be available from the third release of ExtVal (*.*.3). See the *Using Bean Validation* section at the end of this chapter.

Snapshot builds of ExtVal can be downloaded from ExtVal's Maven snapshot repository, which can be found at `http://people.apache.org/maven-snapshot-repository/org/apache/myfaces/extensions/validator/`. In the case of snapshot builds, no single ZIP file is available, and each module has to be downloaded separately as a JAR file. Note that if Maven is used, there is no need to manually download the snapshots. In that case, we only have to change the version number in the `pom.xml` file to a snapshot version number, and Maven will automatically download the latest snapshot. The following table lists the URLs within the Maven repository from where the modules can be downloaded:

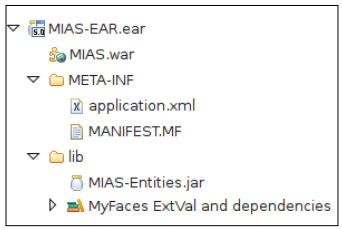
Module	URL
Core	`myfaces-extval-core/`
Property Validation	`validation-modules/myfaces-extval-property-validation/`
Generic Support	`component-support-modules/myfaces-extval-generic-support/`
Trinidad Support	`component-support-modules/myfaces-extval-trinidad-support/`
Bean Validation (JSR 303)	`validation-modules/myfaces-extval-bean-validation/`

URLs in this table are relative to the URL of the Maven repository that we just saw. After each URL, `1.2.x-SNAPSHOT/` has to be appended, where 1.2.x has to be replaced by the appropriate version number.

Once we've finished downloading, we can start adding the JARs to our project. ExtVal differs in one thing from other libraries—it needs to access our Model *and* View project. So we have to add the ExtVal libraries to the `lib` directory of the EAR, instead of the WAR or the JAR with the entities. Some libraries that ExtVal uses have to be moved there as well. If we don't do this, we'll end up with all sorts of weird exceptions related to class-loading errors.

Libraries that are added to the `lib` directory of an EAR are automatically available to all contained WAR and JAR files. However, depending on the IDE and build system that we are using, we may have to take some additional steps to be able to build the WAR and JAR with dependencies to the libraries in the EAR's `lib` directory.

www.it-ebooks.info



This image shows a simplified structure of the EAR with ExtVal's libraries added to it. Note that the **MyFaces ExtVal and dependencies** node in the image actually represents multiple JAR files. It is important to verify that none of the libraries that are in the `lib` directory of the EAR are included in either the WAR or the entities JAR. Otherwise, we could still encounter class-loading conflicts. The following table lists all of the libraries that have to be moved into the EAR to avoid these class-loading conflicts:

Library	Explanation
`myfaces-extval-*.jar`	Of course, all of the required ExtVal JARs should be in the EAR.
`asm-1.5.x.jar`, `cglib-2.x_y.jar`	These are libraries that ExtVal depends on. They are bundled with the ExtVal download. They're not bundled with snapshot releases.
`jsf-facelets.jar`	We're using Facelets, so ExtVal has to use it to add validations within our Facelets pages. So if we didn't use Facelets, this one would not be needed.
`myfaces-api-1.2.*`, `myfaces-impl-1.2.*`	We're using MyFaces Core as the JSF implementation. ExtVal will need these libs too. Note that if we use the application server's default JSF implementation, we don't have to add these either to the EAR or to the WAR.
`trinidad-api-1.2.*`, `trinidad-impl-1.2.*`	We're using Trinidad, and ExtVal offers some Trinidad-specific features through the "Trinidad support" extension. In this case, the Trinidad libraries should be in the EAR too.
`commons-*.jar`	Various libraries that we just mentioned depend on one or more libraries from the Apache Commons project. They should also be moved to the EAR file to be sure that no class-loading errors occur.



www.it-ebooks.info



www.it-ebooks.info

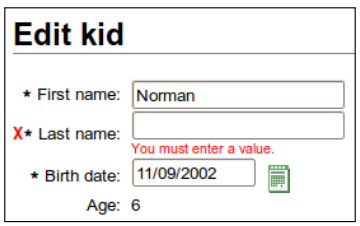
Basic usage

After setting up ExtVal, the basic usage is very simple. Let's explore a simple example in our MIAS application. In our `Kid.java` entity, we have some JPA annotations that map the properties of the `Kid` bean to a database table. Let's take a closer look at the `lastName` property of our `Kid` bean:

```
@Column(name = "LAST_NAME", nullable = false, length = 30)
private String lastName;
```

The `@Column` annotation maps the `lastName` property to the `LAST_NAME` column in the database. It also shows some information that is derived from the table definition in the database. `nullable = false` means the database won't accept an empty value in this field, and `length = 30` means that no more than 30 characters can be stored in the corresponding database column. This information could be used for validation in our View layer. If we hadn't used ExtVal, we would have added a `required="true"` attribute to the input element in our `EditKid.xhtml` page. We also would have added a `<tr:validateLength>` component to the input component, or we could have set the `maximumLength` attribute. But all of these things would have been a duplication of information and logic, and would thus break the DRY principle.

With ExtVal, we don't have to duplicate this information anymore. Whenever ExtVal encounters a `nullable = false` setting, it will automatically add a `required="true"` attribute to the corresponding input element. In the same way, it will translate the `length = 30` from the `@Column` annotation into a `maximumLength` attribute on the input component. The next screenshot shows ExtVal in action. (Note that all validators, and the `required` and `maximumLength` attributes were removed from the JSF code before the screenshot was taken.) The really nice thing about this example is that the validations created by ExtVal make use of Trinidad's client-side validation capabilities. In other words, the error message is created within the user's web browser before any input is sent to the server.



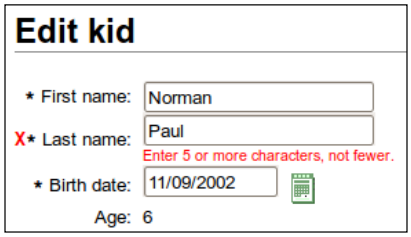
www.it-ebooks.info

Complementing JPA annotations

It's nice that we can reuse our JPA annotations for validation. But the chances are that not all validation that we want can be expressed in JPA annotations. For that reason, ExtVal offers a set of extra annotations that we can add to our beans to complement the implicit validation constraints that ExtVal derives from JPA annotations. These annotations are a part of the `myfaces-extval-property-validation-1.2.x.jar` library. For example, if we want to add a minimum length to the `lastName` field, we could use the `@Length` annotation as follows:

```
@Length(minimum = 5)
@Column(name = "LAST_NAME", nullable = false, length = 30)
private String lastName;
```

Note that if, for some reason, we couldn't use the `length = 30` setting on the `@Column` annotation, the `@Length` annotation also has a maximum property that can be set. The `@Length` annotation can be imported from the `org.apache.myfaces.extensions.validator.baseval.annotation` package, which is where the other annotations that ExtVal offers are also located. The following image shows the minimum length validation in action:



As the example in the screenshot shows, setting a minimum input length of five characters for a name might not be a good idea. However, that's an entirely different discussion.

Using ExtVal annotations for standard JSF validators



www.it-ebooks.info

Defining length validation

For the length validation of input strings, the `@Length` annotation can be used, as shown in the previous example. This annotation relies on the `javax.faces.validator.LengthValidator` to implement the validation. The following table lists the available properties:

Property	Type	Explanation
`minimum`	`int`	The minimum length (inclusive) in characters of the input string
`maximum`	`int`	The maximum length (inclusive) in characters of the input string

Defining double range validation

To validate if a double value is within a certain range, the `@DoubleRange` annotation can be used, which delegates the implementation of the validation to the `javax.faces.validator.DoubleRangeValidator` validator. See the following table for the available properties:

Property	Type	Explanation
`minimum`	`double`	The minimum value (inclusive) of the double input
`maximum`	`double`	The maximum value (inclusive) of the double input

Defining long range validation

What `@DoubleRange` annotation does for `doubles`, the `@LongRange` annotation does for `long` values. It uses `javax.faces.validator.LongRangeValidator` for the implementation:

Property	Type	Explanation
`minimum`	`long`	The minimum value (inclusive) of the long input
`maximum`	`long`	The maximum value (inclusive) of the long input

Defining required fields

The example given at the beginning of this section showed how ExtVal can create a `required="true"` attribute based on an `@Column` annotation with the `nullable = false` setting. If it is not possible to use this setting, ExtVal also has an alternative `@Required` annotation. Just add this annotation to a field to make it required.

www.it-ebooks.info

Using ExtVal's additional annotations

Apart from the annotations that correspond to the standard JSF validators, some additional annotations exist in the Property Validation module that perform other validations. These are listed in the following subsections.

Whereas the validations based on the JSF standard validators use the error messages provided by the JSF validators, the additional validations cannot use standard messages from JSF. Therefore, standard messages are provided by ExtVal. Should you want to use your own message, all additional annotations have a `validationErrorMsgKey` property that can be used to assign a message key for the error message. We'll discuss custom error messages in more detail later in this chapter .

Defining pattern-based validation

Validation with **regular expressions** is very powerful, and is very usable for validating phone numbers, postal codes, tax numbers, and any other data that has to fit in a certain pattern. Regular expression-based validation can be added by using the `@Pattern` annotation. For example, to allow only letters and spaces in the `firstName` field, we could write:

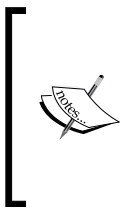
```
@Pattern(value="[A-Za-z ]*")
@Column(name = "FIRST_NAME", nullable = false, length = 30)
private String firstName;
```

For completeness, the following table lists the arguments of the `@Pattern` annotation:

Property	Type	Required	Explanation
`value`	`String[]`	Required	Array of regular expression patterns. Be aware that if you add more than one pattern, then the input must match all of the newly defined patterns. It is easy to make any input invalid by using two patterns that are mutually exclusive. In the *Creating our own validation strategy* section, we will see how we can change this behavior.
`validationErrorMsgKey`	`String`	Optional	Optional key for an alternative error message.

www.it-ebooks.info

Using custom validators



www.it-ebooks.info

The custom validator does not have to be configured in the `faces-config.xml` file as is normally the case with JSF validators. ExtVal will add the custom validator at runtime, when needed.

Now that we have prepared our custom validator, we can use it with the `@Validator` annotation in our entities. For example, in the `Kid` entity:

```
package inc.monsters.mias.data;

import inc.monsters.mias.validators.NoBartValidator;
...

public class Kid implements Serializable {

  ...

  @Column(name = "FIRST_NAME")
  @Validator(NoBartValidator.class)
  private String firstName;

  ...
}
```

Note that we have to import the `NoBartValidator` class, because it is referenced in the `@Validator` annotation in a typesafe way.

Reusing validation

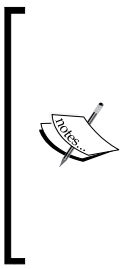
If multiple fields share the same validation rules, then according to the DRY principle, we should not repeat these validations for all of the fields. For this reason, the `@JoinValidation` annotation exists. As a simple example, we could reuse the validation rules of the `lastName` field for the `firstName` field, as shown here:

```
@Column(name = "FIRST_NAME")
@JoinValidation(value="lastName")
private String firstName;

@Length(minimum = 5)
@Column(name = "LAST_NAME", nullable = false, length = 30)
private String lastName;
```

In this example, all validation of the `lastName` field will be copied to the `firstName` field by ExtVal.

www.it-ebooks.info



www.it-ebooks.info

- **Static reference**: To overcome the problems of Expression Language reference, static reference can be used as an alternative. With static reference, a property is referenced by the fully-qualified name of its parent class, followed by the property name, separated by a colon. So to reference the `lastName` property of `Employee` bean, the complete annotation could look as follows: `@JoinValidation(value="inc.monsters.mias.data.Employee:lastName")`. Note that static reference is not available in the second release of ExtVal. So as long as there is no stable third release, we have to use a snapshot build of ExtVal 1.2.3 to be able to use static reference.

To summarize, the easiest way to reference another validation for reuse is, of course, **local reference**; but that's only possible for properties within the same bean. To refer to a property outside the current bean, property chain reference is a clean and easy solution in some cases, but it can't be used in all cases. For the cases where local or property chain reference can't be used, static reference is the best choice. Expression Language reference could be used as an alternative, but it comes at the price of breaking the MVC separation of concerns—a reason to avoid it if possible.

Applying cross validation



www.it-ebooks.info

Using cross validation for date values

Cross validation for dates is possible with the `@DateIs` annotation. The most important arguments for `@DateIs` are listed in the following table:

Argument	Required	Description
`type`	Optional	The type of date comparison that has to be executed. The possible values are: • `DateIsType.before`: The date to be validated has to be before the referenced date. • `DateIsType.after`: The date to be validated has to be after the referenced date. • `DateIsType.same`: The date to be validated has to be the same as the referenced date. This is the default setting. So if you leave out the `type` setting, the "same" comparison will be used.
`valueOf`	Required	The referenced date. This should be the name of a property that contains the date to which the entered date should be compared. More than one date can be referenced. If that is desired, the `valueOf` argument should be set to an array of `Strings`, where each `String` references one other value. See the information box in the *Applying Cross Validation* section for information about the available referencing strategies.
`notBeforeErrorMsgKey`	Optional	The key that will be used to look up the error message if a `DateIsType.before` rule is violated. The default key is `wrong_date_not_before`.
`notAfterErrorMsgKey`	Optional	The key that will be used to look up the error message if a `DateIsType.after` rule is violated. The default key is `wrong_date_not_after`.
`notEqualErrorMsgKey`	Optional	The key that will be used to look up the error message if a `DateIsType.same` rule is violated. The default key is `wrong_date_not_equal`.

www.it-ebooks.info

Argument	Required	Description
`errorMessageDateStyle`	Optional	The date format style for dates that are used within an error message. This should be one of the date style constants defined in `java.text.DateFormat`. The default is `DateFormat.MEDIUM`.
`validationErrorMsgKey`	Optional	Can be used to set the key to look up a more general error message. Is empty by default.

Using cross validation based on equality

Two annotations exist to check if two values are equal or not—`@Equals` and `@NotEquals`. The attributes for both annotations are the same, as listed in the following table:

Argument	Required	Description
`value`	Required	The value(s) to compare to. This can be a single `String`, or an array of `String`s if comparison to multiple values is required. See the information box in the *Applying Cross Validation* section for information about the available referencing strategies.
`validationErrorMsgKey`	Optional	Can be used to override the default key that is used to look up the error message in a message bundle in case the validation fails. The default value is: • `duplicated_content_required` for `@Equals` • `duplicated_content_denied` for `@NotEquals`

As an example, the `@Equals` annotation could be used to verify a "retype password" field:

```
@Equals(value = "password",
validationErrorMsgKey = "retype_password")
private String retypePassword;
```


www.it-ebooks.info

Making a value required conditionally

Another form of cross validation is making a field required only if one or more other fields are not empty. This can easily be accomplished with the `@RequiredIf` annotation. The arguments of the `@RequiredIf` annotation are shown in the following table:

Argument	Required	Description
`valueOf`	Required	The value(s) that should be checked to see if they are empty or not. This can be a single `String`, or an array of `String`s if comparison to multiple values is required. See the information box in the *Applying Cross Validation* section for information about the available referencing strategies.
`is`	Optional	One of `RequiredIfType.not_empty` or `RequiredIfType.empty`. If set to `RequiredIfType.empty`, the field will be required only if the referenced field is empty. The default is `RequiredIfType.not_empty`.
`validationErrorMsgKey`	Optional	Can be used to override the default key that is used to look up the error message in a message bundle, in case the validation fails. The default value is `empty_field`.

This section discussed the annotations that are a part of ExtVal's Property Validation module. From version *.*.3 onwards, ExtVal also offers a Bean Validation module that allows us to use annotations from the Bean Validation (JSR 303) standard, instead of, or in combination with, ExtVal's own annotations. See the *Using Bean Validation* section for more information. In the following section, we will see how we can customize all of the error messages generated by ExtVal.

Creating custom error messages

With ExtVal, the error messages shown if validation fails come from different sources, as follows:

- For validation that is derived from JPA annotations, ExtVal relies on the standard JSF validators. Hence, the error messages shown are the standard JSF error messages. The way in which standard JSF messages can be overridden is defined in the JSF standard. This is covered in the next section, *Overriding standard JSF error messages*.

www.it-ebooks.info

- The ExtVal annotations `@Length`, `@DoubleRange`, `@LongRange`, and `@Required` also rely on standard JSF mechanisms for implementing the validation. So these will lead to standard JSF error messages as well.
- All other ExtVal annotations have their own default error messages. How to override these ExtVal messages is covered in the *Overriding ExtVal default error messages* section.

Overriding standard JSF error messages

Although overriding standard JSF messages is not a feature of ExtVal, we cover it briefly here for convenience. Standard JSF error messages can be overridden by configuring a message bundle in the `faces-config.xml` file, and adding certain key/value pairs to that message bundle. In our MIAS application, we've configured a message bundle as follows:

```
<message-bundle>inc.monsters.mias.Messages</message-bundle>
```

This means that the JSF framework expects a file called `Messages.properties` to be present in the `inc/monsters/mias` directory. In that file, we can configure our custom messages. For example, to override the default message for required fields that are left empty, we could add the following to the file:

```
javax.faces.component.UIInput.REQUIRED =
                              Hey dude, this field is required!
```

The important thing here is the key—`javax.faces.component.UIInput.REQUIRED`. A list of all JSF error messages with their keys can be found in *Appendix D*. This appendix also shows the placeholders that can be used in the message texts. The placeholders will be replaced by the label of the input element that the message is related to, and examples of good values or maximum and minimum values where applicable.

Overriding ExtVal default error messages

ExtVal always looks in a fixed location for a message bundle; it doesn't care about the JSF message bundle configuration. To change a message, we can either put a message bundle in that default location, or we can tell ExtVal to look for the message bundle in another location. The default message bundle that ExtVal looks for is `validation_messages` in the `org.apache.myfaces.extensions.validator.custom` package. Of course, we could create that package within our application and put a `validation_messages.properties` file there. But wouldn't it be great if we could just use our application-wide message bundle? That's possible by telling ExtVal to look somewhere else for a message bundle. This is done by setting a context parameter in the `web.xml` file as follows:

www.it-ebooks.info

```
  <context-param>
    <param-name>
org.apache.myfaces.extensions.validator.CUSTOM_MESSAGE_BUNDLE
    </param-name>
    <param-value>inc.monsters.mias.Messages</param-value>
  </context-param>
```

Now we can put custom messages in our own `Messages.properties` file. To override the default error message for the `@Pattern` annotation, we could add:

```
no_match = Pattern not matched
```

But in this case, "pattern not matched" might be a bit too generic as an error message for end users. ExtVal lets us override the error message on a per-field basis, allowing us to define more specific error messages. For example, we could have a `firstName` field with a pattern that allows only letters. Now we would like to have a message saying that only letters are allowed in names. In that case, we could write:

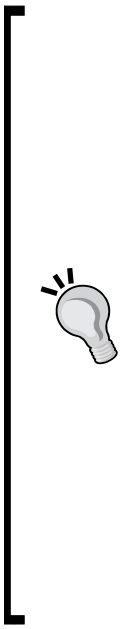
```
@Column(name = "FIRST_NAME")
@Pattern(value="[A-Za-z]*"
         validationErrorMsgKey="name_characters")
private String firstName;
```

Now if we add a `name_characters` key to our message bundle, we can set our customized, field-specific message:

```
name_characters = A name may only contain letters
```

In case we want to override the default ExtVal messages, a list of the default messages and their keys can be found in *Appendix E*.

www.it-ebooks.info



www.it-ebooks.info

Although combining two regular expressions with an "and" relation might be useful sometimes, having multiple expressions where only one of them has to be matched can be quite powerful too. We can think of a list of patterns for various (international) phone number formats. The input would be valid if one of the patterns is matched. The same can be done for postal codes, social security codes, and so on. So let's see how we can change the behavior of ExtVal to achieve this.

Implementing a custom validation strategy

ExtVal uses the concept of **Validation Strategy** for every type of validation. So, if an `@Pattern` annotation is used, ExtVal will use a `PatternStrategy` to execute the validation. We can implement our own `ValidationStrategy` to override the functionality of ExtVal's standard `PatternStrategy`. The easiest way to do this is to create a subclass of `AbstractAnnotationValidationStrategy<Pattern>`:

```
package inc.monsters.mias.extval;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

import org.apache.myfaces.extensions.validator.baseval
                        .annotation.Pattern;
import org.apache.myfaces.extensions.validator.core
                        .metadata.MetaDataEntry;
import org.apache.myfaces.extensions.validator.core
                        .validation.strategy
                        .AbstractAnnotationValidationStrategy;

public class PatternOrValidationStrategy extends
                        AbstractAnnotationValidationStrategy<Pattern> {

  @Override
  protected String getValidationErrorMsgKey(Pattern annotation) {
    return annotation.validationErrorMsgKey();
  }

  @Override
  protected void processValidation(
                  FacesContext facesContext,
                  UIComponent uiComponent,
                  MetaDataEntry metaDataEntry,
                  Object convertedObject) throws ValidatorException  {

    Pattern annotation = metaDataEntry.getValue(Pattern.class);
```


www.it-ebooks.info

```
        boolean matched = false;
        String expressions = null;

        for (String expression : annotation.value()) {
          if (convertedObject != null &&
               java.util.regex.Pattern.compile(expression)
                .matcher(convertedObject.toString()).matches()) {
            matched = true;
            break;
          } else {
            if (expressions == null) {
              expressions = expression;
            } else {
              expressions += ", " + expression;
            }
          }
        }

        if(!matched) {
          FacesMessage fm = new FacesMessage(
                            FacesMessage.SEVERITY_ERROR,
                            getErrorMessageSummary(annotation),
                            getErrorMessageDetail(annotation)
                                     .replace("{0}",expressions))

          throw new ValidatorException(fm);
        }
      }
    }
```

The most important part of this class is, of course, the `processValidation()` method. This uses the `MetaDataEntry` object to access the annotation that defines the validation. By calling `annotation.value()`, the array of `String`s that was set in the `@Pattern` annotation's `value` attribute is obtained. By iterating over that array, the user input (`convertedObject.toString()`) is matched against each of the patterns. If one of the patterns matches the input, the `boolean` variable `matched` is set to `true` and the iteration is stopped. A `ValidatorException` is thrown if none of the patterns matches the input. The `else` branch of the outer `if` statement is used to create a list of patterns that didn't match. That list is appended to the error message if none of the patterns matches.

Now that we've created our own custom validation strategy, we will have to tell ExtVal to use that instead of the default strategy for the `@Pattern` annotation. The next section shows how to do that.

www.it-ebooks.info

Configuring ExtVal to use a custom validation strategy

The most straightforward way to configure a custom Validation Strategy in ExtVal is to write a custom **Startup Listener** that will add our Validation Strategy to the ExtVal configuration. A Startup Listener is just a JSF `PhaseListener` with some specific ExtVal functionality—it deregisters itself after being executed, thus guaranteeing that it will be executed only once. We can simply subclass ExtVal's `AbstractStartupListener`. That way, we don't have to implement much ourselves:

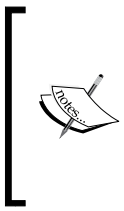
```
package inc.monsters.mias.extval;

import org.apache.myfaces.extensions.validator
                  .baseval.annotation.Pattern;
import org.apache.myfaces.extensions.validator
                  .core.ExtValContext;
import org.apache.myfaces.extensions.validator
                  .core.initializer
                  .configuration.StaticConfigurationNames;
import org.apache.myfaces.extensions.validator
                  .core.initializer
                  .configuration.StaticInMemoryConfiguration;
import org.apache.myfaces.extensions.validator
                  .core.startup.AbstractStartupListener;

public class PatternOrStartupListener
                           extends AbstractStartupListener {

  @Override
  protected void init() {
    // 1.
    StaticInMemoryConfiguration config
                        = new StaticInMemoryConfiguration();
    // 2.
    config.addMapping(
               Pattern.class.getName(),
               PatternOrValidationStrategy.class.getName());
    // 3.
    ExtValContext.getContext().addStaticConfiguration(
               StaticConfigurationNames
                    .META_DATA_TO_VALIDATION_STRATEGY_CONFIG,
               config);
  }
}
```


www.it-ebooks.info



www.it-ebooks.info

- One alternative is the **annotation-based configuration**. In this case, custom implementations can be annotated with special annotations, and should be put in a special base package. At application startup, the base package will be scanned for annotations, and the found annotations will be used to create the necessary configuration. See the *Extending ExtVal with add-ons* section for the download location, and installation instructions for this add-on. Some basic usage documentation is provided at `http://os890.blogspot.com/2008/10/myfaces-extval-config-extension.html`.
- The other alternative way to configure ExtVal is to use Java in a way that is inspired by the way Google Guice does this sort of things. In this case, a custom startup listener has to be created in which the Google Guice style code can be executed. Basic usage information can be found at `http://os890.blogspot.com/2009/09/myfaces-extval-java-config-extension.html`. See the *Extending ExtVal with add-ons* section for the download location and installation instructions.

Testing the custom validation strategy

Now that we've implemented our custom Validation Strategy, let's do a simple test. For example, we could add the `@Pattern` annotation to the `firstName` property of the `Kid` class, as follows:

```
@Column(name = "FIRST_NAME")
@Pattern(value={"[A-Za-z]*", "[0-9]*"})
private String firstName;
```

In this case, "Shirley" would be valid input, as would be "4623". But "Shirley7" wouldn't be valid, as none of the regular expressions allow both letters and digits. If we had used the default `PatternStrategy`, no valid input for the `firstName` field would be possible, as the regular expressions in this example exclude each other.

Of course this test case is not very useful. As mentioned before, having different patterns where only one of them has to be matched can be very useful for different (international) phone number formats, postal codes, social security codes, and so on. The example here is kept simple in order to make it easy to understand what input will match and what input won't match.

www.it-ebooks.info

Extending ExtVal in many other ways

Implementing a custom Validation Strategy is just one example of the many concepts in ExtVal that can be overridden by implementing a custom subclass, albeit one of the most useful ones. Here's a list of other concepts in ExtVal that can be overridden:

- `StartupListener` can be used to perform various actions at startup, such as registering any overridden ExtVal class. See the example in the *Configuring ExtVal to use a custom Validation Strategy* section.
- `ValidationStrategy` can be used to customize the validation behavior, as discussed in the previous section. The easiest way to implement this interface is to subclass one of the abstract classes provided by ExtVal.
- `MessageResolver` can be used to customize the error messages.
- `ComponentInitializer` allows the initialization of components before they are rendered. This can be used, for example, to add special client-side validation behavior to components.
- `MetaDataTransformer` transforms constraints to an independent format so a component initializer doesn't have any knowledge about the annotation used. A detailed explanation of this mechanism can be found in the *Empower the Client* section of an article about ExtVal on JSF Central, at `http://jsfcentral.com/articles/myfaces_extval_3.html`.
- `MetaDataExtractionInterceptor` allows on-the-fly manipulation of metadata.
- `InformationProviderBean` makes it possible to customize name conventions.
- `ProcessedInformationRecorder` can be used to capture values after they are converted by JSF. For example, the ExtVals implementation of cross validation is based on this mechanism.
- `RendererInterceptor` is one of the base mechanisms of ExtVal that is used to intercept renderers. All methods of `javax.faces.render.Renderer` can be intercepted.
- `NameMapper` is used extensively throughout the ExtVal framework, in order to map sources to targets. In most cases, names are mapped; for example, annotation names are mapped to validation strategy names.
- ExtVal makes extensive use of the **Factory design pattern,** and comes with a lot of factories that can be used when extending ExtVal. It is also possible to override the default factories as a way of changing ExtVal's behavior. An overview of all of the factories can be found in the `org.apache.myfaces.extensions.validator.core.factory.FactoryNames` class in the ExtVal sources.

www.it-ebooks.info

This list can be used as a starting point for exploring the extension opportunities in ExtVal. Some more information can be found on the ExtVal wiki at `http://wiki.apache.org/myfaces/Extensions/Validator/DevDoc` and `http://wiki.apache.org/myfaces/Extensions/Validator/ConceptOverview`.

Extending ExtVal with add-ons

ExtVal is a very flexible framework that was built with the possibility to extend it in mind. As we saw in the previous section, the ExtVal framework is full of hooks that can be a starting point for extending it. Of course, because it is an open source framework, anyone has the opportunity to extend and modify the framework to fit his needs. This can be a challenging job, even for advanced programmers. Everyone who has the time and knowledge should be encouraged to do so, as they can help in improving and expanding the ExtVal framework, or any other part of MyFaces. However, this section will focus on an easier way to expand the possibilities of the ExtVal framework: by using add-ons.

Getting add-ons for ExtVal

As ExtVal is a relatively new project and is not yet widely used, there are currently no "third party" open source add-ons for ExtVal. However, the lead developer of the ExtVal project has created some very useful add-ons. The following tables give an overview of the ExtVal add-ons that are available at the time of writing of this chapter. Keep an eye on the weblog of ExtVal's lead developer for the latest news about add-ons—`http://os890.blogspot.com/`.

Add-on name	Annotation-based configuration.
Description	Change ExtVal's defaults by using annotations instead of creating a startup listener. See the description in the *Using alternative configuration add-ons* section.
Documentation	`http://os890.blogspot.com/2008/10/myfaces-extval-config-extension.html`.
Download	`http://os890-m2-repository.googlecode.com/svn/tags/os890/at/gp/web/jsf/extval/extval-annotation-based-config-core/`.

www.it-ebooks.info

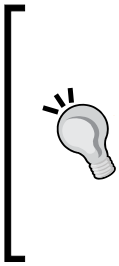
Add-on name	Google Guice style configuration.
Description	Change ExtVal's defaults by using Google Guice style configuration code. See the description in the *Using alternative configuration add-ons* section.
Documentation	`http://os890.blogspot.com/2009/09/myfaces-extval-java-config-extension.html`.
Download	`http://os890-m2-repository.googlecode.com/svn/tags/os890/at/gp/web/jsf/extval/extval-java-based-config-core/`.

Add-on name	Advanced metadata.
Description	This is actually a collection of four add-ons that have something to do with constraint metadata. This add-on can be used to: • Conditionally exclude constraints from validation. • Force priorities for certain constraints by adding special metadata. • Add "virtual" metadata to non-ExtVal constraints. An example of the usage of this add-on can be found in the *Setting the severity level on any constraint* section at the end of this chapter. • Separate the metadata from the main entity class.
Documentation	• About the collection of plugins: `http://os890.blogspot.com/2009/06/myfaces-extval-add-on-advanced-metadata.html`. • About conditional metadata exclusion: `http://os890.blogspot.com/2009/06/myfaces-extval-add-on-conditional.html`. • About virtual metadata: `http://os890.blogspot.com/2009/06/myfaces-extval-add-on-virtual-metadata.html`. • About metadata priority: `http://os890.blogspot.com/2009/06/myfaces-extval-add-on-metadata-priority.html`. • About the metadata provider: `http://os890.blogspot.com/2009/06/myfaces-extval-add-on-metadata-provider.html`.
Download	`http://os890-m2-repository.googlecode.com/svn/trunk/os890/at/gp/web/jsf/extval/extval-advanced-metadata/`.

www.it-ebooks.info

Add-on name	Secured Action
Description	This add-on shows that ExtVal can be used for things that go beyond input validation. It can be used to annotate JSF action methods, so that they can only be executed if certain security rules are met.
Documentation	`http://os890.blogspot.com/2009/04/myfaces-extval-add-on-securedaction.html`.
Download	`http://os890-m2-repository.googlecode.com/svn/tags/os890/at/gp/web/jsf/extval/extval-secure-actions/`.

Add-on name	Continue with warnings.
Description	This add-on allows us to give certain constraint violations a "warning" severity level, and give the user the possibility to ignore the warning. The use of this add-on is discussed in the *Using payloads to set severity levels* section.
Documentation	The *Using payloads to set severity levels* section and `http://os890.blogspot.com/2009/10/add-on-customizable-severity-feature.html`.
Download	`http://os890-m2-repository.googlecode.com/svn/trunk/os890/at/gp/web/jsf/extval/extval-continue-with-warnings/`.

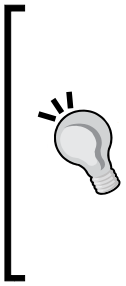


www.it-ebooks.info

Installing ExtVal add-ons

Installing an ExtVal add-on is simple. We only have to add the downloaded JAR to the shared `lib` directory in which all other ExtVal JARs are located, in our EAR file. If we have an application that is deployed as a single WAR file, we could simply add the JARs to the deployed libraries of that WAR file. This is all there is to do to install an ExtVal add-on. Note that each add-on has its specific usage instructions. Refer to the documentation that is linked to in the tables in the previous section.

Using Bean Validation



www.it-ebooks.info

Setting up Bean Validation and ExtVal

To use Bean Validation, we need a JSR 303 implementation, unless we're using a Java EE 6 compliant application server. Currently, the only available JSR 303 implementation is the reference implementation, which is **Hibernate Validator** 4.0. Hibernate Validator can be downloaded from `https://www.hibernate.org/30.html`. We should make sure we download a 4.0 version, as versions before 4.0 do not implement the JSR 303 standard. At the time of writing this chapter, the latest release is 4.0.2 GA.

After downloading Hibernate Validator, we have to add the Bean Validation libraries to our project. As described in the *Setting up ExtVal* section at the beginning of this chapter, all libraries have to be in the shared `lib` directory of our EAR. We also have to add the libraries that Hibernate Validator depends on. The following table shows a list of libraries that have to be added to our project in order to be able to use the Hibernate Validator. If we had used Maven, these libraries would have been downloaded and added to our project automatically by Maven.

Library	Description	Where to get
`hibernate-validator-4.0.2.GA.jar`	The main Hibernate Validator library.	Included in the root directory of the Hibernate Validator distribution.
`validation-api-1.0.0.GA.jar`	Contains all interfaces and annotations defined by the JSR 303 standard.	Included in the `lib` directory of the Hibernate Validator distribution.
`slf4j-log4j12-1.5.6.jar`, `slf4j-api-1.5.6.jar`, `log4j-1.2.14.jar`, `jpa-api-2.0.Beta-20090815.jar`	Runtime dependencies of Hibernate Validator.	Included in the `lib` directory of the Hibernate Validator distribution.
`activation-1.1.jar`, `jaxb-api-2.1.jar`, `jaxb-impl-2.1.3.jar`, `stax-api-1.0-2.jar`	Runtime dependencies for Hibernate Validator. These libraries are only needed if we run Hibernate Validator on a JDK 5 version. So even if we use a Java EE 5 server that runs on a JDK 6 version, we don't need these libs.	Included in the `lib/jdk5` directory of the Hibernate Validator distribution.

Once we have added the Bean Validation libraries to our project, we have to make sure that we have also added ExtVal's Bean Validation module to our project. The Bean Validation module is only available from ExtVal version 1.2.3 onwards. See the *Setting up ExtVal* section for more details.

www.it-ebooks.info

Using Bean Validation annotations

The basic usage of Bean Validation is very similar to the use of ExtVal's Property Validation annotations. There are some differences in the annotations, though. The following table lists all of the annotations that are defined in the Bean Validation specification:

Annotation	Attributes	Description
`@AssertFalse`		Assure that the element that is annotated is `false`.
`@AssertTrue`		Assure that the element that is annotated is `true`.
`@DecimalMin`	`value`	The value of the annotated element must be a numeric value greater than or equal to the indicated value. The `value` attribute must be a `String` that will be interpreted as a `BigDecimal` string representation.
`@DecimalMax`	`value`	The value of the annotated element must be a numeric value less than or equal to the indicated value. The `value` attribute has the same behavior as the `value` attribute of the `@DecimalMin` annotation.
`@Digits`	`integer, fraction`	The annotated element must have a numeric value that can't have more integer digits and fraction digits than indicated by the `integer` and `fraction` attributes.
`@Past`		Can be applied to `java.util.Date` and `java.util.Calendar` elements. The value of the annotated element must be in the past.
`@Future`		Can be applied to `java.util.Date` and `java.util.Calendar` elements. The value of the annotated element must be in the future.
`@Min`	`value`	Only for integer values. The value of the annotated element must be greater than or equal to the given value.
`@Max`	`value`	Only for integer values. The value of the annotated element must be less than or equal to the given value.

www.it-ebooks.info

Annotation	Attributes	Description
`@NotNull`		The annotated value can't be `null`.
`@Null`		The annotated value must be `null`.
`@Pattern`	`regexp, flags`	Can only be applied to `Strings`. The annotated `String` must match the regular expression that is given in the `regexp` attribute. The flags attribute can be set to an array of `Pattern.Flag` values, indicating which flags should be set to the `java.util.regex.Pattern` that will be used to match the value against. Valid flags are `UNIX_LINES`, `CASE_INSENSITIVE`, `COMMENTS`, `MULTILINE`, `DOTALL`, `UNICODE_CASE`, and `CANON_EQ`. See the JavaDoc documentation for `java.util.regex.Pattern` for an explanation of the flags (`http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html`).
`@Size`	`min, max`	Can be applied to `Strings`, `Collections`, `Maps`, and arrays. Verifies that the size of the annotated element is between the given `min` and `max` values, `min` and `max` values included.
`@Valid`		For recursive validation. See the *Using recursive validation* subsection for further explanation.

All annotations are defined in the `javax.validation.constraints` package. Apart from the attributes mentioned in the previous table, all annotations (except the `@Valid` annotation) have the following common attributes:

- `message`: This attribute can be used to set a custom error message that will be displayed if the constraint defined by the annotation is not met. If we want to set a message bundle key instead of a literal message, we should surround it with braces. So we can set `message` to either `"This value is not valid"` or `"{inc.monsters.mias.not_valid}"`.

www.it-ebooks.info

- `groups`: This attribute can be used to associate a constraint with one or more **validation processing groups**. Validation processing groups can be used to influence the order in which constraints get validated, or to validate a bean only partially. (See `http://docs.jboss.org/hibernate/stable/validator/reference/en/html/validator-usingvalidator.html#validator-usingvalidator-validationgroups` for more on validation groups.)
- `payload`: This attribute can be used to attach extra meta information to a constraint. The Bean Validation standard does not define any standard metadata that can be used, but specific libraries can define their own metadata. This mechanism can be used with ExtVal to add severity information to constraints, enabling the JSF pages to show certain constraint violations as warnings instead of errors. See the *Using payloads to set severity levels* section for an example of this.

OK, now we know which annotations can be used. Let's see how we can use Bean Validation annotations on our `Employee` class:

```
// Package declaration and imports omitted for brevity
public class Employee implements Serializable {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int id;
@Temporal(TemporalType.DATE)
@Column(name="BIRTH_DATE")
@Past
private Date birthDate;
@Column(name="FIRST_NAME")
private String firstName;

@Temporal(TemporalType.DATE)
@Column(name="HIRE_DATE")
@Past
private Date hireDate;
@Column(name="JOB_TITLE")
@NotNull
@Size(min=1)
private String jobTitle;
@Column(name="LAST_NAME")
private String lastName;

@Min(value=100)
private int salary;
```


www.it-ebooks.info

```
    @Column(name="KIDS_SCARED")
    private int kidsScared;

    @OneToMany(mappedBy="employee")
    private List<Kid> kids;

  // Getters and setters and other code omitted.
}
```

The Bean Validation annotations are highlighted in the code example. Note that the annotations are applied to the member variables here. Alternatively, we could have applied them to the getter methods. The JPA annotations that we added in Chapter 8 are still present. In this example, the `birthDate` and `hireDate` are annotated with `@Past` so that only dates in the past can be set. The `jobTitle` is set to have a minimum length of one character by the `@Size` annotation. The `salary` must have a minimum value of 100, as set by the `@Min` annotation.

Reusing validation

Bean Validation does not have a solution like the `@JoinValidation` annotation of ExtVal's Property Validation module. However, Bean Validation offers other ways to avoid repetitive code and help us reusing validation. This section describes some of the possibilities.

Inheriting validation

Constraints defined on (the properties of) super classes are inherited. This means that if we have a super class called `Person`, like the following example, our `Employee` class can inherit the properties—including the annotated constraints—as follows:

```
public class Person {
  @Size(min=1)
  private String firstName;

  @Size(min=1)
  private String lastName;

  @Past
  private Date birthDate;

  // Getters and setters omitted.
}
```

No special actions have to be taken to inherit annotated validation constraints.

www.it-ebooks.info

Using recursive validation

We can use the `@Valid` annotation to use **recursive validation** (or **graph validation** as it is called in the JSR 303 specification). The `@Valid` annotation can be used on single member objects as well as on `Collection`s. If applied to a `Collection`, all objects in the collection are validated, but `null` values in the `Collection` are ignored. For example, we could use this to validate the `List` of scared `Kid`s that is part of our `Employee` class, as follows:

```
public class Employee implements Serializable {

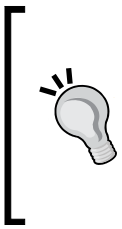
   // Other member variables are left out here.

   @OneToMany(mappedBy="employee")
   @Valid
   private List<Kid> kids;

   // Getters and setters are omitted.
}
```

Now the `List` of `Kid`s that is referenced by the `kids` variable can only contain valid `Kid` objects. This means that all Bean Validation constraints that are defined on the `Kid` class will be checked on all `Kid` objects in the `List`.

Composing custom constraints



www.it-ebooks.info

Defining a custom constraint involves creating a new annotation. This may look a bit complicated at first, but it is less complicated than it seems. Let's see how we can create an `@Name` annotation that we can use on all names in our project:

```
package inc.monsters.mias.data.validation;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.OverridesAttribute;
import java.lang.annotation.Retention; im-
port java.lang.annotation.Target;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@NotNull
@Size(min = 2)

@Constraint(validatedBy = {})
@Retention(RUNTIME)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
public @interface Name {

String message() default
"{inc.monsters.mias.data.validation.Name.invalid_name}";
Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

@OverridesAttribute(constraint = Size.class, name = "max")
int maxLength() default 20;
}
```

In this example, the following interesting things can be observed:

- `public @interface Name`: This defines a new annotation—`@Name`.
- `@Target`: This defines on what elements the new `@Name` annotation can be used. In this case, it can be used on methods, fields, and other annotations. Most of the time, this is fine for new constraints. It is also possible to create constraints that validate a class; in that case, `TYPE` should be used as the target. (See `http://docs.jboss.org/hibernate/stable/validator/reference/en/html/validator-usingvalidator.html#d0e328` for more information on class-level constraints.)

www.it-ebooks.info

- `@Retention`: This defines that this annotation should be executed at runtime. For validation constraints, this should always be set to `RUNTIME`.
- `@Constraint`: This identifies this annotation as being a validation constraint. It should always be used for custom constraints.
- `@NotNull` and `@Size(min=2)` (right above the `@Constraint` annotation, highlighted): These are the constraints that the `@Name` constraint is based on. In other words, any element annotated with `@Name` must not be `null` and must have a size of at least 2.
- `int maxLength() default 20`: This defines an attribute `maxLength` for the `@Name` annotation, with a default value of 20. So if no `maxLength` is specified, the `maxLength` will be 20.
- `@OverridesAttribute(constraint = Size.class, name = "max")`: This causes the `maxLength` attribute to override the `max` attribute of the `Size` annotation.
- `String message() default "{...}"`: This sets the default value of the `message` attribute to a message bundle key.

Other code that is not mentioned in the bulleted list is needed for every constraint definition. We now have an `@Name` annotation that can be used on any name field in our project. The annotated field cannot be empty, and should have a size of at least 2, and at most 20. The maximum size can be overridden by the `maxLength` attribute. We can use it, for example, in our `Employee` class, as follows:

```
public class Employee implements Serializable {

// Other member variable are omitted.

@Column(name="FIRST_NAME")
@Name
private String firstName;

@Column(name="LAST_NAME")
@Name(maxLength = 40)
private String lastName;

// Getters and setters are omitted.

}
```


www.it-ebooks.info

Now, the `firstName` can't be null. It must have at least 2 characters and at most 20 characters. The `lastName` has the same constraints, but can be up to 40 characters long. Note how we have reached the same level of reuse as we did when we used `@JoinValidation` in our `Kid` class earlier in this chapter. Creating our own custom constraint may be a little more work, but it gives us a more structural way of reuse. And we don't get referencing problems, as we did with `@JoinValidation`. As a bonus, we can reuse custom constraints over different projects. We can even create a library of custom constraints to be used in several projects.

As an example of the flexibility and extendability of ExtVal, the next section will show us how we can set severity levels on certain constraints that give the users the possibility to ignore certain warnings.

Using payloads to set severity levels

As mentioned, we can use the `payload` attribute on every Bean Validation annotation to pass on meta information about the constraint. With ExtVal, we can use this to create warning messages for certain constraints. These warning messages will appear the first time the user submits a value that violates the constraint. The user can either change the value or ignore the warning by submitting the value for the second time. This section describes how we can implement this for the `salary` field of our `Employee` class.

Setting up the Continue with warnings add-on

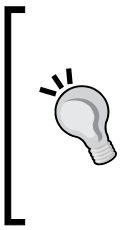
Let's start by downloading and installing the **Continue with warnings** add-on, as described in the *Extending ExtVal with add-ons* section. Once we've downloaded the JAR file and added it to our project, we can start preparing our project to allow the users to ignore warnings. The first thing we'll have to do is to add a hidden input component to all pages where we expect warnings to be shown that the user should be able to ignore. In our example, we only have to add this hidden component to our `EditEmployee.xhtml` page, as we will only be adding a warning-level constraint to our `Employee` entity. The following code snippet shows the hidden component added to the `EditEmployee.xhtml` page:

```
<ui:composition template="templates/template.xhtml">
  <ui:define name="title">Edit employee</ui:define>
  <ui:define name="content">
    <h:inputHidden id="extValWarnState"
                   value="#{extValWarnState.continueWithWarnings}"/>
    <tr:panelFormLayout>
    <!-- Form contents left out to save space. -->
    </tr:panelFormLayout>
  </ui:define>
</ui:composition>
```

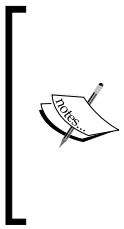

www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info



www.it-ebooks.info

```
    @Column(nullable=false)
    @VirtualMetaData(target=Column.class,
                     parameters=ViolationSeverity.Fatal.class)
    private String lastName;

   // All other variables and methods are omitted.
}
```

Note that the attributes of the `@VirtualMetaData` annotation accept exactly the same types as the attributes of all ExtVal Property Validation annotations. That's why we can use the same `ValdationSeverity` class in this case. The `target` attribute is needed to link the `@VirtualMetaData` annotation to the `@Column` annotation.

Summary

This chapter introduced MyFaces Extensions Validator, or ExtVal for short. After the installation of ExtVal, we saw that no configuration is needed to get started, based on standard JPA annotations. After that, we had a look at the extra annotations that ExtVal adds to facilitate more validation options and to enable cross validation. We saw how we can combine ExtVal with custom JSF validators. We also looked into creating custom error messages. We saw how we can customize and extend ExtVal in various ways. And finally, this chapter showed us how we can integrate ExtVal with JSR 303 Bean Validation.

This chapter has covered only the basics of what is possible with ExtVal. As ExtVal provides a very extensible and flexible infrastructure, the possibilities are virtually endless. More information can be found on the weblog of Gerhard Petracek, the lead developer of ExtVal, at `http://os890.blogspot.com/`. Another resource of additional information is a series of articles about ExtVal on JSF Central. The first article of the series can be found at `http://jsfcentral.com/articles/myfaces_extval_1.html`. The easiest way to find the other articles in the series is to just replace the `1` in the URL by a higher number. The series currently consists of three articles, but more may be added in the future.

The next and last chapter of this book will introduce some best practices for using the various MyFaces libraries.

www.it-ebooks.info

www.it-ebooks.info

11
Best Practices

A lot of best practices have been discussed throughout this book. However, some best practices didn't fit into one of the chapters. These are collected in this chapter.

After reading this chapter, you will be able to:

- Prevent direct access to page definitions, bypassing the Faces Servlet
- Enable container-based security in your JSF application
- Create a login page with JSF components
- Use component bindings wisely
- Save the state of request-scoped components in an elegant way

Preventing direct access to page definitions

A common problem with JSF applications is that the JSP or Facelets files that the Faces Servlet uses to render the pages are also accessible via the 'normal' web server process, bypassing the Faces Servlet. This can lead to unexpected errors, probably something like:

```
java.lang.RuntimeException: Cannot find FacesContext
```

This section presents a simple solution to this problem. A pretty straightforward solution is to implement a `Filter` that redirects requests that go directly to the page. A simple `Filter` implementation could look like this:

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
```

www.it-ebooks.info

```
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class RedirectFilter implements Filter {

  private final static String FACES_PREFIX = "/faces/";
  private final static String FILE_EXTENSION = ".jspx";

  public void init(FilterConfig config)
         throws ServletException {
  }

  public void destroy() {
  }

  public void doFilter(ServletRequest servletRequest,
                       ServletResponse servletResponse,
                       FilterChain chain)
         throws IOException, ServletException {

    HttpServletRequest request =
                       (HttpServletRequest) servletRequest;
    HttpServletResponse response =
                       (HttpServletResponse) servletResponse;
    String uri = request.getRequestURI();

    if ( ! uri.contains(FACES_PREFIX)
        && uri.endsWith(FILE_EXTENSION)) {

      int filePos = uri.lastIndexOf("/");
      String redirectUri = uri.substring(0, filePos)
                         + FACES_PREFIX
                         + uri.substring(filePos+1);
      response.sendRedirect(redirectUri);
    } else {
      chain.doFilter(servletRequest,servletResponse);
    }
  }
}
```


www.it-ebooks.info

Note how the URI is adapted by simple `String` manipulations. If the URI does contain the `/faces/` prefix, or does not end with `.jspx`, the filter does nothing. This is achieved by calling the `doFilter` method on the `FilterChain` object. The filter has to be registered in the `web.xml` file by adding the following fragment:

```
<filter>
   <filter-name>Redirect Filter</filter-name>
   <filter-class>RedirectFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>Redirect Filter</filter-name>
   <url-pattern>*.jspx</url-pattern>
</filter-mapping>
```

This tells the application server to send all URIs that end with `.jspx` to the filter named `Redirect Filter`, and this filter is implemented by the `RedirectFilter` class. Note that the previous implementation does not contain a `package` statement. Should your filter be in a package, then the fully-classified name of your class should be used in the `web.xml` file.

Although the previous code was written as a proof of concept, it might be sufficient for most applications. You might argue that the file extension and the URI prefix should be configurable. But how often do these values change in a real-life project? Most of the time, they don't. So it might not be worth the hassle to make those values configurable.

An alternative solution to the problem is to not use a prefix such as `/faces/`, but to map the Faces Servlet to a file extension instead. This can be done as shown in the following `web.xml` snippet:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```


www.it-ebooks.info

In this example, the Faces Servlet is mapped to any URI ending with `.xhtml`. This means that the URI that accesses a file directly is identical to the URI that maps to the Faces Servlet. Direct access to a file is not possible, as the URI will be mapped to the Faces Servlet. While that's a simpler solution, it might not always be possible, depending on other choices in the project.

Using container-managed security with JSF

In the previous chapters of this book, the topic security was somewhat ignored. That's fine, as it would not have added much to illustrating the possibilities of the various subprojects of MyFaces. However, a production quality application does need some degree of security most of the time. Java EE offers a container-managed security scheme, which allows us to add security to any Java EE application. This **Java Authentication and Authorization Service** (**JAAS**) takes care of pretty much everything that has to do with security. This can save us a lot of effort. Covering all of the possibilities of JAAS goes beyond the scope of this book. There are, however, some tricky things when it comes to using JAAS in conjunction with JavaServer Faces, and that is what this section is focusing on.

Enabling container-managed security

Let's quickly recall the steps needed to add security to our application. First, we have to configure security in our `web.xml` file:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All Access</web-resource-name>
    <url-pattern>/faces/*</url-pattern>       <!-- 1 -->
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>               <!-- 2 -->
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-role>
  <description>Application user</description>
  <role-name>user</role-name>                 <!-- 3 -->
</security-role>
<login-config>
```


www.it-ebooks.info

```
    <auth-method>FORM</auth-method>              <!-- 4 -->
    <realm-name>file</realm-name>
    <form-login-config>                          <!-- 5 -->
      <form-login-page>/faces/Login.xhtml
      </form-login-page>
      <form-error-page>/faces/LoginError.xhtml
      </form-error-page>
    </form-login-config>
  </login-config>
```

In this case, we simply deny all unauthorized access to any URL starting with `/faces/` (1). We only allow users that have the role of `user` to log in (2). We define that role a few lines further on (3). Then we configure it so that users should log in using a custom login form (4), and URLs to the login and error pages are configured too (5). Note that the URLs for the login and error pages are JSF URLs that will be processed by the JSF controller.

After configuring security in the `web.xml` file, we also have to map the roles that we defined in our application to user groups that are known by the application server. The way this is achieved can differ as per the application server. Generally, it can be done in a vendor-specific configuration file. For example, for the GlassFish application server, this can be done by creating a `sun-web.xml` file in the `WEB-INF` directory of the WAR. This file could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE sun-web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Application Server 8.1
                                              Servlet 2.4//EN"
  "http://www.sun.com/software/appserver/dtds/sun-web-
                                              app_2_4-1.dtd">
<sun-web-app>
  <security-role-mapping>
    <role-name>user</role-name>
    <group-name>miasusers</group-name>
  </security-role-mapping>
  <class-loader delegate="false"/>
  <property name="useMyFaces" value="true"/>
</sun-web-app>
```


www.it-ebooks.info

The highlighted lines show how security mapping looks for GlassFish. For most other application servers, the configuration is comparable to this. Refer to the documentation of your application server for further details. Of course, a user group named `miasusers` has to be created on the application server. Generally, this should be done inside a **security realm**. The exact steps to do this differ from application server to application server, and are not covered here.

Navigating to the login page

In the previous section, we configured JSF URLs for the login page and the error page. So we will have to add some navigation cases to our `faces-config.xml` file in order to make those URLs work. This is pretty straightforward:

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>login</from-outcome>
    <to-view-id>/Login.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>loginError</from-outcome>
    <to-view-id>/LoginError.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Creating the login page

The interesting part is creating the login page. The JAAS framework was created long before JSF even existed. So the JAAS framework does not anticipate the use of JSF pages, but expects old school JSPs instead. Unfortunately, no measures were taken to streamline the interaction with JAAS when the JSF standard was created. Due to this, JAAS has some requirements for the login page that are hard to meet through the use of JSF components. For example, the action of the login form has to be `j_security_check`, and the names of the username and password fields have to be `j_username` and `j_password`. For that reason, many examples in books and on the Web fall back on old school JSP login pages or even static HTML login pages. Wouldn't it be nice if we could just use a normal JSF page and use the same JSF components that we use on all other pages? Let's see how we can accomplish this.

www.it-ebooks.info

The main problem is that the components that normally render the HTML `<form>` tag, for example `<tr:form>`, don't let us define a custom action for the `<form>` tag. For the special `j_security_check` action to be called, the `<form>` tag in the rendered HTML page should look like this:

```
<form action="j_security_check" method="post">
```

Luckily, this is fairly easy to accomplish in a Facelets page definition:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:tr="http://myfaces.apache.org/trinidad">
<body>

<f:view>
<ui:composition template="templates/templateNoForm.xhtml">
<ui:define name="title">Login</ui:define>
<ui:define name="content">
<form method="post" action="j_security_check">
<tr:panelFormLayout>
<tr:inputText id="j_username"
required="true"
label="#{msg['userName']}:"
autoComplete="false" />
<tr:inputText id="j_password"
required="true"
label="#{msg['password']}:"
autoComplete="false"
secret="true" />
<h:commandButton value="#{msg['login']}"
id="login" />
</tr:panelFormLayout>
</form>
</ui:define> </
ui:composition>
</f:view> </body>
</html>
```


www.it-ebooks.info

If we had used JSP instead of Facelets, we would have had to surround the HTML `<form>` tag with `<f:verbatim>` tags. Note that we can't use a `<tr:commandButton>` here because Trinidad's command button does expect to be surrounded by a JSF form component such as `<h:form>` or `<tr:form>`. It should also be noted that we used a different template (`templates/templateNoForm.xhtml`) because the template we used for all other pages already includes a `<tr:form>` component.

Alternatives

Using an HTML `<form>` tag instead of a JSF form component is not an ideal solution, but it works and is fairly easy to implement. There are, however, other approaches to solving the login form problem. Some JSF books present a way to create a custom JSF component that creates a complete login form. Such a JSF component can be relatively simple because the processing of the input is done by JAAS outside of the JSF framework. So the only thing such a component should do is render a page with a login form. While this sounds very simple, creating a custom JSF component is not that simple. Creating a prototype might be relatively easy, but it might be more difficult to create a production-quality component.

Another alternative is to use a ready-made login form component. Unfortunately, none of the MyFaces component libraries offer such a component. Of course, you could try to find a login component in another component library. A downside to this could be that different component libraries do not always work well together in a single project, and it might be difficult to get your login page in the same look and feel as the rest of your application.

Logout link

To complete the security functionality, we also need to create a logout link. JAAS does not have standard functionality for this, but logging out requires nothing more than invalidating the user session, which can be done easily in a Servlet. So let's create a `LogoutServlet.java` class:

```
package inc.monsters.mias;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LogoutServlet extends HttpServlet {
  public LogoutServlet() {
    super();
```


www.it-ebooks.info

```
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                  throws ServletException, IOException {

      request.getSession().invalidate();
      response.sendRedirect("faces/Start.xhtml");
    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                   throws ServletException, IOException {

      doPost(request, response);
    }
  }
```

The highlighted lines invalidate the session and then redirect the user to the start page of the application. Because there is no logged-in user, JAAS will redirect to the login page automatically. We should register this Servlet in our `web.xml` file as follows:

```
<servlet>
  <display-name>LogoutServlet</display-name>
  <servlet-name>LogoutServlet</servlet-name>
  <servlet-class>inc.monsters.mias.LogoutServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>LogoutServlet</servlet-name>
  <url-pattern>/logout</url-pattern>
</servlet-mapping>
```

Now we can add a logout link to any page. Of course, the smartest thing to do is to add it to our template. In that case, a logout link will be available on all pages. We could, for example, use Trinidad's `<tr:goLink>` component:

```
<tr:goLink destination="/logout" text="Logout"/>
```


www.it-ebooks.info

Component bindings

Sometimes, one needs to programmatically access a JSF component that is used on a page. For such cases, all JSF components have a `binding` attribute that can be used to bind the component to a property in a backing bean. An example can be found on our `Kids.xhtml` page, where the delete button is bound to the `kidsTable` bean in order to make it possible to dynamically enable and disable the button, depending on the selection in the table:

```
<tr:commandButton actionListener="#{kidsTable.deleteSelected}"
                  text="#{msg.delete}"
                  disabled="#{!kidsTable.deleteEnabled}"
                  partialSubmit="true"
                  id="btnDelete"
                  binding="#{kidsTable.deleteButton}" />
```

The `kidsTable` bean has a `deleteButton` property:

```
public class KidsTable {
  ...
  private CoreCommandButton deleteButton;

  public CoreCommandButton getDeleteButton() {
    return deleteButton;
  }

  public void setDeleteButton(
                            CoreCommandButton deleteButton) {
    this.deleteButton = deleteButton;
  }
  ...
}
```

This is all fine, and it works as intended. However, whenever you use the `binding` attribute, caution should be taken. As can be seen in the previous code, the backing bean has a property of the `CoreCommandButton` type that holds a reference to the button object. It should be noted that a new button object is created for every request in the Restore View phase of the JSF life cycle. As long as the bean that holds the reference to this button object is in the request scope, the lifetime of both the button object and the bean is more or less the same and everything is fine. However, if the bean is in another scope and has a longer lifetime, there is no guarantee that the reference that the bean holds references a valid object. Consider the case where the bean is on the session scope and the user navigates to another page. The bean can still hold a reference to the button from the first page, but it doesn't make any sense, because that button isn't a part of the second page.

www.it-ebooks.info

This leads to a very simple rule of thumb: Never bind a component to a bean that is not in the request scope. Should you need to combine JSF component access with non-request-scoped beans, you could use dependency injection to inject a reference to your non-request-scoped bean into the request-scoped bean where all code that manipulates the JSF component resides.

Keeping the state of a component

Converter and validator components are part of the component tree of the view. This tree is recreated for every request. In other words, the tree lives in the request scope. A common problem in JSF is keeping the state of converters and validators or other components that live in the request scope. In Chapter 6, we created a custom converter that had a configurable `separator` property. This separator is part of the state of the converter and is needed in every request. Let's see how we can adapt that converter to keep this state over multiple requests.

In order to be able to save the state, we have to implement the `javax.faces.component.StateHolder` interface, which involves implementing four methods. If we implement these methods, our example converter from Chapter 6 will now look as follows:

```
package inc.monsters.mias.conversion;
import javax.faces.component.StateHolder;
// other import omitted
public class FoodListConverter implements Converter,
                                          ClientConverter,
                                          StateHolder {
  private String separator;
  // other members and methods omitted for brevity
  public boolean isTransient() {
    return false;
  }
  public void setTransient(boolean b) {
  }
  public void restoreState(FacesContext context,
                           Object object) {
    separator = (String) object;
  }
  public Object saveState(FacesContext context) {
    return separator;
  }
}
```


www.it-ebooks.info

The methods from the `StateHolder` interface are highlighted. The first two methods can be used to dynamically switch on or off state saving; we simply return `false` here to make sure that the state will be saved all the time. Every time a request is handled and the component is removed from memory, the `saveState()` method will be called by the JSF controller. Any value that is returned will be saved and passed to the `restoreState()` method whenever a new request has to be handled and the state has to be restored.

In our case the state consists of one simple property, so the implementation can be super simple. If more properties have to be saved, they must be wrapped in a single object (for example, an array or a `Collection`) before they can be saved. This can lead to a lot of boilerplate code just to save and restore the state. Luckily, the MyFaces Trinidad project offers a well-designed mechanism to solve this. The mechanism is implemented around an interface called `FacesBean`. If we choose to use the `FacesBean` approach, our class would look like this:

```
package inc.monsters.mias.conversion;

import javax.faces.component.StateHolder;
import org.apache.myfaces.trinidad.bean.FacesBean;
import org.apache.myfaces.trinidad.bean.PropertyKey;
// other imports omitted

public class FoodListConverter implements Converter,
                                          ClientConverter,
                                          StateHolder {

  private static final FacesBean.Type TYPE =
                                new FacesBean.Type();
  private static final PropertyKey SEPARATOR =
        TYPE.registerKey("foodListConverterSeparator",
                         String.class);
  private FacesBean facesBean = new FacesBeanImpl() {
    @Override
    public Type getType() {
      return TYPE;
    }
  };
  // other methods omitted

  public String getSeparator() {
    return (String) facesBean.getProperty(SEPARATOR);
  }

  public void setSeparator(String separator) {
    facesBean.setProperty(SEPARATOR, separator);
  }
```


www.it-ebooks.info

```
    public boolean isTransient() {
      return false;
    }

    public void setTransient(boolean arg0) {
    }

    public void restoreState(FacesContext context,
                             Object state) {

      facesBean.restoreState(context, state);
    }

    public Object saveState(FacesContext context) {
      return facesBean.saveState(context);
    }
  }
```

The first highlighted block creates a new `FacesBean` object by subclassing the abstract `FacesBeanImpl` class, inline. This somewhat complicated approach has to do with the typesafe way in which the `FacesBean` idea is implemented. We have adapted the `getSeparator()` and `setSeparator()` methods to store the `separator` value in the `FacesBean`, instead of in a `private` variable. The saving and restoring of the state is now delegated to the `FacesBean`, as shown in the last two highlighted lines. Although the `FacesBean` approach requires a bit more typing if we only want to save the state of a single variable, it saves us work if we have more variables to save. Also, it adds type safety in all cases.

Summary



www.it-ebooks.info

www.it-ebooks.info

Index

Symbols

<accessibility-mode> element 255
<accessibility-profile> element 255
<c:choose> element 146
<client-validation> element 260
@Column annotation 275, 323
@Constraint 352
<currency-code> component 262
@DoubleRange annotation 323
@EJB annotation 282
@Entity annotation 274
<f:convertNumber> component 89
@GeneratedValue annotation 274
<h:commandLink> component 99
<h:ouptutText> component 99
<h:outputText> component 99
@Id annotation 274
@JoinValidation annotation 327
@Length annotation 323
@LongRange annotation 323
<managed-property> element 290
<mias:field> composition component
constructing 135
<tr:inputText> component, declaring 136
@NotNull 352
<number-grouping-separator> component 262
<output-mode> element 260
@OverridesAttribute() 352
<ox:dynaForm> component
about 312
attributes 312
<ox:dynaForm> component, attributes
id 312
uri 312
valueBindingPrefix 312
<page-flow-scope-lifetime> element 256
@PersistenceContext annotation 277
@Required annotation 323
@Retention 352
<right-to-left> element 261
@Size(min=2) 352
<skin-family> element 261
@Stateless annotation 277
@Table annotation 274
<t:aliasBean> 83
<t:aliasBeanScope> 83
@Target 351
<t:buffer> 83
<t:captcha> 83, 84, 85, 86
<t:commandSortHeader> component 95
attributes 95
<t:commandSortHeader> component, attributes
columnName 95
propertyName 95
<t:dataScroller> component
about 89
attributes 89
facets 90
<t:dataScroller> component, attributes
fastStep 90
for 89
paginator 90
paginatorMaxPages 90
<t;dataTable> component 99
about 87
attributes 87
<t:dataTable> component, attributes
id 87
rows 87

www.it-ebooks.info

value 87
var 87
@Temporal annotation 275
<t-graphicImage> component 91
<time-zone> element 261
<t:inputCalendar> component
about 109
attributes 109
<t:inputCalendar> component, attributes
addResources 109
id 110
javascriptLocation 110
popupButtonImageUrl 111
popupButtonString 111
popupDateFormat 109
popupLeft 111
popupSelectMode 110
popupTodayDateFormat 110
renderAsPopup 109
renderPopupButtonAsImage 110
styleLocation 110
value 110
<t:inputFileUpload> component
about 106
attributes 106
<t:inputFileUpload> component, attributes
accept 106
maxlength 107
storage 107
value 106
@Transactional annotation 309
<tr:breadCrumbs> component 161
<tr:column> component 124
features 124
<tr:commandNavigationItem> component 161
<tr:convertDateTime> component 137
<tr:convertNumber> converter 149
<tr:document> tag 122
<tr:inputDate> component 136
<tr:inputFile> component 152
<tr:inputNumberSpinbox> component 150
<tr:inputText> component 134
<tr:navigationPane> component 160
<tr:navigationTree> component 162
<tr:outputFormatted> component 173
<tr:panelAccordion> component 169
<tr:panelBorderLayout> component 163, 164
<tr:panelBox> component 172
<tr:panelButtonBar> component 168, 176
<tr:panelCaptionGroup> component 177
<tr:panelChoice> component 171
<tr:panelFormLayout> component 132
<tr:panelFormLayout> layout component 167
<tr:panelHeader> component 174
<tr:panelHorizontalLayout> component 166
<tr:panel...Layout> component 163
<tr:panelList> component 179
<tr:panelPage> component
about 180
facets 180
<tr:panelPageHeader> component
about 182
facets 182
<tr:panelPopup> component 175
<tr:panelRadio> component 172
<tr:panelTabbed> component 170
<tr:selectItem> component 142
features 144
<tr:selectManyListbox> component 144
<tr:selectManyShuttle> component 144
<tr:selectOneChoice> component 142
<tr:selectOneListbox> component 142, 144
<tr:selectOneRadio> component 142, 143
<tr:selectOrderShuttle> component 145
<tr:showDetailItem> component 169
<tr:table> component
about 123
features 123
<tr:validateDateRestriction> 139
<tr:validateDateTimeRange> component 138
<tr:validateDoubleRange> component 150
<tr:validateLongRange> component 150
<t:validateCreditCard> component 116
<t:validateEmail> validator component 115
<t:validateEqual> component 115
<t:validateRegExpr> component 116
<two-digit-year-start> element 261
<ui:component> tag
about 70
attributes 71

www.it-ebooks.info

<ui:composition> tag
about 71
attributes 71
<ui:debug> tag
about 71
attributes 71
<ui:decorate> tag
about 72
attributes 72
<ui:define> tag
about 72
attributes 73
<ui:fragment> tag
about 73
attributes 73
<ui:include> tag
about 74
attributes 74
<ui:insert> tag
about 74
attributes 74
<ui:param> tag
about 74
attributes 74
<ui:remove> tag 75
<ui:repeat> tag 75
<uploaded-file-processor> element 256
@Validator annotation 325
@ViewController annotation 299

A

Abstract Window Toolkit (AWT) 85
accessibility mode, Trinidad
about 255
default 255
inaccessible 255
screenReader 255
accessibility options, Trinidad
about 255
accessibility mode 255
accessibility profile 255
lightweight dialogs 256
accessibility profile, Trinidad
about 255
high-contrast 255
large-fonts 255
accessKey attribute 132
accordion
creating 169, 170
action attribute 99
addPartialTarget() method 209
ADF Faces project 13
advanced data table features, Tomahawk
details inline, showing 98, 99
edit form, linking to 100-103
newspaper columns 104, 105
rows, grouping 103
sort arrows, improving 97
sorting 94-97
Advanced Meta Data add-on 356
alias selectors 249
animationDuration 191
Apache Batik 80
Apache Commons IO 80
Apache Commons Validator 80
Apache Geronimo application server 35
Apache Incubator project 9
Apache MyFaces
about 9
community support 10
example use 41, 42
license 10
sub-projects 11
Sun JSF RI 10
appAbout facet 180
appCopyright facet 180
appearance, Trinidad
client validation 260
output mode 260
skin family 261
application scope 199
application structure, Orchestra
adapting 286
application view caching 258
appPrivacy facet 180
arguments, @DateIs annotation
errorMessageDateStyle 330
notAfterErrorMsgKey 329
notBeforeErrorMsgKey 329
notEqualErrorMsgKey 329
type 329
validationErrorMsgKey 330
valueOf 329

www.it-ebooks.info

arguments, @Equals and @NotEquals
validationErrorMsgKey 330
value 330
arguments, @RequiredIf annotation
is 331
validationErrorMsgKey 331
value of 331
attributes, Tomahawk components
disabledOnClientSide 81
displayValueOnly 81
displayValueOnlyStyle 81
displayValueOnlyStyleClass 81
enabledOnUserRole 81
forceIdIndex 82
visibleOnUserRole 82
autoComplete attribute 134
autoSubmit attribute 133 203
autowire attribute 295

B

basic data tables, Tomahawk
columns, adding 88, 89
creating 86
CSS styling, adding 91
data scroller, styling 93
data table, setting up 87
pagination, using 89, 90
styling 92
Bean Validation
about 15, 344
annotations, using 346
continue with warnings add-on, setting up 353, 354
libraries, adding 345
payload attribute, using 353
reusing 349
setting up 345
severity level of constraint, setting 354, 355
severity level on any constraint, setting 356
severity level on ExtVal Property Validation constraints, setting 355
severity levels, setting 353
using 344
Bean Validation annotations
about 349
groups attribute 348
message attribute 347
payload attribute 348
using 346
Bean Validation, reusing
custom constraints, composing 350-353
recursive validation, using 350
validation, inheriting 349
boxes
displaying 172
brandingAppContextual facet 183
brandingApp facet 183
branding facet 180-183
bravenessCalcReturn method 220
bug, partialTriggers 205
bullet lists
creating 179
button bars
creating 176

C

CAPTCHA 83
caption groups
using 177
captionText attribute 178
Cascading Style Sheets (CSS) 241
chart types
area chart 194
area chart, stacked version 194
bar line chart 194
circularGauge chart 196
funnel chart 196
line chart 193
normal bar chart, rotated version 192
pie chart 195
radar area chart 195
radar chart 195
scatterPlot chart 197
semiCircularGauge chart 196
simple bar chart 192
stackedVerticalBar, roated version 193
standard bar chart, stacked version 193
XYLine chart 197
choice panel
creating 171
ClientConverter interface
getClientConversion() method 225

www.it-ebooks.info

getClientImportNames() method 225
getClientLibrarySource() method 225
getClientScript() method 225
implementing 225
client-side conversion
ClientConverter interface, implementing 225
client-side code, implementing 227
enabling 225
client-side validation and conversion
about 221, 222
client-side capabilities, enabling 225, 226
converter, creating 224, 225
data structure, defining 222, 223
internationalization, of messages 233
JavaScript, debugging 235
JavaScript, testing 235
JavaScript, writing 235
Trinidad JavaScript API, using 235
validator, creating 228, 229
wiring 231
client validation
about 260
alert 260
disabled 260
inline 260
ColorBean class 142
columnBandingInterval attribute 128
columns attribute 134
component bindings, JSF 368, 369
component piece selectors 248
component state selectors 247
components, Tomahawk
CAPTCHA component 12
date selection components 12
extensive data table component 12
file upload component 12
composition components, Facelets
<h:message> component, adding 64, 65
actual composition component, defining 62
creating 58
redundancies, identifying 60
required indicator, adding 64, 65
skeleton, creating 61
tag library, creating 58, 59
using 66
validators, adding 63
constraint compositions, Bean Validation 350
container-managed security
enabling 362, 363
using, with JSF 362
content interweaving, Facelets 44
continue with warnings add-on 353
Conversation Scope 14, 300
converter
creating 224, 225
Core project
about 9-12
JSF 1.1 11
JSF 1.2 11
JSF 2.0 11
relevant versions 11
createNamedQuery() method 277
createQuery() method 277
cross validation
applying 328
using, based on equality 330
using, for date values 329
value, making required conditionally 331
CSS selector 247
custom constraints, Bean Validation
composing 350
custom error messages
creating 331
ExtVal default error messages, overriding 332, 333
standard JSF error messages, overriding 332
custom validation strategy
annotation-based configuration 339
configuring 337, 338
configuring, alternative configuration add-ons used 339
creating 334
implementing 335, 336
using, in ExtVal 337, 338
custom validators
using, @Validator annotation used 325

D

data
passing, with page flows 198-201

www.it-ebooks.info

database
connecting to 268, 269
creating 267, 268
managing 269, 270
table, creating 271
table, populating with data 272
Database Development perspective 269
database environment
preparing 267
data input fields, creating
about 136
dates, converting 137
dates, validating 138
ultimate date input component, creating 140, 141
Data Source Explorer 269
data tables, creating
about 123, 124
banding, configuring 128
columns, adding 124-126
grid, configuring 128
inline details, displaying 127
pagination, using 126, 127
row selection, using 129-131
Data Tools Platform 269
data visualization
about 185
chart types 192
data display, changing 191
data model, creating 186
data model, initializing 189, 190
graph, adding to page 190
graph look, changing 191
ideas 197
minimal data model, implementing 186-188
values, calculating 188, 189
dates and calendars, Tomahawk
calendar, using in form 113, 114
inline calendar, using 112
pop-up calendar, localizing 111
pop-up calendar, using 109
working with 108
debugging, Facelets 49, 50
debugging skins 242
debugging, Trinidad
about 258
compression, turning off 259
debug output, enabling 259
deployed files, changing 260
obfuscation, turning off 259
debug output
enabling 259
detailToggler object 99
development environment
configuring 17
Eclipse, configuring 18
JDeveloper, configuring 25
dialog
backing bean, creating 216-218
building 215, 216
calling 218, 219
creating 214
inputListOfValues, using as alternative 220, 221
lightweight dialogs, using 221
output, receiving 219, 220
values, returning in alternative way 218
direct access, to page definitions
preventing 359-361
displayValueOnly attribute 82
dispose() method 153
doApply() method 102, 105
done() method 218
Dont Repeat Yourself (DRY) 45
double range validation
defining, @DoubleRange annotation used 323
DRY principle 15
DynaForm
about 310
forms, generating with 310
installing 310
using 311-313
dynamic web project, Eclipse
preparing 21-25

E

EAR
creating 265
creating, in Eclipse 267
Eclipse
configuring 18
extra plugins, installing 18

www.it-ebooks.info

libraries, installing 20
new project, preparing 21
Trinidad tag support 18
web page editor 18
editKidForm bean 153
EJB 3.0
limitations 283
EJB 3.0, limitations
data validation 284
transactions 283, 284
EJB JAR
about 264
application server specific 265
creating, in Eclipse 265
ejb-jar.xml 264
MANIFEST.MF 264
orm.xml 265
persistence.xml 265
EL 45
elements, faces-config.xml configuration file
application 38
converter 38
managed-bean 38
navigation-rule 38
render-kit 38
validator 38
Employee object 277
EmployeeServiceBean class 277
endConversationAndSave() method 309
entity
creating 272-275
EntityManager object 277
event methods, Orchestra ViewController
about 299
initView 299
preProcess 299
preRenderView 300
Expression Language. *See* **EL**
extended components, Tomahawk
<t:aliasBean> 83
<t:aliasBeanScope> 83
<t:buffer> 83
<t:captcha> 83
about 83
Extensions Filter 79
Extensions Validator project 315
extra validators, Tomahawk
about 115
credit card numbers, validating 116
e-mail addresses, validating 115
equality, validating 115
user input, validating against regular expression 116
ExtVal
about 14, 315
additional annotations, using 324
add-ons, getting 341
add-ons, installing 344
basic usage 321
Bean Validation, using 344
convention over configuration pattern, using 320
cross validation, applying 328
custom error messages, creating 331
custom validation strategy, creating 334
extending 340
extending, with add-ons 341
extra annotations 322
factory design pattern 340
features 316
JPA annotations, complementing 322
libraries, adding to EAR 319, 320
library 317
overridden concepts 340
setting up 316
ExtVal add-ons
advanced metadata 342
annotation-based configuration 341
description 341
documentation 341
getting 341
Google Guice style configuration 342
installing 344
secured action 343
warnings 343
ExtVal annotations
@JoinValidation annotation 326
@Length annotation 322
@Pattern annotation 324
custom validators, using 325
double range validation, defining 323
length validation, defining 323
long range validation, defining 323
pattern-based validation, defining 324

www.it-ebooks.info

required fields, defining 323
using, for standard JSF validators 322
@Validator annotation 325
ExtVal default error messages
overriding 332, 333

F

facade 275
Facelets
about 9, 43
benefits 45
comments, using in page definitions 55-57
composition components, creating 58
content interweaving 44
DRY 45
EL, expanding 45
inline texts, using 69, 70
need for 43
static functions, using 67, 69
template, creating 51, 52
template, using 52-54
templating 44
templating with 51
XHTML files 57
Facelets project
debugging 49, 50
faces-config.xml, preparing 47
setting up 46
test page, creating 47, 49
web.xml, preparing 46
Facelets tags
<ui:component> tag 70, 71
<ui:composition> tag 71
<ui:debug> tag 71
<ui:decorate> tag 72
<ui:define> tag 72
<ui:fragment> tag 73
<ui:include> tag 74
<ui:insert> tag 74
<ui:param> tag 74
<ui:remove> tag 75
<ui:repeat> tag 75
faces-config.xml file
about 37, 38
configuring 121
configuring, for Spring 293
elements 38
example 39, 40
Faces Servlet 35
fieldWidth attribute 167
file upload component
creating 152
file, saving in backing bean 153-155
using 152
fileUploadField.xhtml file 152
file upload limits
configuring 155
setting, in trinidad-config.xml 156, 157
setting, in web.xml 156
file upload, Tomahawk 105, 106
file upload, Trinidad
about 151
file upload component, using 152
file upload limits, configuring 155
prerequisites 151
find() method 277
FoodListConverter class 226
full submit 202

G

generic application, JDeveloper
preparing 28-32
getAge() method 218
getAgeVsBraveness() method 190
getAsObject() method 225
getAsString() method 225
getBravenessInput() method 220
getBraveness() method 89, 218
getBytes() method 106
getCaptcha() method 85
getClientConversion() 225
getClientImportNames() 225
getClientLibrarySource() 225
getClientScript() 225
getContentType() method 153
getEmployees() method 277
getFilename() method 153
getGroupLabels() method 188
getId() method 274
getInputStream() method 153
getKids() method 86
getLength() method 153

www.it-ebooks.info

getName() method 106
getResultList() method 277
getReturnValue() method 220
getRowData() method 131
getSeriesLabels() method 188
getValue() method 131
getYValues() method 187
global styles
  setting, alias selectors used 248
gradientsUsed 191

H

headercolspan attribute 88
headerColSpan variable 89
header panel
  using 174
headerText attribute 124
horizontalGridVisible attribute 128

I

infoFootnote facet 180
infoReturn facet 181
infoStatus facet 181
infoUser facet 181
initView, event methods 299
inline calendar 112
inline texts, Facelets
  using 69, 70
input and edit forms
  creating 132
  date input fields, creating 136
  fields for numerical input, creating 149
  plain text input fields, creating 134
  selection lists, creating 141
input components
  about 132
  features 132
input components, features
  automatic label rendering, using 132
  auto submit, using 133
  error message support, using 133
  required indicator, using 133
input forms layouts
  components, grouping 167
  creating 167
  footer facet 168
  label 168
  message 168
inputListOfValues
  using, as alternative 220, 221
internationalization, of messages
  about 233
  error message, formatting 234, 235
  getClientValidator(), changing 233
  JavaScript constructor, changing 234
int maxLength() default 20 352
invalidDays attribute 139
invalidDaysOfWeek attribute 139
invalidMonths attribute 139
itemLabel attribute 147
itemValue attribute 147

J

Java Authentication and Authorization Service (JAAS) 362
Java DB database 267
Java Persistence Query Language (JPQL) 278
JavaServer Pages. *See* **JSP**
Java Virtual Machine (JVM) 85
JDeveloper
  configuring 25
  libraries, installing 25-28
  new project, preparing 28
JEE application structure
  EAR, creating to wrap 265, 266
  setting up 264
  skeleton EJB JAR, creating 264
JPA annotations
  complementing 322
JPQL queries 278
JSF
  login page, creating 364
  login page, navigating to 364
JSF Reference Implementation 10
JSF security 362
JSP 43
JSR 303 15

www.it-ebooks.info

K

kidsTable bean 295

L

labelAndAccessKey attribute 132
label attribute 132
labelWidth attribute 167
layout attribute 165
layout methods
  expand 164
  positioned 164
leadingDescShown attribute 144
leadingHeader attribute 144
legendPosition 192
length validation
  defining, @Length annotation used 323
level attribute 161
library, ExtVal
  myfaces-extval-core-1.2.x.jar 317
  myfaces-extval-generic-support-1.2.x.jar 317
  myfaces-extval-property-validation-1.2.x.jar 317
  myfaces-extval-trinidad-support-1.2.x.jar 317
lightweight dialogs
  using 221
lightweight dialogs, Trinidad 256
localization attributes, <t:inputCalendar> component
  popupButtonString 111
  popupGotoString 111
  popupScrollLeftMessage 111
  popupScrollRightMessage 111
  popupSelectDateMessage 111
  popupSelectMonthMessage 111
  popupSelectYearMessage 111
  popupTodayString 111
  popupWeekString 112
localization, Trinidad
  about 261
  direction, reading 261
  number notation 262
  time zone 261
  two-digit year start 261
location facet 181
login page, JSF
  alternatives 366
  creating 364, 366
  logout link 366, 367
  navigating to 364
long range validation
  defining, @LongRange annotation used 323

M

Maven
  benefits 33
maxColumns attribute 167
maximum attribute 138
maximumLength attribute 134
maxPrecision 191
menuSwitch facet 183
merge() method 278
messageDetailConvertBoth attribute 137
messageDetailConvertDate attribute 137
messageDetailConvertPattern 149
messageDetailConvertTime attribute 137
messageDetailInvalidDays attribute 139
messageDetailInvalidDaysOfWeek attribute 139
messageDetailInvalidMonths attribute 139
messageDetailMaximum attribute 138
messageDetailMinimum attribute 138
messageDetailNotInRange attribute 138
MIAS-Entities.jar 287
mias skin family, example 243
MIAS.war 287
minimal, default skin 245
minimum attribute 138
model implementation, JEE application
  about 272
  data source, defining 280, 281
  entity, creating 272-274
  named queries, creating 278
  persistence units, defining 279
  service facade, creating 275, 277
modules, ExtVal
  bean validation 318
  core 318
  downloading 318
  generic support 318
  property validation 318

www.it-ebooks.info

trinidad support 318
Mojarra 10, 40
MVC pattern
about 263, 264
controller 264
goal 263
model 263
view 264
MyFaces
using, on GlassFish 40
MyFaces Extensions umbrella project 14
MyFaces Orchestra
about 285
setting up 286

N

navigation1 facet 181, 183
navigation2 facet 181, 183
navigation3 facet 181
navigationGlobal facet 181, 183
new project
creating, Maven used 33
numerical input fields
conversion, adding 149
creating 149
spin box, adding 150
validation, adding 150

O

Orchestra
about 14
application structure, adapting 286
configuring 297, 298
downloading 296
installing 296
setting up 286
Spring, configuring 288
Spring framework, downloading 287
Orchestra conversations
creating 301-304
ending 307, 309
extending 305, 306
setting up 300, 301
Orchestra Sandbox project 310
Orchestra ViewController
about 299
event methods, using 299
using 299
orientation attributes 162
output mode
about 260
default 260
email 260
printable 260
overridden concepts, ExtVal
componentInitializer 340
InformationProviderBean 340
MessageResolver 340
MetaDataExtractionInterceptor 340
MetaDataTransformer 340
NameMapper 340
ProcessedInformationRecorder 340
RendererInterceptor 340
StartupListener 340
ValidationStrategy 340

P

page flow scope 198, 301
page header panel
using 182
page layouts, creating
about 163
accordion, creating 169
border layout, using 163
boxes, displaying 172, 173
bullet lists, creating 179, 180
button bars, creating 176, 177
caption groups, using 177
choice panel, creating 171
group layout, using 165, 166
header panel, using 174, 175
horizontal layout, using 166
input forms layout, creating 167
page header panel, using 182
pop ups, using 175, 176
radio panel, creating 172
tabbed panel, creating 170
tips, displaying 174
Partial Page Rendering
about 202
addPartialTarget() method, using 209, 210
autoSubmit attribute, using 203-206

www.it-ebooks.info

components, dynamically hiding or showing 211, 212
full submit, comparing with partial submit 202, 203
naming containers, working with 207, 208
partial submit, comparing with full submit 202, 203
partialTriggers attribute, using 203-206
partialTriggers, working with 207, 208
polling 212
possibilities, exploring 213, 214
progress indicator, creating 209
partial submit 203
partialTriggers attribute
about 204
working with 207, 208
pattern-based validation
defining, @Pattern annotation used 324
performance, Trinidad
about 256
application view caching 258
page flow scope lifetime 256
state saving 257
uploaded file processor 256
perspective 192
pop-up calendar
about 109
localizing 111
pop ups
using 175
preProcess, event methods 299
preRenderView, event methods 300
progress indicator
creating 209
propertyName attribute 96
property reference, beans
expression language reference 327
property chain reference 327
static reference 328
pseudo selectors 247
public @interface Name 351

R

radio panel
creating 172
recursive validation, Bean Validation 350
RequestContext class 218
request scope 198, 300
required fields
defining, @Required annotation used 323
returnFromDialog() method 218
rowBandingInterval attribute 128
rows attribute 134, 167

S

Sandbox 13
savePhoto() method 105
saveSelected() method 282
search facet 181
secret attribute 134
security realm 364
selectedKid object 153
selectedKid property 101
selectionChanged() method 143
selection lists, creating
checkboxes 143
choice list 144
listboxes 144
list contents, adding 142
options, selection components 143
shuttle 144
shuttle, ordering 145
universal composition component, creating 146-149
selectItem component 143
separator facet 165
service facade
creating 275, 277
Session Bean 276
session scope 198, 300
setRowKey() method 131
setSelectedKid() method 102
setSubmittedValue(null) 220
shortDesc attribute 142
simple attribute 133
skeleton EJB JAR
creating 264
skin family 261
skinning, Trinidad. *See* **Trinidad skinning**
sortable attribute 95
specific application servers
settings 40

www.it-ebooks.info

Spring configuration
about 288
beans, managing 288-292
faces-config.xml file, configuring 293
for persistence 294
services, accessing 295
web.xml file, configuring 293
Spring framework
configuring 288
downloading 287
standard JSF error messages
overriding 332
state saving 257
state-saving mechanism
implementing, for request-scoped components 369-371
static functions, Facelets
using 67, 69
String message() default {...} 352
sub-projects, Apache MyFaces
about 11
Core 11
Extensions Validator 14
Orchestra 14
Portlet Bridge 14
Sandbox 13
Tobago 14
Tomahawk 12
Trinidad 13
Sun JSF RI 10

T

tabbed panel
creating 170, 171
tables
creating, <tr:table> component used 123
templating, Facelets 44
terminology, Trinidad
group 186
series 186
X axis 186
Y axis 186
text attribute 175
tip panel
displaying 174
title attribute 175
Tobago 14
Tomahawk
about 9, 12, 77
advanced data table features 94
basic data tables, creating 86
components 12
dates and calendars, working with 108
dependencies, resolving 80
downloading 78
extended components 83
extended versions, standard components 81, 82
extra validators 115
files, uploading 105-107
setting up 78
variants 12
versions 78
web.xml, configuring 79, 80
Tomahawk components
attributes 81
Tomahawk Core 12
Tomahawk Core 1.2 12
Tomahawk versions
JSF 1.1 78
JSF 1.2 78
tooltipsVisible 192
toStringWithSeparator() method 225
trailingDescShown attribute 144
trailingHeader attribute 144
Trinidad
<tr:column> component 124
<tr:table> component 123
about 13, 185, 241
advanced features 185
AJAX, using 202
characteristics 13
client-side conversion 221
client-side validation 221
data, passing with page flows 198
data tables, creating 123
data visualization 185
dialogs, creating 214
downloading 120
file, uploading 151
input components 132
JSF 1.1 version 120
JSF 1.2 version 120

www.it-ebooks.info

layout components 163
navigation framework 158
overview 119
Partial Page Rendering 202
setting up 120
skinning 241
terminology 186
tuning 253
versions 120
trinidad-config.xml 254
trinidad-config.xml file
configuring 122
Trinidad JavaScript API
about 235
debugging 236
JavaScript code, writing 235
Trinidad JavaScript APIlogging 236-238
writing 235
Trinidad navigation framework
about 158
breadcrumbs, creating 161, 162
hierarchical menu, creating 162
hierarchy, configuring 158-160
navigation panes, creating 160
Trinidad setup
faces-config.xml file, configuring 121
template, adapting 122, 123
trinidad-config.xml file, configuring 122
web.xml file, configuring 120, 121
Trinidad skin
creating 246
extending 253
Trinidad skin, creating
component piece selectors, using 248
component state selectors, using 247
global styles, setting 248, 249
icons, skinning 249, 250
skinning components 246
text, skinning 251, 252
Trinidad skinning
about 242
render kit 242
self-contained skin, creating 243
setting up 242, 243
skin, creating 246
skin family 242
skin ID 242
skin, selecting 244-246
Trinidad tuning
about 253
accessibility options 255
appearance 260
debugging 258
localization 261
performance 256
trinidad-config.xml 254
web.xml file 254

U

UIXTable 131
updateEmployee() method 277
UploadedFile class 153
dispose() method 153
getContentType() method 153
getFilename() method 153
getInputStream() method 153
getLength() method 153
URL specifying ways, Trinidad skinning
absolute URL 250
relative to context root 250
relative to CSS file 250
relative to server 250

V

validate() function 231
validate() method 84
validator
client-side capabilities, enabling 229
client-side code, implementing 230
ClientValidator interface 230
creating 228
Validator project 14
verticalGridVisible attribute 128
viewIds parameter 299
view layer, JEE application
pages, updating 282, 283
service facade, using 281, 282
Virtual Meta Data add-on 356

W

web.xml configuration file 35-37
web.xml file

www.it-ebooks.info

about 254
configuring 120, 121
configuring, for Spring 293
wiring
about 231
converter, declaring in faces-config.xml 231
converter, using in page 232
custom tags, creating 232
validator, declaring in faces-config.xml 231
validator, using in page 232
wrap attribute 134

X

XHTML files 57
XMajorGridLineCount 191
XMLMenuModel class 160

Y

YMajorGridLineCount 191
YMinorGridLineCount 191

www.it-ebooks.info

www.it-ebooks.info



Thank you for buying

Apache MyFaces 1.2 Web Application Development

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing Apache MyFaces 1.2 Web Application Development, Packt will have given some of the money received to the Apache project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

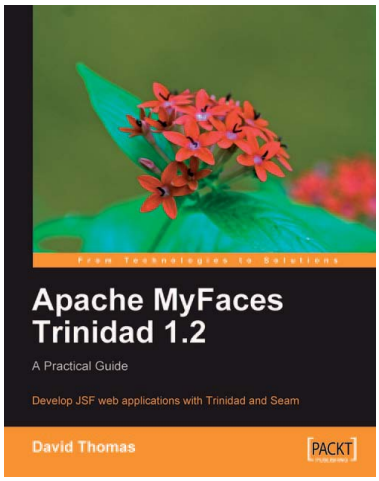
About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

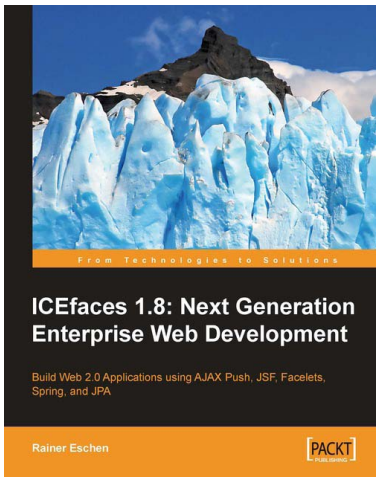
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

www.it-ebooks.info



www.it-ebooks.info

PACKT PUBLISHING

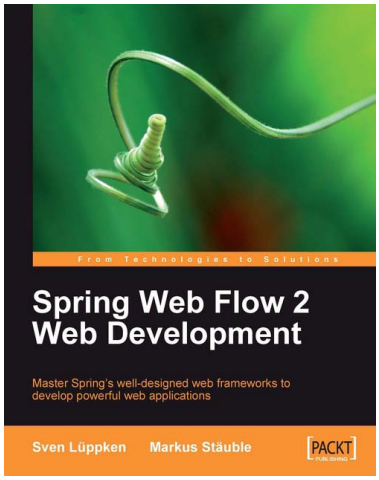


ICEfaces 1.8: Next Generation Enterprise Web Development

ISBN: 978-1-847197-24-5 Paperback: 292 pages

Build Web 2.0 Applications using AJAX Push, JSF, Facelets, Spring and JPA

1. Develop a full-blown Web application using ICEfaces
2. Design and use self-developed components using Facelets technology
3. Integrate AJAX into a JEE stack for Web 2.0 developers using JSF, Facelets, Spring, JPA



Spring Web Flow 2 Web Development

ISBN: 978-1-847195-42-5 Paperback: 272 pages

Master Spring's well-designed web frameworks to develop powerful web applications

1. Design, develop, and test your web applications using the Spring Web Flow 2 framework
2. Enhance your web applications with progressive AJAX, Spring security integration, and Spring Faces
3. Stay up-to-date with the latest version of Spring Web Flow
4. Integrate MySQL with OpenSER

Please check **www.PacktPub.com** for information on our titles

www.it-ebooks.info

www.it-ebooks.info