JBoss Enterprise Application Platform 5

Seam Reference Guide

for use with JBoss Enterprise Application Platform 5

Edition 5.2.0

Last Updated: 2017-10-13

JBoss Enterprise Application Platform 5 Seam Reference Guide

for use with JBoss Enterprise Application Platform 5
Edition 5.2.0

Gavin King

Pete Muir

Norman Richards

Shane Bryzak

Michael Yuan

Mike Youngstrom

Christian Bauer

Jay Balunas

Dan Allen

Max Andersen

Emmanuel Bernard

Nicklas Karlsson

Daniel Roth

Matt Drees

Jacob Orshalick

Marek Novotny

Edited by

Elspeth Thorne

Eva Kopalova

Laura Bailey

Petr Penicka

Russell Dickenson

Samson Kittoli

Scott Mumford

Legal Notice

Copyright © 2012 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This book is a Reference Guide for JBoss Enterprise Application Platform 5 and its patch releases.

Table of Contents

CHAPTER 1. SEAM TUTORIAL **14**
1.1. USING THE SEAM EXAMPLES 14
1.1.1. Running the examples on JBoss Enterprise Application Platform 14
1.1.2. Running the example tests 14
1.2. YOUR FIRST SEAM APPLICATION: THE REGISTRATION EXAMPLE 15
1.2.1. Understanding the code 15
1.2.1.1. The entity bean: User.java 15
1.2.1.2. The stateless session bean class: RegisterAction.java 17
1.2.1.3. The session bean local interface: Register.java 20
1.2.1.4. The view: register.xhtml and registered.xhtml 20
1.2.1.5. The Seam component deployment descriptor: components.xml 21
1.2.1.6. The web deployment description: web.xml 22
1.2.1.7. The JSF configuration: faces-config.xml 23
1.2.1.8. The EJB deployment descriptor: ejb-jar.xml 23
1.2.1.9. The EJB persistence deployment descriptor: persistence.xml 24
1.2.1.10. The EAR deployment descriptor: application.xml 24
1.2.2. How it works 25
1.3. CLICKABLE LISTS IN SEAM: THE MESSAGES EXAMPLE 26
1.3.1. Understanding the code 26
1.3.1.1. The entity bean: Message.java 26
1.3.1.2. The stateful session bean: MessageManagerBean.java 27
1.3.1.3. The session bean local interface: MessageManager.java 30
1.3.1.4. The view: messages.jsp 30
1.3.2. How it works 31
1.4. SEAM AND JBPM: THE TODO LIST EXAMPLE 31
1.4.1. Understanding the code 32
1.4.2. How it works 38
1.5. SEAM PAGEFLOW: THE NUMBERGUESS EXAMPLE 39
1.5.1. Understanding the code 39
1.5.2. How it works 46
1.6. A COMPLETE SEAM APPLICATION: THE HOTEL BOOKING EXAMPLE 47
1.6.1. Introduction 47
1.6.2. Overview of the booking example 49
1.6.3. Understanding Seam conversations 49
1.6.4. The Seam Debug Page 57
1.7. NESTED CONVERSATIONS: EXTENDING THE HOTEL BOOKING EXAMPLE 57
1.7.1. Introduction 58
1.7.2. Understanding Nested Conversations 59
1.8. A COMPLETE APPLICATION FEATURING SEAM AND JBPM: THE DVD STORE EXAMPLE 65
1.9. BOOKMARKABLE URLS WITH THE BLOG EXAMPLE 66
1.9.1. Using "pull"-style MVC 67
1.9.2. Bookmarkable search results page 69
1.9.3. Using "push"-style MVC in a RESTful application 71

CHAPTER 2. MIGRATION **75**
2.1. MIGRATING FROM SEAM 1.2.X TO SEAM 2.0 75
2.1.1. Migrating to JavaServer Faces 1.2 75
2.1.2. Code Migration 76
2.1.3. Migrating components.xml 77
2.1.4. Migrating to Embedded JBoss 78
2.1.5. Migrating to jBPM 3.2 78

2.1.6. Migrating to RichFaces 3.1 78
2.1.7. Changes to Components 79
2.2. MIGRATING FROM SEAM 2.0 TO SEAM 2.1 OR 2.2 80
2.2.1. Changes to dependency jar names 80
2.2.2. Changes to Components 83

CHAPTER 3. GETTING STARTED WITH SEAM-GEN 88
3.1. BEFORE YOU START 88
3.2. SETTING UP A NEW PROJECT 88
3.3. CREATING A NEW ACTION 94
3.4. CREATING A FORM WITH AN ACTION 95
3.5. GENERATING AN APPLICATION FROM AN EXISTING DATABASE 96
3.6. GENERATING AN APPLICATION FROM EXISTING JPA/EJB3 ENTITIES 96
3.7. DEPLOYING THE APPLICATION AS AN EAR 96
3.8. SEAM AND INCREMENTAL HOT DEPLOYMENT 97

CHAPTER 4. GETTING STARTED WITH JBOSS DEVELOPER STUDIO 99
4.1. HOT DEPLOYMENT WITH JBOSS DEVELOPER STUDIO 99

CHAPTER 5. THE CONTEXTUAL COMPONENT MODEL 100
5.1. SEAM CONTEXTS 100
5.1.1. Stateless context 100
5.1.2. Event context 100
5.1.3. Page context 101
5.1.4. Conversation context 101
5.1.5. Session context 101
5.1.6. Business process context 101
5.1.7. Application context 102
5.1.8. Context variables 102
5.1.9. Context search priority 102
5.1.10. Concurrency model 102
5.2. SEAM COMPONENTS 103
5.2.1. Stateless session beans 103
5.2.2. Stateful session beans 103
5.2.3. Entity beans 104
5.2.4. JavaBeans 104
5.2.5. Message-driven beans 105
5.2.6. Interception 105
5.2.7. Component names 106
5.2.8. Defining the component scope 107
5.2.9. Components with multiple roles 107
5.2.10. Built-in components 107
5.3. BIJECTION 108
5.4. LIFE CYCLE METHODS 110
5.5. CONDITIONAL INSTALLATION 110
5.6. LOGGING 112
5.7. THE MUTABLE INTERFACE AND @READONLY 112
5.8. FACTORY AND MANAGER COMPONENTS 114

CHAPTER 6. CONFIGURING SEAM COMPONENTS 116
6.1. CONFIGURING COMPONENTS VIA PROPERTY SETTINGS 116
6.2. CONFIGURING COMPONENTS VIA COMPONENTS.XML 116
6.3. FINE-GRAINED CONFIGURATION FILES 119
6.4. CONFIGURABLE PROPERTY TYPES 120

6.5. USING XML NAMESPACES 122

CHAPTER 7. EVENTS, INTERCEPTORS AND EXCEPTION HANDLING 125
7.1. SEAM EVENTS 125
7.2. PAGE ACTIONS 125
7.3. PAGE PARAMETERS 127
7.3.1. Mapping request parameters to the model 127
7.4. PROPAGATING REQUEST PARAMETERS 127
7.5. URL REWRITING WITH PAGE PARAMETERS 128
7.6. CONVERSION AND VALIDATION 129
7.7. NAVIGATION 130
7.8. FINE-GRAINED FILES FOR DEFINING NAVIGATION, PAGE ACTIONS AND PARAMETERS 132
7.9. COMPONENT-DRIVEN EVENTS 132
7.10. CONTEXTUAL EVENTS 134
7.11. SEAM INTERCEPTORS 136
7.12. MANAGING EXCEPTIONS 137
7.12.1. Exceptions and transactions 138
7.12.2. Enabling Seam exception handling 138
7.12.3. Using annotations for exception handling 138
7.12.4. Using XML for exception handling 139
7.12.4.1. Suppressing exception logging 140
7.12.5. Some common exceptions 140

CHAPTER 8. CONVERSATIONS AND WORKSPACE MANAGEMENT 142
8.1. SEAM'S CONVERSATION MODEL 142
8.2. NESTED CONVERSATIONS 144
8.3. STARTING CONVERSATIONS WITH GET REQUESTS 145
8.4. REQUIRING A LONG-RUNNING CONVERSATION 146
8.5. USING <S:LINK> AND <S:BUTTON> 146
8.6. SUCCESS MESSAGES 148
8.7. NATURAL CONVERSATION IDS 148
8.8. CREATING A NATURAL CONVERSATION 149
8.9. REDIRECTING TO A NATURAL CONVERSATION 149
8.10. WORKSPACE MANAGEMENT 150
8.10.1. Workspace management and JSF navigation 150
8.10.2. Workspace management and jPDL pageflow 151
8.10.3. The conversation switcher 151
8.10.4. The conversation list 152
8.10.5. Breadcrumbs 153
8.11. CONVERSATIONAL COMPONENTS AND JSF COMPONENT BINDINGS 153
8.12. CONCURRENT CALLS TO CONVERSATIONAL COMPONENTS 154
8.12.1. How should we design our conversational AJAX application? 154
8.12.2. Dealing with errors 155
8.12.3. RichFaces (Ajax4jsf) 156

CHAPTER 9. PAGEFLOWS AND BUSINESS PROCESSES 158
9.1. PAGEFLOW IN SEAM 158
9.1.1. The two navigation models 158
9.1.2. Seam and the back button 161
9.2. USING JPDL PAGEFLOWS 162
9.2.1. Installing pageflows 162
9.2.2. Starting pageflows 163
9.2.3. Page nodes and transitions 163
9.2.4. Controlling the flow 165

9.2.5. Ending the flow 165
9.2.6. Pageflow composition 165
9.3. BUSINESS PROCESS MANAGEMENT IN SEAM 165
9.4. USING JPDL BUSINESS PROCESS DEFINITIONS 166
9.4.1. Installing process definitions 166
9.4.2. Initializing actor IDs 167
9.4.3. Initiating a business process 167
9.4.4. Task assignment 167
9.4.5. Task lists 168
9.4.6. Performing a task 168

CHAPTER 10. SEAM AND OBJECT/RELATIONAL MAPPING 170
10.1. INTRODUCTION 170
10.2. SEAM MANAGED TRANSACTIONS 170
10.2.1. Disabling Seam-managed transactions 171
10.2.2. Configuring a Seam transaction manager 171
10.2.3. Transaction synchronization 172
10.3. SEAM-MANAGED PERSISTENCE CONTEXTS 172
10.3.1. Using a Seam-managed persistence context with JPA 173
10.3.2. Using a Seam-managed Hibernate session 173
10.3.3. Seam-managed persistence contexts and atomic conversations 174
10.4. USING THE JPA "DELEGATE" 175
10.5. USING EL IN EJB-QL/HQL 176
10.6. USING HIBERNATE FILTERS 176

CHAPTER 11. JSF FORM VALIDATION IN SEAM 178

CHAPTER 12. GROOVY INTEGRATION 183
12.1. GROOVY INTRODUCTION 183
12.2. WRITING SEAM APPLICATIONS IN GROOVY 183
12.2.1. Writing Groovy components 183
12.2.1.1. Entity 183
12.2.2. Seam component 184
12.2.3. seam-gen 185
12.3. DEPLOYMENT 185
12.3.1. Deploying Groovy code 185
12.3.2. Native .groovy file deployment at development time 185
12.3.3. seam-gen 185

CHAPTER 13. THE SEAM APPLICATION FRAMEWORK 186
13.1. INTRODUCTION 186
13.2. HOME OBJECTS 187
13.3. QUERY OBJECTS 191
13.4. CONTROLLER OBJECTS 193

CHAPTER 14. SEAM AND JBOSS RULES 195
14.1. INSTALLING RULES 195
14.2. USING RULES FROM A SEAM COMPONENT 197
14.3. USING RULES FROM A JBPM PROCESS DEFINITION 197

CHAPTER 15. SECURITY 199
15.1. OVERVIEW 199
15.2. DISABLING SECURITY 199
15.3. AUTHENTICATION 199
15.3.1. Configuring an Authenticator component 199

15.3.2. Writing an authentication method 200
15.3.2.1. Identity.addRole() 201
15.3.2.2. Writing an event observer for security-related events 202
15.3.3. Writing a login form 202
15.3.4. Configuration Summary 203
15.3.5. Remember Me 203
15.3.5.1. Token-based Remember Me Authentication 204
15.3.6. Handling Security Exceptions 206
15.3.7. Login Redirection 207
15.3.8. HTTP Authentication 207
15.3.8.1. Writing a Digest Authenticator 208
15.3.9. Advanced Authentication Features 208
15.3.9.1. Using your container's JAAS configuration 208
15.4. IDENTITY MANAGEMENT 208
15.4.1. Configuring IdentityManager 209
15.4.2. JpaIdentityStore 209
15.4.2.1. Configuring JpaIdentityStore 209
15.4.2.2. Configuring the Entities 210
15.4.2.3. Entity Bean Examples 211
15.4.2.3.1. Minimal schema example 211
15.4.2.3.2. Complex Schema Example 212
15.4.2.4. JpaIdentityStore Events 214
15.4.2.4.1. JpaIdentityStore.EVENT_PRE_PERSIST_USER 214
15.4.2.4.2. JpaIdentityStore.EVENT_USER_CREATED 214
15.4.2.4.3. JpaIdentityStore.EVENT_USER_AUTHENTICATED 214
15.4.3. LdapIdentityStore 214
15.4.3.1. Configuring LdapIdentityStore 214
15.4.3.2. LdapIdentityStore Configuration Example 217
15.4.4. Writing your own IdentityStore 217
15.4.5. Authentication with Identity Management 218
15.4.6. Using IdentityManager 218
15.5. ERROR MESSAGES 221
15.6. AUTHORIZATION 222
15.6.1. Core concepts 222
15.6.1.1. What is a role? 222
15.6.1.2. What is a permission? 223
15.6.2. Securing components 223
15.6.2.1. The @Restrict annotation 223
15.6.2.2. Inline restrictions 224
15.6.3. Security in the user interface 225
15.6.4. Securing pages 226
15.6.5. Securing Entities 226
15.6.5.1. Entity security with JPA 228
15.6.5.2. Entity security with a Managed Hibernate Session 228
15.6.6. Typesafe Permission Annotations 228
15.6.7. Typesafe Role Annotations 229
15.6.8. The Permission Authorization Model 230
15.6.8.1. PermissionResolver 230
15.6.8.1.1. Writing your own PermissionResolver 231
15.6.8.2. ResolverChain 231
15.6.9. RuleBasedPermissionResolver 232
15.6.9.1. Requirements 232
15.6.9.2. Configuration 233

15.6.9.3. Writing Security Rules 233
15.6.9.4. Non-String permission targets 235
15.6.9.5. Wildcard permission checks 235
15.6.10. PersistentPermissionResolver 236
15.6.10.1. Configuration 236
15.6.10.2. Permission Stores 236
15.6.10.3. JpaPermissionStore 237
15.6.10.3.1. Permission annotations 238
15.6.10.3.2. Example Entity 239
15.6.10.3.3. Class-specific Permission Configuration 240
15.6.10.3.4. Permission masks 241
15.6.10.3.5. Identifier Policy 241
15.6.10.3.6. ClassIdentifierStrategy 242
15.6.10.3.7. EntityIdentifierStrategy 242
15.7. PERMISSION MANAGEMENT 243
15.7.1. PermissionManager 243
15.7.2. Permission checks for PermissionManager operations 244
15.8. SSL SECURITY 245
15.8.1. Overriding the default ports 246
15.9. CAPTCHA 246
15.9.1. Configuring the CAPTCHA Servlet 246
15.9.2. Adding a CAPTCHA to a form 246
15.9.3. Customizing the CAPTCHA algorithm 247
15.10. SECURITY EVENTS 247
15.11. RUN AS 248
15.12. EXTENDING THE IDENTITY COMPONENT 248
15.13. OPENID 249
15.13.1. Configuring OpenID 250
15.13.2. Presenting an OpenIdLogin form 250
15.13.3. Logging in immediately 251
15.13.4. Deferring log in 251
15.13.5. Logging out 251

CHAPTER 16. INTERNATIONALIZATION, LOCALIZATION AND THEMES 252
16.1. INTERNATIONALIZING YOUR APPLICATION 252
16.1.1. Application server configuration 252
16.1.2. Translated application strings 252
16.1.3. Other encoding settings 253
16.2. LOCALES 253
16.3. LABELS 254
16.3.1. Defining labels 254
16.3.2. Displaying labels 255
16.3.3. Faces messages 255
16.4. TIMEZONES 256
16.5. THEMES 256
16.6. PERSISTING LOCALE AND THEME PREFERENCES VIA COOKIES 257

CHAPTER 17. SEAM TEXT 258
17.1. BASIC FORMATTING 258
17.2. ENTERING CODE AND TEXT WITH SPECIAL CHARACTERS 259
17.3. LINKS 260
17.4. ENTERING HTML 260
17.5. USING THE SEAMTEXTPARSER 261

CHAPTER 18. ITEXT PDF GENERATION **263**
18.1. USING PDF SUPPORT 263
18.1.1. Creating a document 263
18.1.2. Basic Text Elements 264
18.1.3. Headers and Footers 269
18.1.4. Chapters and Sections 270
18.1.5. Lists 271
18.1.6. Tables 273
18.1.7. Document Constants 275
18.1.7.1. Color Values 276
18.1.7.2. Alignment Values 276
18.2. CHARTING 276
18.3. BAR CODES 283
18.4. FILL-IN-FORMS 284
18.5. RENDERING SWING/AWT COMPONENTS 285
18.6. CONFIGURING ITEXT 286
18.7. FURTHER DOCUMENTATION 287

CHAPTER 19. THE MICROSOFT® EXCEL® SPREADSHEET APPLICATION **288**
19.1. MICROSOFT EXCEL SUPPORT 288
19.2. CREATING A SIMPLE WORKBOOK 288
19.3. WORKBOOKS 289
19.4. WORKSHEETS 291
19.5. COLUMNS 294
19.6. CELLS 294
19.6.1. Validation 295
19.6.2. Format masks 298
19.6.2.1. Number masks 298
19.6.2.2. Date masks 298
19.7. FORMULAS 298
19.8. IMAGES 299
19.9. HYPERLINKS 300
19.10. HEADERS AND FOOTERS 300
19.11. PRINT AREAS AND TITLES 302
19.12. WORKSHEET COMMANDS 303
19.12.1. Grouping 303
19.12.2. Page breaks 304
19.12.3. Merging 305
19.13. DATATABLE EXPORTER 306
19.14. FONTS AND LAYOUT 306
19.14.1. Stylesheet links 306
19.14.2. Fonts 307
19.14.3. Borders 307
19.14.4. Background 308
19.14.5. Column settings 308
19.14.6. Cell settings 309
19.14.7. The datatable exporter 309
19.14.8. Limitations 309
19.15. INTERNATIONALIZATION 309
19.16. LINKS AND FURTHER DOCUMENTATION 310

CHAPTER 20. EMAIL **311**
20.1. CREATING A MESSAGE 311

20.1.1. Attachments 312
20.1.2. HTML/Text alternative part 313
20.1.3. Multiple recipients 313
20.1.4. Multiple messages 313
20.1.5. Templating 313
20.1.6. Internationalization 314
20.1.7. Other Headers 314
20.2. RECEIVING EMAILS 315
20.3. CONFIGURATION 315
20.3.1. mailSession 316
20.3.1.1. JNDI look up in EAP 316
20.3.1.2. Seam-configured Session 316
20.4. TAGS 316

CHAPTER 21. ASYNCHRONICITY AND MESSAGING 319
21.1. ASYNCHRONICITY 319
21.1.1. Asynchronous methods 319
21.1.2. Asynchronous methods with the Quartz Dispatcher 321
21.1.3. Asynchronous events 323
21.1.4. Handling exceptions from asynchronous calls 324
21.2. MESSAGING IN SEAM 324
21.2.1. Configuration 324
21.2.2. Sending messages 325
21.2.3. Receiving messages using a message-driven bean 326
21.2.4. Receiving messages in the client 327

CHAPTER 22. CACHING 328
22.1. USING CACHING IN SEAM 328
22.2. PAGE FRAGMENT CACHING 330

CHAPTER 23. WEB SERVICES 331
23.1. CONFIGURATION AND PACKAGING 331
23.2. CONVERSATIONAL WEB SERVICES 331
23.2.1. A Recommended Strategy 332
23.3. AN EXAMPLE WEB SERVICE 333
23.4. RESTFUL HTTP WEB SERVICES WITH RESTEASY 334
23.4.1. RESTEasy configuration and request serving 334
23.4.2. Resources and providers as Seam components 337
23.4.3. Securing resources 339
23.4.4. Mapping exceptions to HTTP responses 339
23.4.5. Exposing entities via RESTful API 340
23.4.5.1. ResourceQuery 340
23.4.5.2. ResourceHome 341
23.4.6. Testing resources and providers 342

CHAPTER 24. REMOTING 344
24.1. CONFIGURATION 344
24.2. THE SEAM OBJECT 345
24.2.1. A Hello World example 345
24.2.2. Seam.Component 346
24.2.2.1. Seam.Component.newInstance() 346
24.2.2.2. Seam.Component.getInstance() 347
24.2.2.3. Seam.Component.getComponentName() 347
24.2.3. Seam.Remoting 348

24.2.3.1. Seam.Remoting.createType() 348
24.2.3.2. Seam.Remoting.getTypeName() 348
24.3. EVALUATING EL EXPRESSIONS 348
24.4. CLIENT INTERFACES 348
24.5. THE CONTEXT 349
24.5.1. Setting and reading the Conversation ID 349
24.5.2. Remote calls within the current conversation scope 349
24.6. BATCH REQUESTS 349
24.7. WORKING WITH DATA TYPES 350
24.7.1. Primitives / Basic Types 350
24.7.1.1. String 350
24.7.1.2. Number 350
24.7.1.3. Boolean 350
24.7.2. JavaBeans 350
24.7.3. Dates and Times 351
24.7.4. Enums 351
24.7.5. Collections 351
24.7.5.1. Bags 351
24.7.5.2. Maps 351
24.8. DEBUGGING 352
24.9. HANDLING EXCEPTIONS 352
24.10. THE LOADING MESSAGE 352
24.10.1. Changing the message 352
24.10.2. Hiding the loading message 353
24.10.3. A Custom Loading Indicator 353
24.11. CONTROLLING WHAT DATA IS RETURNED 353
24.11.1. Constraining normal fields 354
24.11.2. Constraining Maps and Collections 354
24.11.3. Constraining objects of a specific type 354
24.11.4. Combining Constraints 354
24.12. TRANSACTIONAL REQUESTS 355
24.13. JMS MESSAGING 355
24.13.1. Configuration 355
24.13.2. Subscribing to a JMS Topic 355
24.13.3. Unsubscribing from a Topic 355
24.13.4. Tuning the Polling Process 356

CHAPTER 25. SEAM AND THE GOOGLE WEB TOOLKIT 357
25.1. CONFIGURATION 357
25.2. PREPARING YOUR COMPONENT 357
25.3. HOOKING UP A GWT WIDGET TO THE SEAM COMPONENT 358
25.4. GWT ANT TARGETS 359

CHAPTER 26. SPRING FRAMEWORK INTEGRATION 361
26.1. INJECTING SEAM COMPONENTS INTO SPRING BEANS 361
26.2. INJECTING SPRING BEANS INTO SEAM COMPONENTS 362
26.3. MAKING A SPRING BEAN INTO A SEAM COMPONENT 363
26.4. SEAM-SCOPED SPRING BEANS 363
26.5. USING SPRING PLATFORMTRANSACTIONMANAGEMENT 364
26.6. USING A SEAM-MANAGED PERSISTENCE CONTEXT IN SPRING 364
26.7. USING A SEAM-MANAGED HIBERNATE SESSION IN SPRING 366
26.8. SPRING APPLICATION CONTEXT AS A SEAM COMPONENT 366
26.9. USING A SPRING TASKEXECUTOR FOR @ASYNCHRONOUS 367

CHAPTER 27. HIBERNATE SEARCH 368
27.1. INTRODUCTION 368
27.2. CONFIGURATION 368
27.3. USAGE 369

CHAPTER 28. CONFIGURING SEAM AND PACKAGING SEAM APPLICATIONS 371
28.1. BASIC SEAM CONFIGURATION 371
28.1.1. Integrating Seam with JSF and your servlet container 371
28.1.2. Using Facelets 372
28.1.3. Seam Resource Servlet 372
28.1.4. Seam Servlet filters 372
28.1.4.1. Exception handling 373
28.1.4.2. Conversation propagation with redirects 373
28.1.4.3. URL rewriting 373
28.1.4.4. Multipart form submissions 374
28.1.4.5. Character encoding 374
28.1.4.6. RichFaces 374
28.1.4.7. Identity Logging 375
28.1.4.8. Context management for custom servlets 375
28.1.4.9. Adding custom filters 375
28.1.5. Integrating Seam with your EJB container 376
28.1.6. Remember 379
28.2. USING ALTERNATE JPA PROVIDERS 379
28.3. CONFIGURING SEAM IN JAVA EE 5 380
28.3.1. Packaging 380
28.4. CONFIGURING SEAM IN J2EE 381
28.4.1. Boostrapping Hibernate in Seam 381
28.4.2. Boostrapping JPA in Seam 382
28.4.3. Packaging 382
28.5. CONFIGURING SEAM IN JAVA SE, WITHOUT JBOSS EMBEDDED 383
28.6. CONFIGURING SEAM IN JAVA SE, WITH JBOSS EMBEDDED 383
28.6.1. Packaging 383
28.7. CONFIGURING JBPM IN SEAM 384
28.7.1. Packaging 385
28.8. CONFIGURING SFSB AND SESSION TIMEOUTS IN EAP 386
28.9. RUNNING SEAM IN A PORTLET 386
28.10. DEPLOYING CUSTOM RESOURCES 387

CHAPTER 29. SEAM ANNOTATIONS 390
29.1. ANNOTATIONS FOR COMPONENT DEFINITION 390
29.2. ANNOTATIONS FOR BIJECTION 393
29.3. ANNOTATIONS FOR COMPONENT LIFE CYCLE METHODS 395
29.4. ANNOTATIONS FOR CONTEXT DEMARCATION 396
29.5. ANNOTATIONS FOR USE WITH SEAM JAVABEAN COMPONENTS IN A J2EE ENVIRONMENT 399
29.6. ANNOTATIONS FOR EXCEPTIONS 400
29.7. ANNOTATIONS FOR SEAM REMOTING 401
29.8. ANNOTATIONS FOR SEAM INTERCEPTORS 401
29.9. ANNOTATIONS FOR ASYNCHRONICITY 401
29.10. ANNOTATIONS FOR USE WITH JSF 402
29.10.1. Annotations for use with dataTable 403
29.11. META-ANNOTATIONS FOR DATABINDING 404
29.12. ANNOTATIONS FOR PACKAGING 404
29.13. ANNOTATIONS FOR INTEGRATING WITH THE SERVLET CONTAINER 404

CHAPTER 30. BUILT-IN SEAM COMPONENTS **406**
30.1. CONTEXT INJECTION COMPONENTS 406
30.2. JSF-RELATED COMPONENTS 406
30.3. UTILITY COMPONENTS 407
30.4. COMPONENTS FOR INTERNATIONALIZATION AND THEMES 408
30.5. COMPONENTS FOR CONTROLLING CONVERSATIONS 410
30.6. JBPM-RELATED COMPONENTS 411
30.7. SECURITY-RELATED COMPONENTS 413
30.8. JMS-RELATED COMPONENTS 413
30.9. MAIL-RELATED COMPONENTS 413
30.10. INFRASTRUCTURAL COMPONENTS 414
30.11. MISCELLANEOUS COMPONENTS 416
30.12. SPECIAL COMPONENTS 416

CHAPTER 31. SEAM JSF CONTROLS **419**
31.1. TAGS 419
31.1.1. Navigation Controls 419
31.1.1.1. <s:button> 419
31.1.1.2. <s:conversationId> 419
31.1.1.3. <s:taskId> 420
31.1.1.4. <s:link> 420
31.1.1.5. <s:conversationPropagation> 420
31.1.1.6. <s:defaultAction> 421
31.1.2. Converters and Validators 421
31.1.2.1. <s:convertDateTime> 421
31.1.2.2. <s:convertEntity> 422
31.1.2.3. <s:convertEnum> 423
31.1.2.4. <s:convertAtomicBoolean> 423
31.1.2.5. <s:convertAtomicInteger> 424
31.1.2.6. <s:convertAtomicLong> 424
31.1.2.7. <s:validateEquality> 424
31.1.2.8. <s:validate> 425
31.1.2.9. <s:validateAll> 425
31.1.3. Formatting 426
31.1.3.1. <s:decorate> 426
31.1.3.2. <s:div> 427
31.1.3.3. <s:span> 427
31.1.3.4. <s:fragment> 427
31.1.3.5. <s:label> 428
31.1.3.6. <s:message> 428
31.1.4. Seam Text 428
31.1.4.1. <s:validateFormattedText> 428
31.1.4.2. <s:formattedText> 428
31.1.5. Form support 429
31.1.5.1. <s:token> 429
31.1.5.2. <s:enumItem> 430
31.1.5.3. <s:selectItems> 430
31.1.5.4. <s:fileUpload> 431
31.1.6. Other 432
31.1.6.1. <s:cache> 432
31.1.6.2. <s:resource> 433
31.1.6.3. <s:download> 434
31.1.6.4. <s:graphicImage> 434

31.1.6.5. <s:remote> 435
31.2. ANNOTATIONS 435

CHAPTER 32. JBOSS EL 437
32.1. PARAMETERIZED EXPRESSIONS 437
32.1.1. Usage 437
32.1.2. Limitations and Hints 438
32.2. PROJECTION 439

CHAPTER 33. CLUSTERING AND EJB PASSIVATION 440
33.1. CLUSTERING 440
33.1.1. Programming for clustering 440
33.1.2. Deploying a Seam application to a EAP cluster with session replication 441
33.1.3. Tutorial 441
33.1.4. Validating the distributable services of an application running in a EAP cluster 442
33.2. EJB PASSIVATION AND THE MANAGEDENTITYINTERCEPTOR 443
33.2.1. The friction between passivation and persistence 444
33.2.2. Case #1: Surviving EJB passivation 444
33.2.3. Case #2: Surviving HTTP session replication 445

CHAPTER 34. PERFORMANCE TUNING 446
34.1. BYPASSING INTERCEPTORS 446

CHAPTER 35. TESTING SEAM APPLICATIONS 447
35.1. UNIT TESTING SEAM COMPONENTS 447
35.2. INTEGRATION TESTING SEAM COMPONENTS 448
35.2.1. Using mocks in integration tests 448
35.3. INTEGRATION TESTING SEAM APPLICATION USER INTERACTIONS 449
35.3.1. Configuration 451
35.3.2. Using SeamTest with another test framework 452
35.3.3. Integration Testing with Mock Data 453
35.3.4. Integration Testing Seam Mail 454

CHAPTER 36. SEAM TOOLS 456
36.1. JBPM DESIGNER AND VIEWER 456
36.1.1. Business process designer 456
36.1.2. Pageflow viewer 456

CHAPTER 37. DEPENDENCIES 458
37.1. JAVA DEVELOPMENT KIT DEPENDENCIES 458
37.1.1. Sun's JDK 6 Considerations 458
37.2. PROJECT DEPENDENCIES 458
37.2.1. Core 458
37.2.2. RichFaces 459
37.2.3. Seam Mail 459
37.2.4. Seam PDF 460
37.2.5. Seam Microsoft®Excel® 460
37.2.6. JBoss Rules 460
37.2.7. JBPM 461
37.2.8. GWT 461
37.2.9. Spring 461
37.2.10. Groovy 461

APPENDIX A. REVISION HISTORY 463

CHAPTER 1. SEAM TUTORIAL

1.1. USING THE SEAM EXAMPLES

Seam provides a number of example applications which demonstrate how to use a variety of Seam's features. This tutorial will guide you through a few of those examples to help you get started learning Seam. The Seam examples are located in the **`examples`** subdirectory of the Seam distribution. The first example, on registration, is in the **`examples/registration`** directory.

Each example has the same directory structure:

- The **`view`** directory contains view-related files such as web page templates, images and stylesheets.
- The **`resources`** directory contains deployment descriptors and other configuration files.
- The **`src`** directory contains the application source code.

Note that all examples are built and run from the Ant **`build.xml`**, so you will need a recent version of Ant installed before you get started.

1.1.1. Running the examples on JBoss Enterprise Application Platform

The examples are configured for use on JBoss Enterprise Application Platform. You will need to set **`jboss.home`**, in the shared **`build.properties`** file (in the root folder of your Seam installation) to the location of your JBoss Enterprise Application Platform installation.

Once you have set the location of JBoss Enterprise Application Platform and started the application server, you can build and deploy any example by typing **`ant explode`** in that example's directory. Any example that is packaged as an EAR (Enterprise Archive) deploys to a URL like **`/seam-`*`example`***, where *example* is the name of the example folder, with one exception: if the example folder begins with "seam", the prefix "seam" is omitted. For instance, if JBoss Enterprise Application Platform is running on port 8080, the URL for the Registration example is **`http://localhost:8080/seam-registration/`**, whereas the URL for the SeamSpace example is **`http://localhost:8080/seam-space/`**.

If, on the other hand, the example is packaged as a WAR, then it deploys to a URL like **`/jboss-seam-`*`example`***.

NOTE

Several of the examples — groovybooking, hibernate, jpa, and spring — can only be deployed as a WAR.

1.1.2. Running the example tests

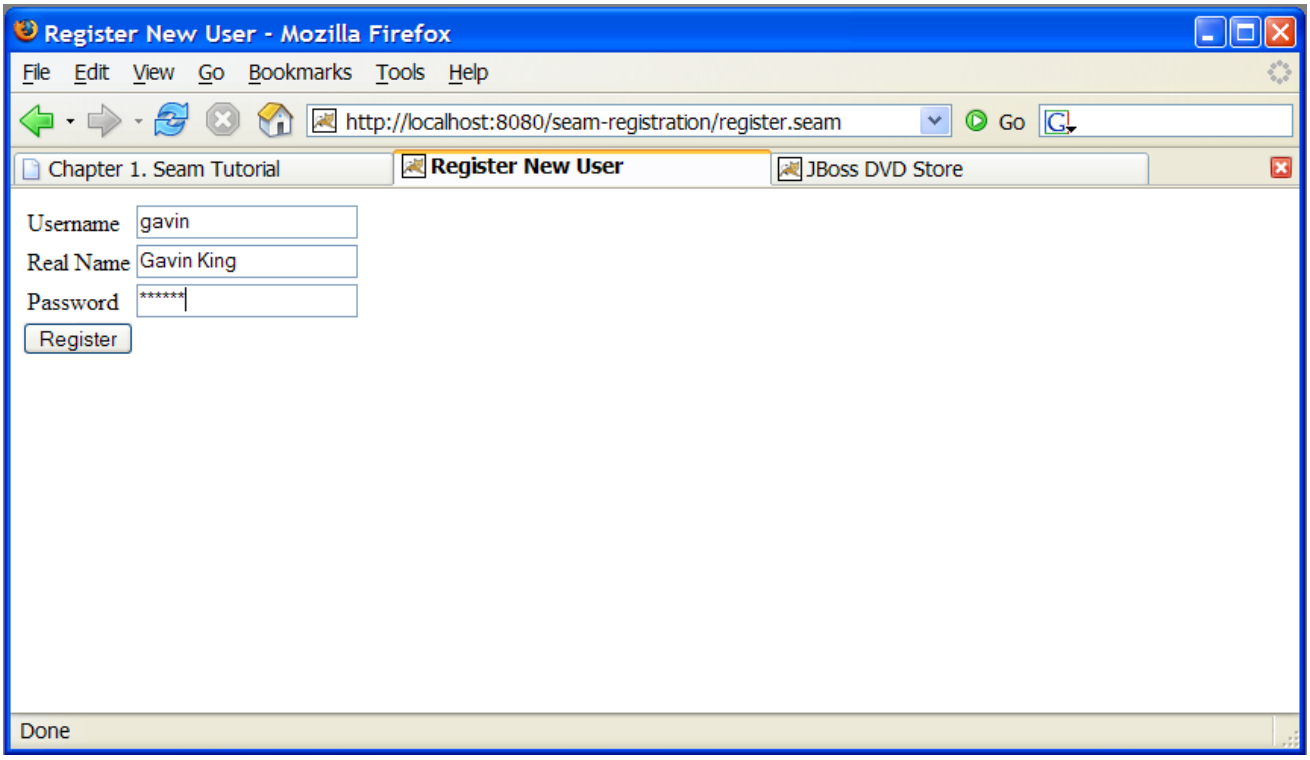
Most of the examples come with a suite of TestNG integration tests. The easiest way to run the tests is to run **`ant test`**.

It is also possible to run the tests inside your IDE using the TestNG plug-in. This requires you to run ant test before running or debugging Seam test cases in JBoss Developer Studio. Consult the **`readme.txt`** in the examples directory of the Seam distribution for more information.

1.2. YOUR FIRST SEAM APPLICATION: THE REGISTRATION EXAMPLE

The registration example is a simple application that lets a new user store their username, real name, and password in the database. This example uses only basic functions to demonstrate the use of an EJB3 session bean as a JSF action listener, and the basic configuration of Seam.

The start page displays a basic form with three input fields. Filling them in and submitting the form will save a user object in the database.



1.2.1. Understanding the code

The example is implemented with two Facelets templates: one entity bean, and one stateless session bean. This section will take you through the code in detail, starting from the base level.

1.2.1.1. The entity bean: `User.java`

We need an EJB entity bean for user data. This class defines *persistence* and *validation* declaratively, via annotations. It also requires some extra annotations to define the class as a Seam component.

Example 1.1. User.java

```
@Entity

@Name("user")

@Scope(SESSION)

@Table(name="users")
public class User implements Serializable
{
```

```
private static final long serialVersionUID =
    1881413500711441951L;
```

5

```
private String username;
private String password;
private String name;

public User(String name,
            String password,
            String username)
{
    this.name = name;
    this.password = password;
    this.username = username;
}
```

6

```
public User() {}
```

7

```
@NotNull @Length(min=5, max=15)
public String getPassword()
{
    return password;
}

public void setPassword(String password)
{
    this.password = password;
}

@NotNull
public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}
```

8

```
@Id @NotNull @Length(min=5, max=15)
public String getUsername()
{
    return username;
}

public void setUsername(String username)
{
    this.username = username;
```

```
    }

}
```

1	The EJB3 standard **@Entity** annotation indicates that the **User** class is an entity bean.
2	A Seam component needs a *component name* specified by the **@Name** annotation. This name must be unique within the Seam application. When JSF asks Seam to resolve a context variable with a name that is the same as a Seam component name, and the context variable is currently undefined (null), Seam will instantiate that component, and bind the new instance to the context variable. In this case, Seam will instantiate a **User** the first time JSF encounters a variable named **user**.
3	Whenever Seam instantiates a component, it binds the new instance to a context variable in the component's *default context*. The default context is specified using the **@Scope** annotation. The **User** bean is a session scoped component.
4	The EJB standard **@Table** annotation indicates that the **User** class is mapped to the **users** table.
5	**name**, **password** and **username** are the persistent attributes of the entity bean. All of our persistent attributes define accessor methods. These are needed when this component is used by JSF in the render response and update model values phases.
6	An empty constructor is required by both the EJB specification and Seam.
7	The **@NotNull** and **@Length** annotations are part of the Hibernate Validator framework. Seam integrates Hibernate Validator and lets you use it for data validation (even if you are not using Hibernate for persistence).
8	The EJB standard **@Id** annotation indicates the primary key attribute of the entity bean.

The most important things to notice in this example are the **@Name** and **@Scope** annotations. These annotations establish that this class is a Seam component.

In the next section, you will see that the properties of the **User** class are bound directly to JSF components and populated by JSF during the update model values phase. There is no glue code to copy data back and forth between the JSP pages and the entity bean domain model.

However, entity beans should not perform transaction management or database access, so this component should not be used as a JSF action listener. In this situation, a session bean is a better choice.

1.2.1.2. The stateless session bean class: `RegisterAction.java`

Most Seam applications use session beans as JSF action listeners, though you may also use JavaBeans.

We have exactly one JSF action in this application, and one session bean method attached to it. In this case, we will use a stateless session bean, since all the state associated with our action is held by the **User** bean.

The relevant code is shown below:

Example 1.2. RegisterAction.java

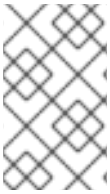
```
@Stateless
(1)
@Name("register")
public class RegisterAction implements Register
{
    @In
(2)
    private User user;

    @PersistenceContext
(3)
    private EntityManager em;

    @Logger
(4)
    private Log log;

    public String register()
(5)
    {
        List existing = em.createQuery("select username " +
                                       "from User " +
                                       "where username = #
                                  (6)
{user.username}")
            .getResultList();

        if (existing.size()==0)
            {
                em.persist(user);
                log.info("Registered new user #{user.username}");
(7)
                return "/registered.xhtml";
(8)
            }
        else
            {
                FacesMessages.instance().add("User #{user.username}
                          (9)
already exists");
                return null;
            }
```

```
    }

}
```

1	The EJB **@Stateless** annotation marks this class as a stateless session bean.
2	The **@In** annotation marks an attribute of the bean as injected by Seam. In this case, the attribute is injected from a context variable named **user** (the instance variable name).
3	The EJB standard **@PersistenceContext** annotation is used to inject the EJB3 entity manager.
4	The Seam **@Logger** annotation is used to inject the component's **Log** instance.
5	The action listener method uses the standard EJB3 **EntityManager** API to interact with the database, and returns the JSF outcome. Note that, since this is a session bean, a transaction is automatically begun when the **register()** method is called, and committed when it completes.
6	Notice that Seam lets you use a JSF EL expression inside EJB-QL. Under the covers, this results in an ordinary JPA **setParameter()** call on the standard JPA **Query** object.
7	The **Log** API lets us easily display templated log messages which can also make use of JSF EL expressions.
8	JSF action listener methods return a string-valued outcome that determines what page will be displayed next. A null outcome (or a void action listener method) redisplays the previous page. In plain JSF, it is normal to always use a JSF *navigation rule* to determine the JSF view id from the outcome. For complex applications this indirection is useful and a good practice. However, for very simple examples like this one, Seam lets you use the JSF view id as the outcome, eliminating the requirement for a navigation rule. *Note that when you use a view id as an outcome, Seam always performs a browser redirect.*
9	Seam provides a number of *built-in components* to help solve common problems. The **FacesMessages** component makes it easy to display templated error or success messages. (As of Seam 2.1, you can use **StatusMessages** instead to remove the semantic dependency on JSF). Built-in Seam components may be obtained by injection, or by calling the **instance()** method on the class of the built-in component.

Note that we did not explicitly specify a **@Scope** this time. Each Seam component type has a default scope, which will be used if scope is not explicitly specified. For stateless session beans, the default scope is the stateless context.

The session bean action listener performs the business and persistence logic for our mini-application. In a more complex application, a separate service layer might be necessary, but Seam allows you to implement your own strategies for application layering. You can make any application as simple, or as complex, as you want.



NOTE

This application is more complex than necessary for the sake of clear example code. All of the application code could have been eliminated by using Seam's application framework controllers.

1.2.1.3. The session bean local interface: `Register.java`

The session bean requires a local interface.

Example 1.3. Register.java

```
@Local
public interface Register
{
     public String register();
}
```

That's the end of the Java code. The next level to examine is the view.

1.2.1.4. The view: `register.xhtml` and `registered.xhtml`

The view pages for a Seam application can be implemented using any technology that supports JSF. This example was written with Facelets.

Example 1.4. register.xhtml

```
<?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:s="http://jboss.com/products/seam/taglib"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <title>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <s:validateAll>
          <h:panelGrid columns="2">
            Username: <h:inputText value="#{user.username}"
            required="true"/>
            Real Name: <h:inputText value="#{user.name}"
            required="true"/>
            Password: <h:inputSecret value="#{user.password}"
            required="true"/>
          </h:panelGrid>
        </s:validateAll>
        <h:messages/>
        <h:commandButton value="Register" action="#
{register.register}"/>
      </h:form>
    </f:view>
  </body>

</html>
```

The only Seam-specific tag here is **<s:validateAll>**. This JSF component tells JSF to validate all the contained input fields against the Hibernate Validator annotations specified on the entity bean.

Example 1.5. registered.xhtml

```
<?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:f="http://java.sun.com/jsf/core">

  <head>
    <title>Successfully Registered New User</title>
  </head>
  <body>
    <f:view>
      Welcome, #{user.name}, you are successfully
      registered as #{user.username}.
    </f:view>
  </body>

</html>
```

The above is a simple Facelets page, created with inline EL — it contains nothing specific to Seam.

1.2.1.5. The Seam component deployment descriptor: `components.xml`

Before looking at deployment descriptors, it is worth noting that Seam strongly values minimal configuration. These configuration files will be created for you when you create a Seam application, and there will rarely be any need to alter them. They are presented here solely to assist you in understanding the purpose and function of all of the example code.

If you have used Java frameworks previously, you will be used to declaring your component classes in an XML file. You have probably also noticed that as a project matures, these XML files tend to become unmanageable. Fortunately, Seam does not require application components to be accompanied by XML. Most Seam applications require only a small amount of XML, which does not tend to increase in size as projects expand.

However, it is often useful to be able to provide for some external configuration of some components, particularly the components that are built into Seam. The most flexible option, here, is to provide this configuration in a file called **components.xml**, located in the **WEB-INF** directory. The **components.xml** file can be used to tell Seam how to find our EJB components in JNDI:

Example 1.6. components.xml example

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
            "http://jboss.com/products/seam/core
             http://jboss.com/products/seam/core-2.2.xsd
```

```
            http://jboss.com/products/seam/components
            http://jboss.com/products/seam/components-2.2.xsd">

  <core:init jndi-pattern="@jndiPattern@"/>

</components>
```

The above code configures a property named **`jndiPattern`**, which belongs to a built-in Seam component named **`org.jboss.seam.core.init`**. The `@` symbols are used to direct the Ant build script to insert the correct JNDI pattern from the components.properties file when the application is deployed. You will learn more about this process in Section 6.2, “Configuring components via **`components.xml`**”.

1.2.1.6. The web deployment description: `web.xml`

The presentation layer for our mini-application will be deployed in a WAR, so a web deployment descriptor is required:

Example 1.7. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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-app_2_5.xsd"
         version="2.5">

  <listener>
    <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
  </listener>

  <context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>

  <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>*.seam</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

The above **web.xml** file configures both Seam and JSF. The configuration you see here changes very little between Seam applications.

1.2.1.7. The JSF configuration: faces-config.xml

Most Seam applications use JSF views as the presentation layer, so **faces-config.xml** is usually a requirement. In this case, Facelets is used to define our views, so we need to tell JSF to use Facelets as its templating engine.

Example 1.8. faces-config.xml

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

</faces-config>
```

Note that JSF managed bean declarations are unnecessary because the managed beans are annotated Seam components. In Seam applications, **faces-config.xml** is used much less often than in plain JSF. Here, we use it simply to enable Facelets (and not JSP) as the view handler.

Once you have set up all the basic descriptors, the only XML you need write to add functionality to a Seam application will be for orchestration: navigation rules or jBPM process definitions. Seam operates on the principle that *process flow* and *configuration data* are all that truly belongs in XML.

The above example does not require a navigation rule, since the view ID was embedded in our action code.

1.2.1.8. The EJB deployment descriptor: ejb-jar.xml

The **ejb-jar.xml** file integrates Seam with EJB3 by attaching the **SeamInterceptor** to all session beans in the archive.

Example 1.9. ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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/ejb-jar_3_0.xsd"
         version="3.0">
```

```
  <interceptors>
    <interceptor>
      <interceptor-class>
        org.jboss.seam.ejb.SeamInterceptor
      </interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        org.jboss.seam.ejb.SeamInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>

</ejb-jar>
```

1.2.1.9. The EJB persistence deployment descriptor: `persistence.xml`

The **`persistence.xml`** file directs the EJB persistence provider to the appropriate datasource, and contains some vendor-specific settings. In this case, it enables automatic schema export at start time.

Example 1.10. persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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"
             version="1.0">

  <persistence-unit name="userDatabase">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>

</persistence>
```

1.2.1.10. The EAR deployment descriptor: `application.xml`

Finally, since our application is deployed as an EAR, we also require a deployment descriptor.

Example 1.11. registration application

```
<?xml version="1.0" encoding="UTF-8"?>
<application 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/application_5.xsd"
             version="5">

  <display-name>Seam Registration</display-name>

  <module>
    <web>
      <web-uri>jboss-seam-registration.war</web-uri>
      <context-root>/seam-registration</context-root>
    </web>
  </module>
  <module>
    <ejb>jboss-seam-registration.jar</ejb>
  </module>
  <module>
    <ejb>jboss-seam.jar</ejb>
  </module>
  <module>
    <java>jboss-el.jar</java>
  </module>

</application>
```

This deployment descriptor links modules in the enterprise archive and binds the web application to the context root **`/seam-registration`**.

You have now seen all of the files in the application.

1.2.2. How it works

When the form is submitted, JSF asks Seam to resolve the variable named **`user`**. Since no value is yet bound to that name (in any Seam context), Seam instantiates the **`user`** component, and returns the resulting **`User`** entity bean instance to JSF after storing it in the Seam session context.

The form input values are now validated against the Hibernate Validator constraints specified on the **`User`** entity. If the constraints are violated, JSF redisplays the page. Otherwise, JSF binds the form input values to properties of the **`User`** entity bean.

Next, JSF asks Seam to resolve the variable named **`register`**. Seam uses the JNDI pattern mentioned earlier to locate the stateless session bean, wraps it as a Seam component, and returns it. Seam then presents this component to JSF and JSF invokes the **`register()`** action listener method.

Seam then intercepts the method call and injects the **`User`** entity from the Seam session context, before allowing the invocation to continue.

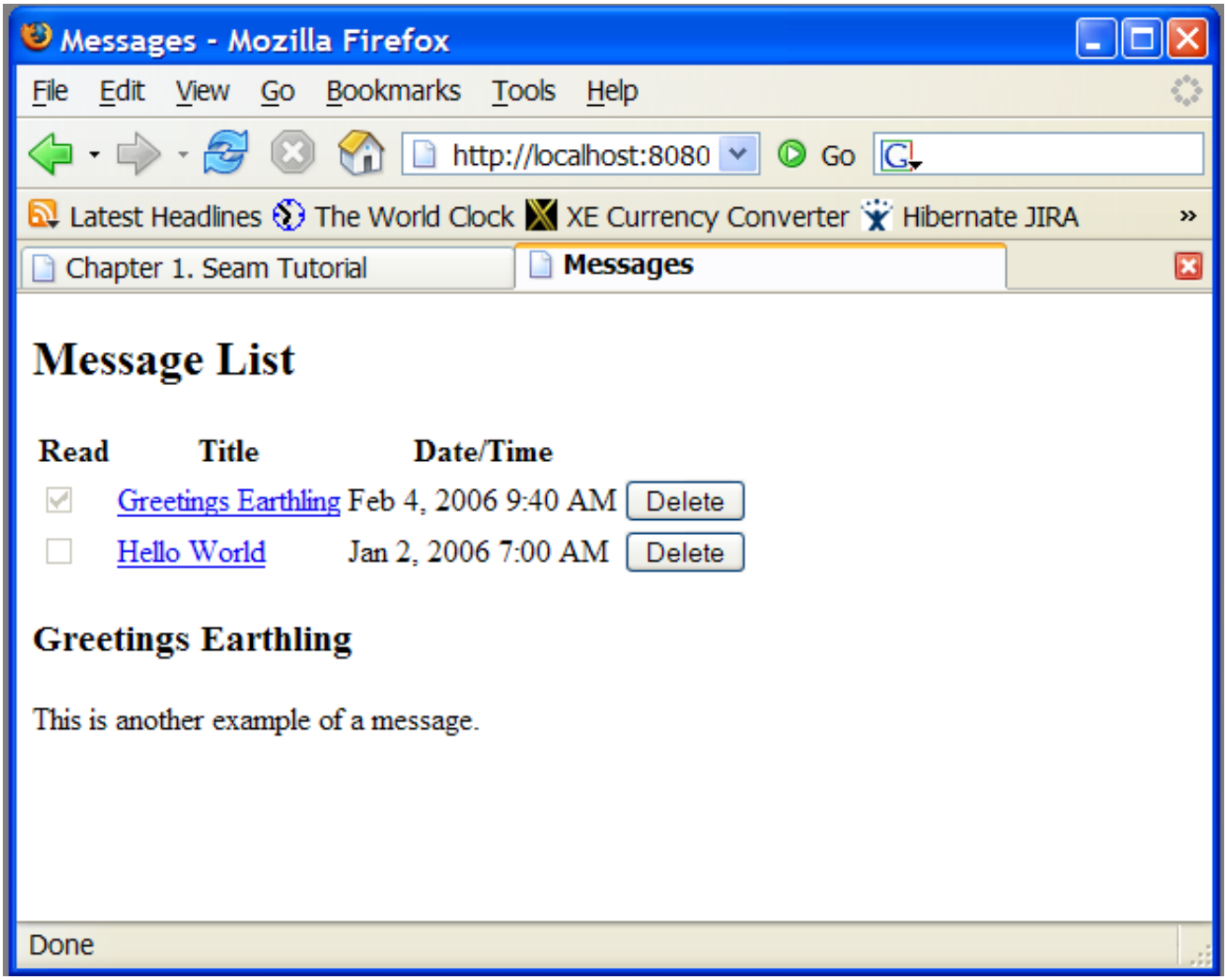
The **`register()`** method checks if a user with the entered username already exists. If so, an error message is queued with the **`FacesMessages`** component, and a null outcome is returned, causing a page redisplay. The **`FacesMessages`** component interpolates the JSF expression embedded in the

message string and adds a JSF **FacesMessage** to the view.

If no user with that username exists, the **"/registered.xhtml"** outcome triggers a browser redirect to the **registered.xhtml** page. When JSF comes to render the page, it asks Seam to resolve the variable named **user** and uses property values of the returned **User** entity from Seam's session scope.

1.3. CLICKABLE LISTS IN SEAM: THE MESSAGES EXAMPLE

Clickable lists of database search results are a vital part of any online application. Seam provides special functionality on top of JSF to make it easier to query data with EJB-QL or HQL, and display it as a clickable list using a JSF **<h:dataTable>**. The messages example demonstrates this functionality.



1.3.1. Understanding the code

The message list example has one entity bean (**Message**), one session bean (**MessageListBean**), and one JSP.

1.3.1.1. The entity bean: `Message.java`

The **Message** entity defines the title, text, date and time of a message, and a flag indicating whether the message has been read:

> **Example 1.12. Message.java**

```
@Entity
@Name("message")
@Scope(EVENT)
public class Message implements Serializable {
    private Long id;
    private String title;
    private String text;
    private boolean read;
    private Date datetime;

    @Id @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }

    @NotNull @Length(max=100)
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    @NotNull @Lob
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

    @NotNull
    public boolean isRead() {
        return read;
    }
    public void setRead(boolean read) {
        this.read = read;
    }

    @NotNull
    @Basic @Temporal(TemporalType.TIMESTAMP)
    public Date getDatetime() {
        return datetime;
    }
    public void setDatetime(Date datetime) {
        this.datetime = datetime;
    }
}
```

1.3.1.2. The stateful session bean: `MessageManagerBean.java`

As in the previous example, this example contains a session bean (**MessageManagerBean**) which defines the action listener methods for both of the buttons on our form. As in the previous example, one of the buttons selects a message from the list and displays that message; the other button deletes a message.

However, **MessageManagerBean** is also responsible for fetching the list of messages the first time we navigate to the message list page. There are various ways for users to navigate to the page, not all of which are preceded by a JSF action. (Navigating to the page from your favorites will not necessarily call the JSF action, for example.) Therefore, fetching the message list must take place in a Seam *factory method*, instead of in an action listener method.

We want to cache the list of messages in memory between server requests, so we will make this a stateful session bean.

Example 1.13. MessageManagerBean.java

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean
    implements Serializable, MessageManager
{
    @DataModel
```

(1)

```
    private List<Message> messageList;

    @DataModelSelection
```

(2)

```
    @Out(required=false)
```

(3)

```
    private Message message;

    @PersistenceContext(type=EXTENDED)
```

(4)

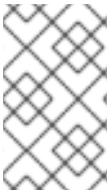
```
    private EntityManager em;

    @Factory("messageList")
```

(5)

```
    public void findMessages()
    {
        messageList = em.createQuery("select msg " +
                                     "from Message msg" +
                                     "order by msg.datetime desc")
                        .getResultList();
    }

    public void select()
```

```
    (6)
    {
        message.setRead(true);
    }

    public void delete()
    (7)
    {
        messageList.remove(message);
        em.remove(message);
        message=null;
    }

    @Remove
    (8)
    public void destroy() {}

}
```

(1)	The **@DataModel** annotation exposes an attributes of type **java.util.List** to the JSF page as an instance of **javax.faces.model.DataModel**. This allows us to use the list in a JSF **<h:dataTable>** with clickable links for each row. In this case, the **DataModel** is made available in a session context variable named **messageList**.
(2)	The **@DataModelSelection** annotation tells Seam to inject the **List** element that corresponded to the clicked link.
(3)	The **@Out** annotation then exposes the selected value directly to the page. So every time a row of the clickable list is selected, the **Message** is injected to the attribute of the stateful bean, and then subsequently *outjected* to the event context variable named **message**.
(4)	This stateful bean has an EJB3 *extended persistence context*. The messages retrieved in the query remain in the managed state as long as the bean exists, so any subsequent method calls to the stateful bean can update them without needing to make any explicit call to the **EntityManager**.
(5)	The first time we navigate to the JSP page, there will be no value in the **messageList** context variable. The **@Factory** annotation tells Seam to create an instance of **MessageManagerBean** and invoke the **findMessages()** method to initialize the value. We call **findMessages()** a *factory method* for **messages**.
(6)	The **select()** action listener method marks the selected **Message** as read, and updates it in the database.
(7)	The **delete()** action listener method removes the selected **Message** from the database.
(8)	All stateful session bean Seam components *must* define a parameterless method marked **@Remove** that Seam uses to remove the stateful bean when the Seam context ends, and clean up any server-side state.



NOTE

This is a session-scoped Seam component. It is associated with the user log in session, and all requests from a log in session share the same instance of the component. Session-scoped components are usually used sparingly in Seam applications.

1.3.1.3. The session bean local interface: `MessageManager.java`

All session beans have a business interface:

Example 1.14. MessageManager.java

```
@Local
public interface MessageManager {
    public void findMessages();
    public void select();
    public void delete();
    public void destroy();
}
```

From this point, local interfaces are no longer shown in these code examples. **`Components.xml`**, **`persistence.xml`**, **`web.xml`**, **`ejb-jar.xml`**, **`faces-config.xml`** and **`application.xml`** operate in a similar fashion to the previous example, and go directly to the JSP.

1.3.1.4. The view: `messages.jsp`

The JSP page is a straightforward use of the JSF **`<h:dataTable>`** component. Once again, these functions are not Seam-specific.

Example 1.15. messages.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
  <head>
    <title>Messages</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <h2>Message List</h2>
        <h:outputText value="No messages to display"
                      rendered="#{messageList.rowCount==0}"/>
        <h:dataTable var="msg" value="#{messageList}"
                     rendered="#{messageList.rowCount>0}">
          <h:column>
            <f:facet name="header">
              <h:outputText value="Read"/>
            </f:facet>
            <h:selectBooleanCheckbox value="#{msg.read}"
disabled="true"/>
          </h:column>
          <h:column>
```

```
            <f:facet name="header">
              <h:outputText value="Title"/>
            </f:facet>
            <h:commandLink value="#{msg.title}"
                           action="#{messageManager.select}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Date/Time"/>
            </f:facet>
            <h:outputText value="#{msg.datetime}">
              <f:convertDateTime type="both" dateStyle="medium"
                                 timeStyle="short"/>
            </h:outputText>
          </h:column>
          <h:column>
            <h:commandButton value="Delete"
                             action="#{messageManager.delete}"/>
          </h:column>
        </h:dataTable>
        <h3><h:outputText value="#{message.title}"/></h3>
        <div><h:outputText value="#{message.text}"/></div>
      </h:form>
    </f:view>
  </body>
</html>
```

1.3.2. How it works

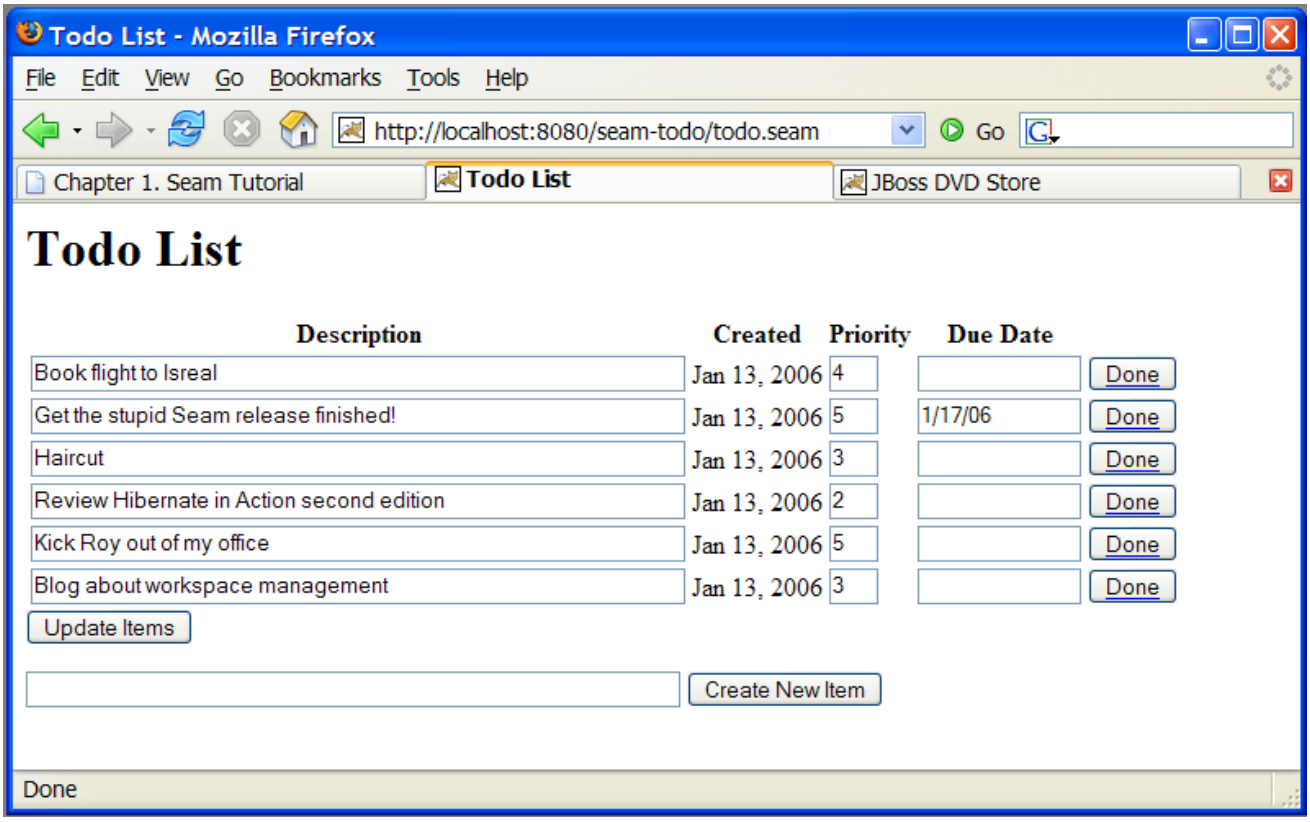
The first time we navigate to **`messages.jsp`**, the page will try to resolve the **`messageList`** context variable. Since this variable is not yet initialized, Seam calls the factory method **`findMessages`**, which queries the database and retrieves a **`DataModel`**. This provides the row data required to render the **`<h:dataTable>`**.

When the user clicks the **`<h:commandLink>`**, JSF calls the **`select()`** action listener. Seam intercepts this call and injects the selected row data into the **`message`** attribute of the **`messageManager`** component. The action listener fires, marking the selected **`Message`** as read. At the end of the call, Seam outjects the selected **`Message`** to the **`message`** context variable. Next, the EJB container commits the transaction, and the change to the **`Message`** is flushed to the database. Finally, the page is re-rendered, redisplaying the message list, and displaying the selected message below it.

When the user clicks the **`<h:commandButton>`**, JSF calls the **`delete()`** action listener. Seam intercepts this call and injects the selected row data into the **`message`** attribute of the **`messageList`** component. The action listener fires, which removes the selected **`Message`** from the list, and also calls **`remove()`** on the **`EntityManager`**. At the end of the call, Seam refreshes the **`messageList`** context variable and clears the **`message`** context variable. The EJB container commits the transaction, and deletes the **`Message`** from the database. Finally, the page is re-rendered, redisplaying the message list.

1.4. SEAM AND JBPM: THE TODO LIST EXAMPLE

jBPM provides sophisticated functionality for workflow and task management. As an example of jBPM's integration with Seam, what follows is a simple "todo list" application. Managing task lists is a core function of jBPM, so little Java is required in this example.



1.4.1. Understanding the code

This example revolves around the jBPM process definition. It also uses two JSPs and two basic JavaBeans. (Session beans are not required here since they would not access the database or have any transactional behavior.) We will start with the process definition:

Example 1.16. todo.jpdl.xml

```
<process-definition name="todo">


  <start-state name="start">                                    (1)
    <transition to="todo"/>
  </start-state>


  <task-node name="todo">                                       (2)

    <task name="todo" description="#{todoList.description}">    (3)

      <assignment actor-id="#{actor.id}"/>                      (4)
    </task>
    <transition to="done"/>
  </task-node>


  <end-state name="done"/>                                      (5)

</process-definition>
```

1	The **<start-state>** node represents the logical start of the process. When the process starts, it immediately transitions to the **todo** node.
2	The **<task-node>** node represents a *wait state*, where business process execution pauses, waiting for one or more tasks to be performed.
3	The **<task>** element defines a task to be performed by a user. Since there is only one task defined on this node, when it is complete, execution resumes, and we transition to the end state. The task gets its description from a Seam component named **todoList** (one of the JavaBeans).
4	Tasks need to be assigned to a user or group of users when they are created. In this case, the task is assigned to the current user, which we get from a built-in Seam component named **actor**. Any Seam component may be used to perform task assignment.
5	The **<end-state>** node defines the logical end of the business process. When execution reaches this node, the process instance is destroyed.

Viewed with the process definition editor provided by JBossIDE, the process definition looks like this:

This document defines our *business process* as a graph of nodes. This is the simplest possible business process: there is one *task* to be performed, and when that task is complete, the business process ends.

The first JavaBean handles the login screen, **login.jsp**. Here, it simply initializes the jBPM actor ID with the **actor** component. In a real application, it would also need to authenticate the user.

Example 1.17. Login.java

```
@Name("login")
public class Login {
```

```
    @In
    private Actor actor;

    private String user;

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String login() {
        actor.setId(user);
        return "/todo.jsp";
    }
}
```

Here we see the use of **`@In`** to inject the built-in **`Actor`** component.

The JSP itself is trivial:

Example 1.18. login.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
    <h1>Login</h1>
    <f:view>
      <h:form>
        <div>
          <h:inputText value="#{login.user}"/>
          <h:commandButton value="Login" action="#{login.login}"/>
        </div>
      </h:form>
    </f:view>
  </body>
</html>
```

The second JavaBean is responsible for starting business process instances, and ending tasks.

Example 1.19. TodoList.java

```
@Name("todoList")
public class TodoList
{
```

```
   private String description;

   public String getDescription()
   {
      return description;
   }

   public void setDescription(String description)
   {
      this.description = description;
   }


   @CreateProcess(definition="todo")
   public void createTodo() {}


   @StartTask @EndTask
   public void done() {}

}
```

1	The description property accepts user input from the JSP page, and exposes it to the process definition, allowing the task description to be set.
2	The Seam **`@CreateProcess`** annotation creates a new jBPM process instance for the named process definition.
3	The Seam **`@StartTask`** annotation starts work on a task. The **`@EndTask`** ends the task, and allows the business process execution to resume.

In a more realistic example, **`@StartTask`** and **`@EndTask`** would not appear on the same method, because some work would need to be done with the application in order to complete the task.

Finally, the core of the application is in **`todo.jsp`**:

Example 1.20. todo.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
  <head>
    <title>Todo List</title>
  </head>
  <body>
    <h1>Todo List</h1>
    <f:view>
      <h:form id="list">
        <div>
```

```
          <h:outputText value="There are no todo items."
                        rendered="#{empty taskInstanceList}"/>
          <h:dataTable value="#{taskInstanceList}" var="task"
                       rendered="#{not empty taskInstanceList}">
            <h:column>
              <f:facet name="header">
                <h:outputText value="Description"/>
              </f:facet>
              <h:inputText value="#{task.description}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Created"/>
              </f:facet>
              <h:outputText value=
                            "#
{task.taskMgmtInstance.processInstance.start}">
                <f:convertDateTime type="date"/>
              </h:outputText>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Priority"/>
              </f:facet>
              <h:inputText value="#{task.priority}" style="width: 30"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Due Date"/>
              </f:facet>
              <h:inputText value="#{task.dueDate}" style="width: 100">
                <f:convertDateTime type="date" dateStyle="short"/>
              </h:inputText>
            </h:column>
            <h:column>
              <s:button value="Done" action="#{todoList.done}"
                        taskInstance="#{task}"/>
            </h:column>
          </h:dataTable>
        </div>
        <div>
          <h:messages/>
        </div>
        <div>
          <h:commandButton value="Update Items" action="update"/>
        </div>
      </h:form>
      <h:form id="new">
        <div>
          <h:inputText value="#{todoList.description}"/>
          <h:commandButton value="Create New Item"
                           action="#{todoList.createTodo}"/>
        </div>
      </h:form>
```

```
    </f:view>
  </body>
</html>
```

For simplicity's sake, we will look at this once section at a time.

The page renders a list of tasks, which it retrieves from a built-in Seam component named **`taskInstanceList`**. The list is defined inside a JSF form.

Example 1.21. todo.jsp (taskInstanceList)

```
<h:form id="list">
  <div>
    <h:outputText value="There are no todo items."
                  rendered="#{empty taskInstanceList}"/>
    <h:dataTable value="#{taskInstanceList}" var="task"
                 rendered="#{not empty taskInstanceList}">
      ...
    </h:dataTable>
  </div>
</h:form>
```

Each element of the list is an instance of the jBPM class **`TaskInstance`**. The following code displays certain properties for every task in the list. Input controls are used for description, priority, and due date to allow users to update these values.

Example 1.22. TaskInstance List Properties

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Description"/>
  </f:facet>
  <h:inputText value="#{task.description}"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Created"/>
  </f:facet>
  <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
    <f:convertDateTime type="date"/>
  </h:outputText>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Priority"/>
  </f:facet>
  <h:inputText value="#{task.priority}" style="width: 30"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Due Date"/>
  </f:facet>
```

```
    <h:inputText value="#{task.dueDate}" style="width: 100">
      <f:convertDateTime type="date" dateStyle="short"/>
    </h:inputText>
  </h:column>
```

NOTE

Seam provides a default JSF date converter for converting a string into a date, so no converter is necessary for the field bound to **#{task.dueDate}**.

This button ends the task by calling the action method annotated **@StartTask @EndTask**. It passes the task ID to Seam as a request parameter:

```
<h:column>
  <s:button value="Done" action="#{todoList.done}"
            taskInstance="#{task}"/>
</h:column>
```

Note that this uses a Seam **<s:button>** JSF control from the **seam-ui.jar** package. This button updates the properties of the tasks. When the form is submitted, Seam and jBPM will make any changes to the tasks persistent. There is no need for any action listener method:

```
 <h:commandButton value="Update Items" action="update"/>
```

A second form on the page creates new items with the action method annotated **@CreateProcess**.

```
 <h:form id="new">
   <div>
      <h:inputText value="#{todoList.description}"/>
      <h:commandButton value="Create New Item"
         action="#{todoList.createTodo}"/>
    </div>
</h:form>
```

1.4.2. How it works

After logging in, **todo.jsp** uses the **taskInstanceList** component to display a table of outstanding todo items for the current user. (Initially there are none.) The page also displays a form to enter a new task item. When the user types the todo item and clicks the **Create New Item** button, **#{todoList.createTodo}** is called. This starts the todo process, as defined in **todo.jpdl.xml**.

When the process instance is created, it transitions immediately to the **todo** state, where a new task is created. The task description is set based on the user input saved to **#{todoList.description}**. The task is then assigned to the current user, stored in the seam actor component. In this example, the process has no extra process state — all the state is stored in the task definition. The process and task information is stored in the database at the end of the request.

When **todo.jsp** is redisplayed, **taskInstanceList** finds the newly-created task and displays it in an **h:dataTable**. The internal state of the task is displayed in each column: **#{task.description}**, **#{task.priority}**, **#{task.dueDate}**, etc. These fields can all be edited and saved to the database.

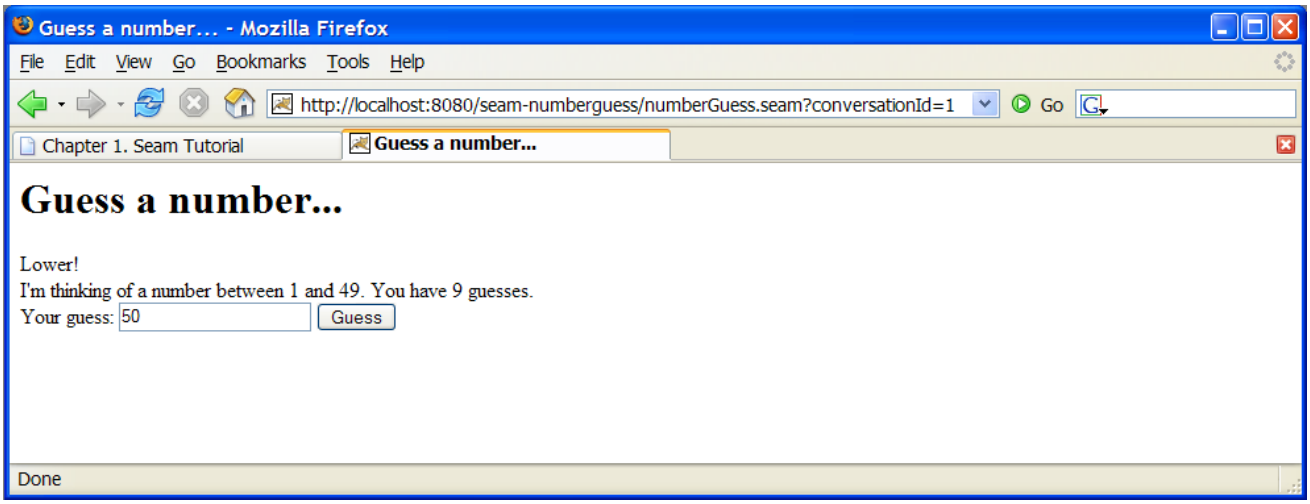
Each todo item also has a **Done** button, which calls **#{todoList.done}**. Each button specifies **taskInstance="#{task}"** (the task for that particular row of the table) so that the **todoList** component is able to distinctly identify which task is complete. The **@StartTask** and **@EndTask** annotations activate and immediately complete the task. The original process then transitions into the **done** state (according to the process definition) and ends. The state of the task and process are both updated in the database.

When **todo.jsp** is displayed again, the completed task is no longer shown in the **taskInstanceList**, since this component displays only incomplete tasks.

1.5. SEAM PAGEFLOW: THE NUMBERGUESS EXAMPLE

For Seam applications with freeform (ad hoc) navigation, JSF/Seam navigation rules are a good way to define page flow. However, in applications with more constrained navigation styles, especially user interfaces that are more stateful, navigation rules make understanding system flow difficult. Combining information from view pages, actions, and navigation rules makes this flow easier to understand.

With Seam, jPDL process definition can be used to define pageflow, as seen in the number guessing example that follows.



1.5.1. Understanding the code

The example uses one JavaBean, three JSP pages and a jPDL pageflow definition. We will start by looking at the pageflow:

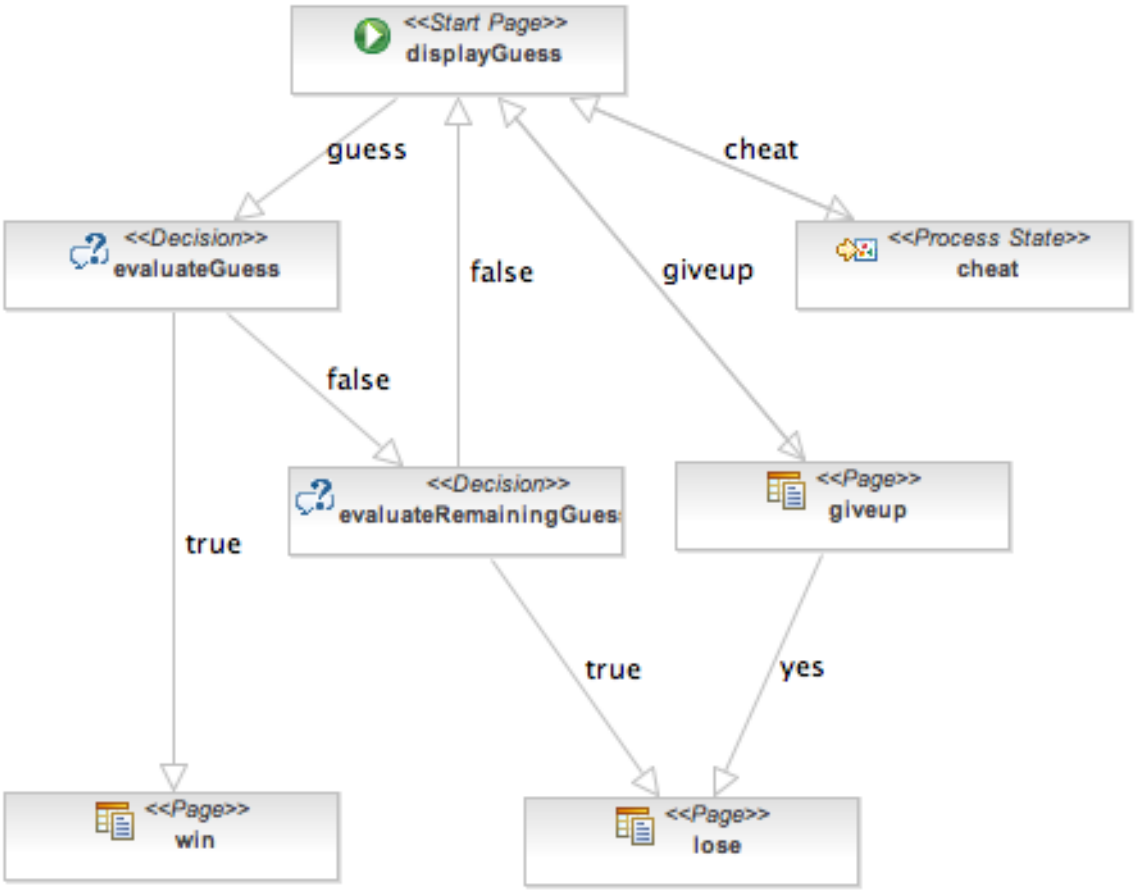
Example 1.23. pageflow.jpdl.xml

```
<pageflow-definition
    xmlns="http://jboss.com/products/seam/pageflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.com/products/seam/pageflow
                        http://jboss.com/products/seam/pageflow-
2.2.xsd"
    name="numberGuess">

  <start-page name="displayGuess" view-id="/numberGuess.jspx">
```

```
    <redirect/>
    <transition name="guess" to="evaluateGuess">
      <action expression="#{numberGuess.guess}"/>
    </transition>
    <transition name="giveup" to="giveup"/>
    <transition name="cheat" to="cheat"/>
  </start-page>

  <decision name="evaluateGuess" expression="#
{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

  <decision name="evaluateRemainingGuesses" expression="#
{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="giveup" view-id="/giveup.jspx">
    <redirect/>
    <transition name="yes" to="lose"/>
    <transition name="no" to="displayGuess"/>
  </page>

  <process-state name="cheat">
    <sub-process name="cheat"/>
    <transition to="displayGuess"/>
  </process-state>

  <page name="win" view-id="/win.jspx">
    <redirect/>
    <end-conversation/>
  </page>

  <page name="lose" view-id="/lose.jspx">
    <redirect/>
    <end-conversation/>
  </page>

</pageflow-definition>
```

1	The **<page>** element defines a wait state where the system displays a particular JSF view and waits for user input. The **view-id** is the same JSF view id used in plain JSF navigation rules. The **redirect** attribute tells Seam to use post-then-redirect when navigating to the page. (This results in friendly browser URLs.)
2	The **<transition>** element names a JSF outcome. The transition is triggered when a JSF action results in that outcome. Execution will then proceed to the next node of the pageflow graph, after invocation of any jBPM transition actions.
3	A transition **<action>** is just like a JSF action, except that it occurs when a jBPM transition occurs. The transition action can invoke any Seam component.
4	A **<decision>** node branches the pageflow, and determines the next node to execute by evaluating a JSF EL expression.

In the JBoss Developer Studio pageflow editor, the pageflow looks like this:



With that pageflow in mind, the rest of the application is much easier to understand.

Here is the main page of the application, **numberGuess.jspx**:

Example 1.24. numberGuess.jspx

```
<?xml version="1.0"?>
```

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:s="http://jboss.com/products/seam/taglib"
          xmlns="http://www.w3.org/1999/xhtml"
          version="2.0">
  <jsp:output doctype-root-element="html"
              doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
              doctype-system=
              "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
  <html>
    <head>
      <title>Guess a number...</title>
      <link href="niceforms.css" rel="stylesheet" type="text/css" />
      <script language="javascript" type="text/javascript"
              src="niceforms.js" />
    </head>
    <body>
      <h1>Guess a number...</h1>
      <f:view>
        <h:form styleClass="niceform">

          <div>
            <h:messages globalOnly="true"/>
            <h:outputText value="Higher!"
                          rendered="#{
                                    numberGuess.randomNumber gt
                                    numberGuess.currentGuess}"/>
            <h:outputText value="Lower!"
                          rendered="#{
                                    numberGuess.randomNumber lt
                                    numberGuess.currentGuess}"/>
          </div>

          <div>
            I'm thinking of a number between
            <h:outputText value="#{numberGuess.smallest}"/> and
            <h:outputText value="#{numberGuess.biggest}"/>. You have
            <h:outputText value="#{numberGuess.remainingGuesses}"/>
            guesses.
          </div>

          <div>
            Your guess:
            <h:inputText value="#{numberGuess.currentGuess}"
                         id="inputGuess" required="true" size="3"
                         rendered="#{
                                    (numberGuess.biggest-
numberGuess.smallest) gt
                                    20}">
              <f:validateLongRange maximum="#{numberGuess.biggest}"
                                   minimum="#{numberGuess.smallest}"/>
            </h:inputText>
            <h:selectOneMenu value="#{numberGuess.currentGuess}"
```

```
                              id="selectGuessMenu" required="true"
                              rendered="#{
                                        (numberGuess.biggest-
numberGuess.smallest) le
                                        20 and
                                        (numberGuess.biggest-
numberGuess.smallest) gt
                                        4}">
              <s:selectItems value="#{numberGuess.possibilities}"
                             var="i" label="#{i}"/>
            </h:selectOneMenu>
            <h:selectOneRadio value="#{numberGuess.currentGuess}"
                              id="selectGuessRadio"
                              required="true"
                              rendered="#{
                                        (numberGuess.biggest-
numberGuess.smallest) le
                                        4}">
              <s:selectItems value="#{numberGuess.possibilities}"
                             var="i" label="#{i}"/>
            </h:selectOneRadio>
            <h:commandButton value="Guess" action="guess"/>
            <s:button value="Cheat" view="/confirm.jspx"/>
            <s:button value="Give up" action="giveup"/>
          </div>

          <div>
            <h:message for="inputGuess" style="color: red"/>
          </div>

        </h:form>
      </f:view>
    </body>
  </html>
</jsp:root>
```

Note that the command button names the **guess** transition instead of calling an action directly.

The **win.jspx** page is predictable:

Example 1.25. win.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns="http://www.w3.org/1999/xhtml"
          version="2.0">
  <jsp:output doctype-root-element="html"
              doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
              doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
  <html>
    <head>
```

```
      <title>You won!</title>
      <link href="niceforms.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
      <h1>You won!</h1>
      <f:view>
        Yes, the answer was
        <h:outputText value="#{numberGuess.currentGuess}" />.
        It took you
        <h:outputText value="#{numberGuess.guessCount}" /> guesses.
        <h:outputText value="But you cheated, so it does not count!"
                      rendered="#{numberGuess.cheat}"/>
        Would you like to <a href="numberGuess.seam">play again</a>?
      </f:view>
    </body>
  </html>
</jsp:root>
```

The **`lose.jspx`** is very similar, so we have not included it here.

Finally, the application code is as follows:

Example 1.26. NumberGuess.java

```
@Name("numberGuess")
@Scope(ScopeType.CONVERSATION)
public class NumberGuess implements Serializable {

   private int randomNumber;
   private Integer currentGuess;
   private int biggest;
   private int smallest;
   private int guessCount;
   private int maxGuesses;
   private boolean cheated;



   @Create
   public void begin()
   {
      randomNumber = new Random().nextInt(100);
      guessCount = 0;
      biggest = 100;
      smallest = 1;
   }

   public void setCurrentGuess(Integer guess)
   {
      this.currentGuess = guess;
   }

   public Integer getCurrentGuess()
   {
      return currentGuess;
```

```
    }

    public void guess()
    {
       if (currentGuess>randomNumber)
       {
          biggest = currentGuess - 1;
       }
       if (currentGuess<randomNumber)
       {
          smallest = currentGuess + 1;
       }
       guessCount ++;
    }

    public boolean isCorrectGuess()
    {
       return currentGuess==randomNumber;
    }

    public int getBiggest()
    {
       return biggest;
    }

    public int getSmallest()
    {
       return smallest;
    }

    public int getGuessCount()
    {
       return guessCount;
    }

    public boolean isLastGuess()
    {
       return guessCount==maxGuesses;
    }

    public int getRemainingGuesses() {
       return maxGuesses-guessCount;
    }

    public void setMaxGuesses(int maxGuesses) {
       this.maxGuesses = maxGuesses;
    }

    public int getMaxGuesses() {
       return maxGuesses;
    }

    public int getRandomNumber() {
       return randomNumber;
    }
```

```
    public void cheated()
    {
       cheated = true;
    }

    public boolean isCheat() {
       return cheated;
    }

    public List<Integer> getPossibilities()
    {
       List<Integer> result = new ArrayList<Integer>();
       for(int i=smallest; i<=biggest; i++) result.add(i);
       return result;
    }

}
```

1. The first time a JSP page asks for a **numberGuess** component, Seam will create a new one for it, and the **@Create** method will be invoked, allowing the component to initialize itself.

The **pages.xml** file starts a Seam *conversation*, and specifies the pageflow definition to use for the conversation's page flow. Refer to Chapter 8, *Conversations and workspace management* for more information.

Example 1.27. pages.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.com/products/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.2.xsd">
  <page view-id="/numberGuess.jspx">
    <begin-conversation join="true" pageflow="numberGuess"/>
  </page>
</pages>
```

This component is pure business logic. Since it requires no information about the user interaction flow, it is potentially more reuseable.

1.5.2. How it works

The game begins in the **numberGuess.jspx** view. When the page is first displayed, the **pages.xml** configuration activates a conversation and associates it with the **numberGuess** pageflow. The pageflow starts with a **start-page** tag (a *wait* state), so the **numberGuess.xhtml** is rendered.

The view references the **numberGuess** component, which causes a new instance to be created and stored in the conversation. the **@Create** method is called, initializing the game's state. The view displays an **h:form**, which allows the user to edit **#{numberGuess.currentGuess}**.

The "Guess" button triggers the **guess** action. Seam refers to the pageflow to handle the action, and the pageflow invokes **#{numberGuess.guess}** (which updates the guess count and highest/lowest suggestions in the **numberGuess** component), and transitions to the **evaluateGuess** state.

The **evaluateGuess** state checks the value of **#{numberGuess.correctGuess}** and transitions to either the **win** or **evaluatingRemainingGuesses** state. Assuming the number was incorrect, the pageflow transitions to **evaluatingRemainingGuesses**. This is also a decision state, which tests the **#{numberGuess.lastGuess}** state to determine whether or not the user is allowed further guesses. If further guesses are allowed (**lastGuess** is **false**), we transition back to the original **displayGuess** state. Since this is a page state, the associated page **/numberGuess.jspx** is displayed. This page also contains a redirect element, so Seam sends a redirect to the user's browser, which begins the process again.

If on a future request either the **win** or the **lose** transition were invoked, the user would be taken to **/win.jspx** or **/lose.jspx** respectively. Both states specify that Seam should end the conversation, stop holding game and pageflow state, and redirect the user to the final page.

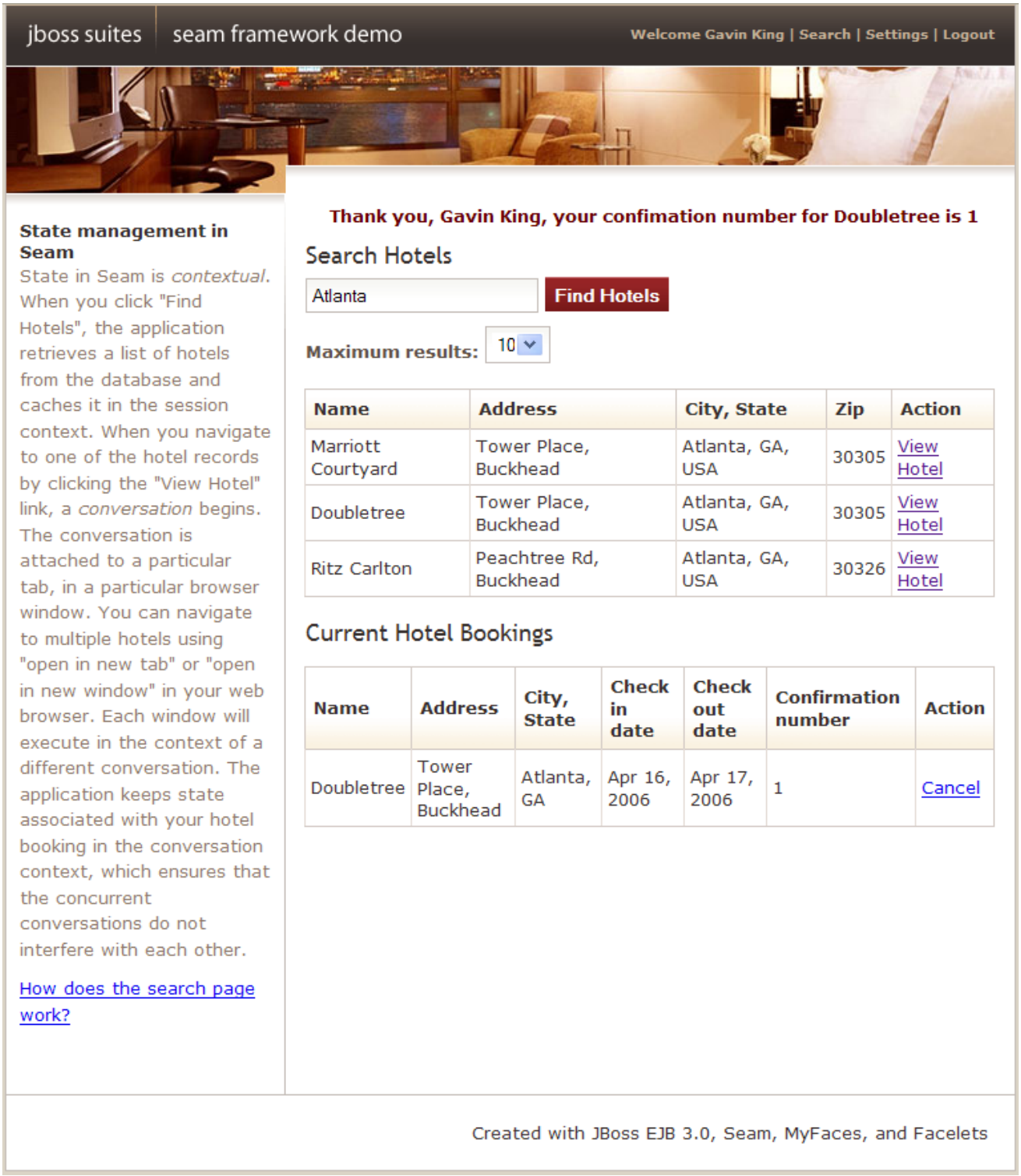
The numberguess example also contains **Give up** and **Cheat** buttons. The pageflow state for both actions is relatively easy to trace, so we do not describe it here. Pay particular attention to the **cheat** transition, which loads a sub-process to handle that particular flow. Although in this application this process is superfluous, this demonstrates how complex pageflows can be broken down into smaller, simpler structures to make them easier to understand.

1.6. A COMPLETE SEAM APPLICATION: THE HOTEL BOOKING EXAMPLE

1.6.1. Introduction

The booking application is a complete hotel room reservation system incorporating the following features:

- User registration
- Login
- Logout
- Set password
- Hotel search
- Hotel selection
- Room reservation
- Reservation confirmation
- Existing reservation list



The booking application uses JSF, EJB 3.0 and Seam, together with Facelets for the view. There is also a port of this application to JSF, Facelets, Seam, JavaBeans and Hibernate3.

One of the things you will notice about this application is that it is extremely *robust*. You can open multiple windows, use the back and browser refresh buttons, and enter nonsensical data, but the application is difficult to crash. Seam was designed to make building robust web applications straightforward, so robustness that would previously be hand-coded comes naturally and automatically with Seam.

As you browse the source code of the example application and learn how the application works, pay particular attention to the way the declarative state management and integrated validation has been used to achieve this robustness.

1.6.2. Overview of the booking example

The project structure here is identical to that of the previous project. To install and deploy this application, refer to Section 1.1, “Using the Seam examples”. Once you have successfully started the application, you can access it by pointing your browser to **`http://localhost:8080/seam-booking/`**

The application uses six session beans to implement the business logic for the following features:

- **`AuthenticatorAction`** provides the log in authentication logic.
- **`BookingListAction`** retrieves existing bookings for the currently logged in user.
- **`ChangePasswordAction`** updates the password of the currently logged in user.
- **`HotelBookingAction`** implements booking and confirmation functionality. This is implemented as a *conversation*, so this is one of the more important classes in the application.
- **`HotelSearchingAction`** implements the hotel search functionality.
- **`RegisterAction`** registers a new system user.

Three entity beans implement the application's persistent domain model:

- **`Hotel`** is an entity bean that represents a hotel
- **`Booking`** is an entity bean that represents an existing booking
- **`User`** is an entity bean representing a user who can make hotel bookings

1.6.3. Understanding Seam conversations

This tutorial concentrates upon one particular piece of functionality: placing a hotel reservation. From the user's perspective, hotel search, selection, booking, and confirmation are one continuous unit of work — a *conversation*. However, from our perspective, it is important that searching remains separate so that users can select multiple hotels from the same search results page, and open distinct conversations in separate browser tabs.

Most web application architectures do not have first class constructs to represent conversations, which makes managing conversational state problematic. Java web applications generally use a combination of several techniques. Some state is transferred in the URL, but what cannot be transferred here is either added to the **`HttpSession`** or recorded to the database at the beginning and end of each request.

Since the database is the least-scalable tier, this drastically reduces scalability. The extra traffic to and from the database also increases latency. In order to reduce redundant traffic, Java applications often introduce a data cache to store commonly-accessed data between requests. However, since invalidation is based upon an LRU policy, rather than whether the user has finished using the data, this cache is inefficient. It is also shared between concurrent transactions, which introduces further issues associated with keeping the cached state consistent with that of the database.

State held in the **`HttpSession`** suffers similar issues. The **`HttpSession`** is fine for storing true session data — data common to all requests between user and application — but for data related to individual request series, it does not work so well. Conversations stored here quickly break down when dealing with multiple windows or the back button. Without careful programming, data in the **`HttpSession`** can also grow quite large, which makes the session difficult to cluster. Developing mechanisms to deal with

the problems these methods present (by isolating session state associated with distinct concurrent conversations, and incorporating failsafes to ensure conversation state is destroyed when a conversation is aborted) can be complicated.

Seam greatly improves conditions by introducing *conversation context* as a first class construct. Conversation state is stored safely in this context, with a well-defined life cycle. Even better, there is no need to push data continually between the application server and the database; the conversation context is a natural cache for currently-used data.

In the following application, the conversation context is used to store stateful session beans. These are sometimes regarded as detrimental to scalability, and in the past, they may have been. However, modern application servers have sophisticated mechanisms for stateful session bean replication. JBoss Enterprise Application Platform performs fine-grained replication, replicating only altered bean attribute values. Used correctly, stateful session beans pose no scalability problems, but for those uncomfortable or unfamiliar with the use of stateful session beans, Seam also allows the use of POJOs.

The booking example shows one way that stateful components with different scopes can collaborate to achieve complex behaviors. The main page of the booking application allows the user to search for hotels. Search results are stored in the Seam session scope. When the user navigate to a hotel, a conversation begins, and a conversation scoped component retrieves the selected hotel from the session scoped component.

The booking example also demonstrates the use of RichFaces Ajax to implement rich client behavior without handwritten JavaScript.

The search function is implemented with a session-scoped stateful session bean, similar to the one used in the message list example.

Example 1.28. HotelSearchingAction.java

```
@Stateful
```

```
@Name("hotelSearch")
@Scope(ScopeType.SESSION)
@Restrict("#{identity.loggedIn}")
```

```
public class HotelSearchingAction implements HotelSearching
{

    @PersistenceContext
    private EntityManager em;

    private String searchString;
    private int pageSize = 10;
    private int page;

    @DataModel
```

```
    private List<Hotel> hotels;

    public void find()
    {
```

```
        page = 0;
        queryHotels();
    }
    public void nextPage()
    {
        page++;
        queryHotels();
    }

    private void queryHotels()
    {
        hotels =
            em.createQuery("select h from Hotel h where lower(h.name)
like #{pattern} " +
                           "or lower(h.city) like #{pattern} " +
                           "or lower(h.zip) like #{pattern} " +
                           "or lower(h.address) like #{pattern}")
              .setMaxResults(pageSize)
              .setFirstResult( page * pageSize )
              .getResultList();
    }

    public boolean isNextPageAvailable()
    {
        return hotels!=null && hotels.size()==pageSize;
    }

    public int getPageSize() {
        return pageSize;
    }

    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }

    @Factory(value="pattern", scope=ScopeType.EVENT)
    public String getSearchPattern()
    {
        return searchString==null ?
            "%" : '%' + searchString.toLowerCase().replace('*', '%') +
'%';
    }

    public String getSearchString()
    {
        return searchString;
    }

    public void setSearchString(String searchString)
    {
        this.searchString = searchString;
    }

    @Remove
```

```
    public void destroy() {}
}
```

1	The EJB standard **@Stateful** annotation identifies this class as a stateful session bean. Stateful session beans are scoped to the conversation context by default.
2	The **@Restrict** annotation applies a security restriction to the component. It restricts access to the component allowing only logged-in users. The security chapter explains more about security in Seam.
3	The **@DataModel** annotation exposes a **List** as a JSF **ListDataModel**. This makes it easy to implement clickable lists for search screens. In this case, the list of hotels is exposed to the page as a **ListDataModel** in the conversation variable named **hotels**.
4	The EJB standard **@Remove** annotation specifies that a stateful session bean should be removed and its state destroyed after invocation of the annotated method. In Seam, all stateful session beans must define a parameterless method marked **@Remove**. This method will be called when Seam destroys the session context.

The main page of the application is a Facelets page. The fragment that relates to searching for hotels is shown below:

Example 1.29. main.xhtml

```
<div class="section">

  <span class="errors">
    <h:messages globalOnly="true"/>
  </span>

  <h1>Search Hotels</h1>

  <h:form id="searchCriteria">
    <fieldset>
      <h:inputText id="searchString" value="#
{hotelSearch.searchString}"
                   style="width: 165px;">
        <a:support event="onkeyup" actionListener="#{hotelSearch.find}"
```

1

```
                   reRender="searchResults" />
      </h:inputText>
       
      <a:commandButton id="findHotels" value="Find Hotels" action="#
{hotelSearch.find}"
                       reRender="searchResults"/>
       
      <a:status>
```

2

```
        <f:facet name="start">
          <h:graphicImage value="/img/spinner.gif"/>
        </f:facet>
      </a:status>
      <br/>
      <h:outputLabel for="pageSize">Maximum
results:</h:outputLabel> 
      <h:selectOneMenu value="#{hotelSearch.pageSize}" id="pageSize">
        <f:selectItem itemLabel="5" itemValue="5"/>
        <f:selectItem itemLabel="10" itemValue="10"/>
        <f:selectItem itemLabel="20" itemValue="20"/>
      </h:selectOneMenu>
    </fieldset>
  </h:form>

</div>

<a:outputPanel id="searchResults">
```

3

```
  <div class="section">
    <h:outputText value="No Hotels Found"
                  rendered="#{hotels != null and hotels.rowCount==0}"/>
    <h:dataTable id="hotels" value="#{hotels}" var="hot"
                 rendered="#{hotels.rowCount>0}">
      <h:column>
        <f:facet name="header">Name</f:facet>
        #{hot.name}
      </h:column>
      <h:column>
        <f:facet name="header">Address</f:facet>
        #{hot.address}
      </h:column>
      <h:column>
        <f:facet name="header">City, State</f:facet>
        #{hot.city}, #{hot.state}, #{hot.country}
      </h:column>
      <h:column>
        <f:facet name="header">Zip</f:facet>
        #{hot.zip}
      </h:column>
      <h:column>
        <f:facet name="header">Action</f:facet>
        <s:link id="viewHotel" value="View Hotel"
```

4

```
                action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
    </h:dataTable>
    <s:link value="More results" action="#{hotelSearch.nextPage}"
            rendered="#{hotelSearch.nextPageAvailable}"/>
  </div>
</a:outputPanel>
```

1	The RichFaces Ajax **`<a:support>`** tag allows a JSF action event listener to be called by asynchronous **`XMLHttpRequest`** when a JavaScript event like **`onkeyup`** occurs. Even better, the **`reRender`** attribute lets us render a fragment of the JSF page and perform a partial page update when the asynchronous response is received.
2	The RichFaces Ajax **`<a:status>`** tag lets us display an animated image while we wait for asynchronous requests to return.
3	The RichFaces Ajax **`<a:outputPanel>`** tag defines a region of the page which can be re-rendered by an asynchronous request.
4	The Seam **`<s:link>`** tag lets us attach a JSF action listener to an ordinary (non-JavaScript) HTML link. The advantage of this over the standard JSF **`<h:commandLink>`** is that it preserves the operation of "open in new window" and "open in new tab". Also notice that we use a method binding with a parameter: **`#{hotelBooking.selectHotel(hot)}`**. This is not possible in the standard Unified EL, but Seam provides an extension to the EL that lets you use parameters on any method binding expression. If you are wondering how navigation occurs, you can find all the rules in **`WEB-INF/pages.xml`**; this is discussed in Section 7.7, “Navigation”.

This page displays search results dynamically as the user types, and passes a selected hotel to the **`selectHotel()`** method of **`HotelBookingAction`**, where the real work occurs.

The following code shows how the booking example application uses a conversation-scoped stateful session bean to achieve a natural cache of persistent data related to the conversation. Think of the code as a list of scripted actions that implement the various steps of the conversation.

Example 1.30. HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

    @PersistenceContext(type=EXTENDED)
(1)

    private EntityManager em;

    @In
        private User user;

    @In(required=false) @Out
    private Hotel hotel;

    @In(required=false)
    @Out(required=false)
(2)

    private Booking booking;
```

```
    @In
    private FacesMessages facesMessages;

    @In
    private Events events;

    @Logger
        private Log log;

    private boolean bookingValid;

    @Begin
```

3

```
    public void selectHotel(Hotel selectedHotel)
    {
        hotel = em.merge(selectedHotel);
    }

    public void bookHotel()
    {
        booking = new Booking(hotel, user);
        Calendar calendar = Calendar.getInstance();
        booking.setCheckinDate( calendar.getTime() );
        calendar.add(Calendar.DAY_OF_MONTH, 1);
        booking.setCheckoutDate( calendar.getTime() );
    }

    public void setBookingDetails()
    {
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.DAY_OF_MONTH, -1);
        if ( booking.getCheckinDate().before( calendar.getTime() ) )
            {
                facesMessages.addToControl("checkinDate",
                                           "Check in date must be a
future date");
                bookingValid=false;
            }
        else if ( !booking.getCheckinDate().before(
booking.getCheckoutDate() ) )
            {
                facesMessages.addToControl("checkoutDate",
                                           "Check out date must be
later " +
                                           "than check in date");
                bookingValid=false;
            }
        else
            {
                bookingValid=true;
            }
    }

    public boolean isBookingValid()
```

```
    {
        return bookingValid;
    }

    @End
```

```
    public void confirm()
    {
        em.persist(booking);
        facesMessages.add("Thank you, #{user.name}, your confimation
number " +
                          " for #{hotel.name} is #{booki g.id}");
        log.info("New booking: #{booking.id} for #{user.username}");
        events.raiseTransactionSuccessEvent("bookingConfirmed");
    }

    @End
    public void cancel() {}

    @Remove
```

```
    public void destroy() {}
}
```

1	This bean uses an EJB3 *extended persistence context*, so that any entity instances remain managed for the whole life cycle of the stateful session bean.
2	The **@Out** annotation declares that an attribute value is *outjected* to a context variable after method invocations. In this case, the context variable named **hotel** will be set to the value of the **hotel** instance variable after every action listener invocation completes.
3	The **@Begin** annotation specifies that the annotated method begins a *long-running conversation*, so the current conversation context will not be destroyed at the end of the request. Instead, it will be reassociated with every request from the current window, and destroyed either by timeout due to conversation inactivity or invocation of a matching **@End** method.
4	The **@End** annotation specifies that the annotated method ends the current long-running conversation, so the current conversation context will be destroyed at the end of the request.
5	This EJB remove method will be called when Seam destroys the conversation context. Do not forget to define this method!

HotelBookingAction contains all the action listener methods that implement selection, booking and booking confirmation, and holds state related to this work in its instance variables. This code is much cleaner and simpler than getting and setting **HttpSession** attributes.

Even better, a user can have multiple isolated conversations per log in session. Log in, run a search, and navigate to different hotel pages in multiple browser tabs. You'll be able to work on creating two different hotel reservations at the same time. If you leave any one conversation inactive for long enough, Seam

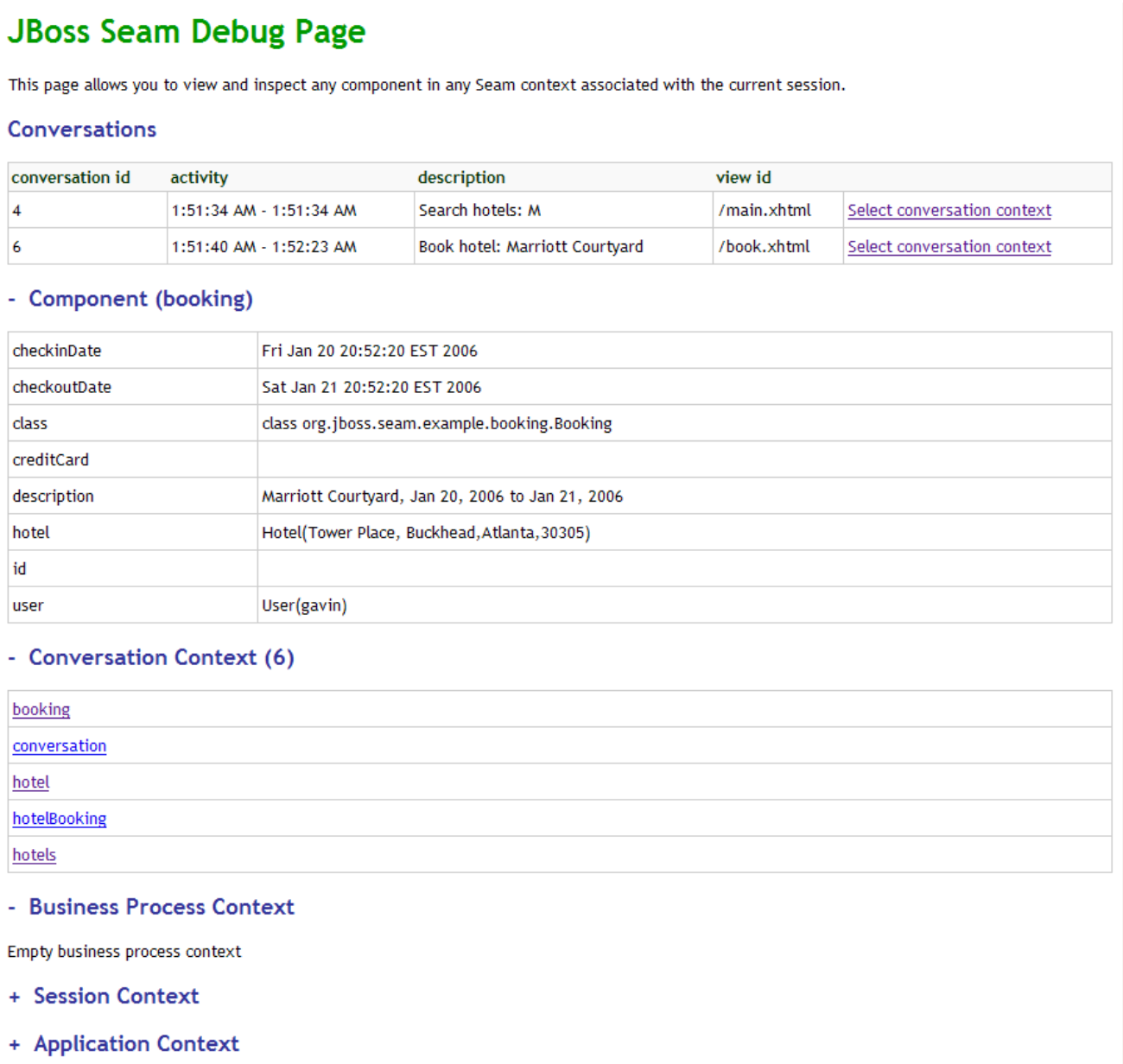
will eventually time out that conversation and destroy its state. If, after ending a conversation, you backtrack to a page of that conversation and try to perform an action, Seam will detect that the conversation was already ended, and redirect you to the search page.

1.6.4. The Seam Debug Page

The WAR also includes **`seam-debug.jar`**. To make the Seam debug page available, deploy this jar in **`WEB-INF/lib`** alongside Facelets, and set the debug property of the **`init`** component as shown here:

```
<core:init jndi-pattern="@jndiPattern@" debug="true"/>
```

The debug page lets you browse and inspect the Seam components in any of the Seam contexts associated with your current log in session. Just point your browser at **`http://localhost:8080/seam-booking/debug.seam`** .



JBoss Seam Debug Page

This page allows you to view and inspect any component in any Seam context associated with the current session.

Conversations

conversation id	activity	description	view id	
4	1:51:34 AM - 1:51:34 AM	Search hotels: M	/main.xhtml	Select conversation context
6	1:51:40 AM - 1:52:23 AM	Book hotel: Marriott Courtyard	/book.xhtml	Select conversation context

- Component (booking)

checkinDate	Fri Jan 20 20:52:20 EST 2006
checkoutDate	Sat Jan 21 20:52:20 EST 2006
class	class org.jboss.seam.example.booking.Booking
creditCard	
description	Marriott Courtyard, Jan 20, 2006 to Jan 21, 2006
hotel	Hotel(Tower Place, Buckhead,Atlanta,30305)
id	
user	User(gavin)

- Conversation Context (6)

booking
conversation
hotel
hotelBooking
hotels

- Business Process Context

Empty business process context

+ Session Context

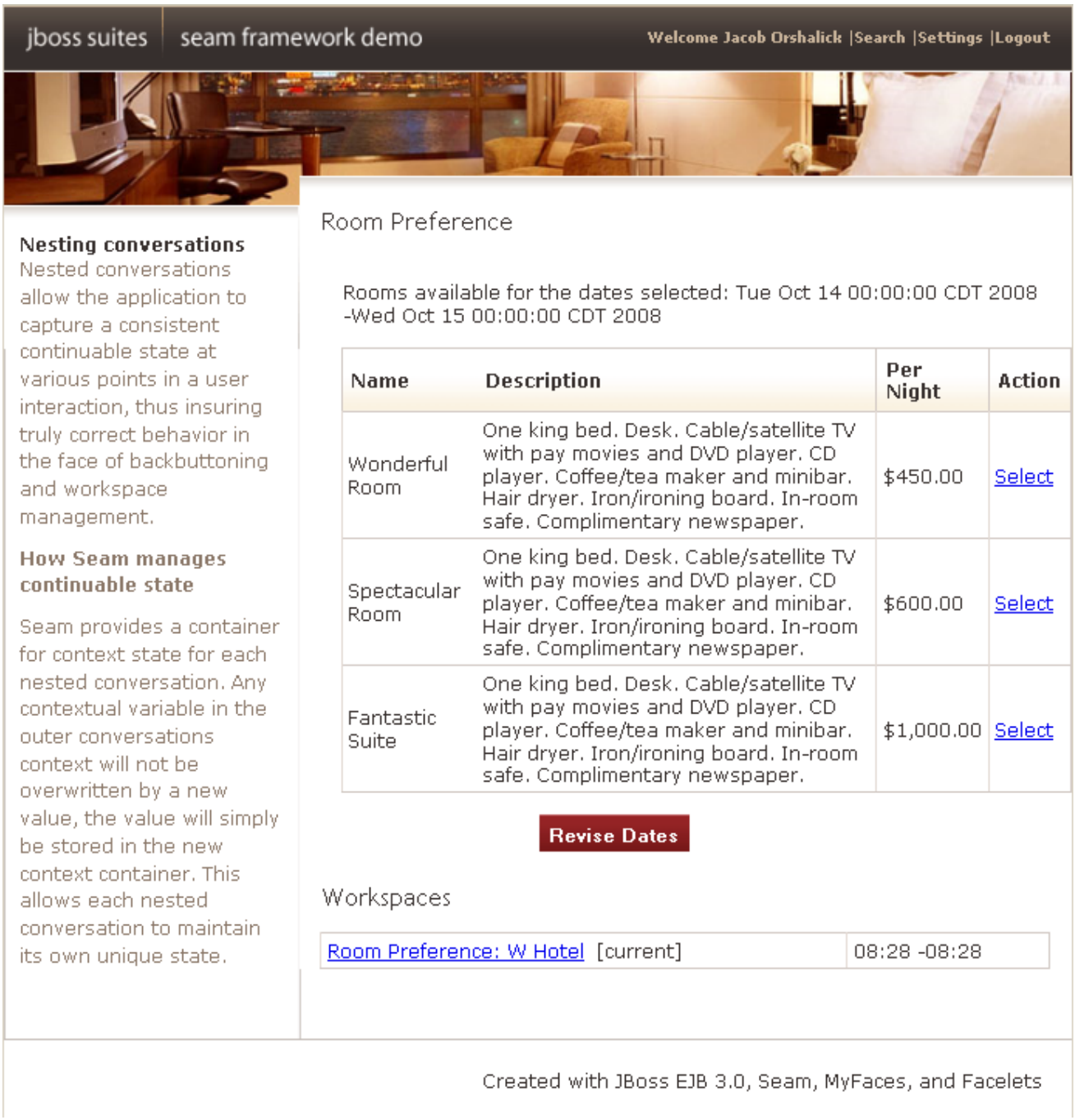
+ Application Context

1.7. NESTED CONVERSATIONS: EXTENDING THE HOTEL BOOKING EXAMPLE

1.7.1. Introduction

Long-running conversations let you easily maintain state consistency in an application, even in the face of multi-window operation and back-buttoning. Unfortunately, simply beginning and ending a long-running conversation is not always enough. Depending on the requirements of the application, inconsistencies between user expectations and application state can still result.

The nested booking application extends the features of the hotel booking application to incorporate room selection. Each hotel has a list of available rooms from which the user can select. This requires the addition of a room selection page in the hotel reservation flow.



The user can now select any available room to be included in the booking. If room selection were left in the same conversation context, this could lead to issues with state consistency — if a conversation variable changes, it affects all windows operating within the same conversation context.

For example, suppose the user clones the room selection screen in a new window. The user then selects the *Wonderful Room* and proceeds to the confirmation screen. To check the cost of a more expensive room, the user returns to the original window, selects the *Fantastic Suite* for booking, and again proceeds

to confirmation. After reviewing the total cost, the user returns to the window showing *Wonderful Room* to confirm.

In this scenario, if all state were stored in the conversation, flexibility for multi-window operation within the same conversation would be limited. Nested conversations allow us to achieve correct behavior even when context can vary within the same conversation.

1.7.2. Understanding Nested Conversations

The following code shows the behavior of the hotel booking application with intended behavior for nested conversations. Again, think of the code as a set of steps to be read in sequence.

Example 1.31. RoomPreferenceAction.java

```
@Stateful
@Name("roomPreference")
@Restrict("#{identity.loggedIn}")
public class RoomPreferenceAction implements RoomPreference
{

    @Logger
        private Log log;

    @In private Hotel hotel;
```

1

```


    @In private Booking booking;

    @DataModel(value="availableRooms")
    private List<Room> availableRooms;

    @DataModelSelection(value="availableRooms")
    private Room roomSelection;

    @In(required=false, value="roomSelection")
    @Out(required=false, value="roomSelection")
    private Room room;

    @Factory("availableRooms")
    public void loadAvailableRooms()
    {
        availableRooms =
hotel.getAvailableRooms(booking.getCheckinDate(),

booking.getCheckoutDate());
        log.info("Retrieved #0 available rooms",
availableRooms.size());
    }

    public BigDecimal getExpectedPrice()
    {
        log.info("Retrieving price for room #0",
roomSelection.getName());
```

```
        return booking.getTotal(roomSelection);
    }

    @Begin(nested=true)
```

```
    public String selectPreference()
    {
        log.info("Room selected");

        this.room = this.roomSelection;
```

```


        return "payment";
    }

    public String requestConfirmation()
    {
        // all validations are performed through the s:validateAll, so
checks are
        // already performed
        log.info("Request confirmation from user");

        return "confirm";
    }

    @End(beforeRedirect=true)
```

```
    public String cancel()
    {
        log.info("ending conversation");

        return "cancel";
    }

    @Destroy @Remove
        public void destroy() {}
}
```

1	The **`hotel`** instance is injected from the conversation context. The hotel is loaded through an *extended persistence context* so that the entity remains managed throughout the conversation. This allows us to lazily load the **`availableRooms`** through an **`@Factory`** method by simply walking the association.
2	When **`@Begin(nested=true)`** is encountered, a nested conversation is pushed onto the conversation stack. When executing within a nested conversation, components still have access to all outer conversation state, but setting any values in the nested conversation's state container does not affect the outer conversation. In addition, nested conversations can exist concurrently stacked on the same outer conversation, allowing independent state for each.

3	The **roomSelection** is outjected to the conversation based on the **@DataModelSelection**. Note that because the nested conversation has an independent context, the **roomSelection** is only set into the new nested conversation. Should the user select a different preference in another window or tab a new nested conversation would be started.
4	The **@End** annotation pops the conversation stack and resumes the outer conversation. The **roomSelection** is destroyed along with the conversation context.

When we begin a nested conversation, it is pushed onto the conversation stack. In the **nestedbooking** example, the conversation stack consists of the external long-running conversation (the booking) and each of the nested conversations (room selections).

Example 1.32. rooms.xhtml

```
<div class="section">
  <h1>Room Preference</h1>
</div>

<div class="section">
  <h:form id="room_selections_form">
    <div class="section">
      <h:outputText styleClass="output"
         value="No rooms available for the dates selected: "
         rendered="#{availableRooms != null and availableRooms.rowCount
== 0}"/>
      <h:outputText styleClass="output"
         value="Rooms available for the dates selected: "
         rendered="#{availableRooms != null and availableRooms.rowCount
> 0}"/>

      <h:outputText styleClass="output" value="#
{booking.checkinDate}"/>
      <h:outputText styleClass="output" value="#
{booking.checkoutDate}"/>

      <br/><br/>

      <h:dataTable value="#{availableRooms}" var="room"
(1)
         rendered="#{availableRooms.rowCount > 0}">
        <h:column>
          <f:facet name="header">Name</f:facet>
          #{room.name}
        </h:column>
        <h:column>
          <f:facet name="header">Description</f:facet>
          #{room.description}
        </h:column>
        <h:column>
          <f:facet name="header">Per Night</f:facet>
          <h:outputText value="#{room.price}">
```

```
            <f:convertNumber type="currency" currencySymbol="$"/>
          </h:outputText>
        </h:column>
        <h:column>
          <f:facet name="header">Action</f:facet>
          <h:commandLink id="selectRoomPreference"
            action="#

{roomPreference.selectPreference}">Select</h:commandLink>
        </h:column>
      </h:dataTable>
    </div>
    <div class="entry">
      <div class="label"> </div>
      <div class="input">
        <s:button id="cancel" value="Revise Dates" view="/book.xhtml"/>


      </div>
    </div>
  </h:form>
</div>
```

1	When requested from EL, the **#{availableRooms}** are loaded by the **@Factory** method defined in **RoomPreferenceAction**. The **@Factory** method will only be executed once to load the values into the current context as a **@DataModel** instance.
2	Invoking the **#{roomPreference.selectPreference}** action results in the row being selected and set into the **@DataModelSelection**. This value is then outjected to the nested conversation context.
3	Revising the dates simply returns to the **/book.xhtml**. Note that we have not yet nested a conversation (no room preference has been selected), so the current conversation can simply be resumed. The **<s:button>** component simply propagates the current conversation when displaying the **/book.xhtml** view.

Now that you have seen how to nest a conversation, the following code shows how we can confirm the booking of a selected room by extending the behavior of the **HotelBookingAction**.

Example 1.33. HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

   @PersistenceContext(type=EXTENDED)
   private EntityManager em;

   @In
   private User user;
```

```
   @In(required=false) @Out
   private Hotel hotel;

   @In(required=false)
   @Out(required=false)
   private Booking booking;

   @In(required=false)
   private Room roomSelection;

   @In
   private FacesMessages facesMessages;

   @In
   private Events events;

   @Logger
   private Log log;

   @Begin
   public void selectHotel(Hotel selectedHotel)
   {
      log.info("Selected hotel #0", selectedHotel.getName());
      hotel = em.merge(selectedHotel);
   }

   public String setBookingDates()
   {
      // the result will indicate whether or not to begin the nested
conversation
      // as well as the navigation.  if a null result is returned, the
nested
      // conversation will not begin, and the user will be returned to
the current
      // page to fix validation issues
      String result = null;

      Calendar calendar = Calendar.getInstance();
      calendar.add(Calendar.DAY_OF_MONTH, -1);

      // validate what we have received from the user so far
      if ( booking.getCheckinDate().before( calendar.getTime() ) )
      {
         facesMessages.addToControl("checkinDate",
                                    "Check in date must be a future
date");
      }
      else if ( !booking.getCheckinDate().before(
booking.getCheckoutDate() ) )
      {
         facesMessages.addToControl("checkoutDate",
                                    "Check out date must be later than
check in date");
      }
      else
```

```
      {
         result = "rooms";
      }

      return result;
   }

   public void bookHotel()
   {
      booking = new Booking(hotel, user);
      Calendar calendar = Calendar.getInstance();
      booking.setCheckinDate( calendar.getTime() );
      calendar.add(Calendar.DAY_OF_MONTH, 1);
      booking.setCheckoutDate( calendar.getTime() );
   }

   @End(root=true)
```

1

```
   public void confirm()
   {
      // on confirmation we set the room preference in the booking.  the
room preference
      // will be injected based on the nested conversation we are in.
      booking.setRoomPreference(roomSelection);
```

2

```
      em.persist(booking);
      facesMessages.add("Thank you, #{user.name}, your confimation
number" +
                        " for #{hotel.name} is #{booking.id}");
      log.info("New booking: #{booking.id} for #{user.username}");
      events.raiseTransactionSuccessEvent("bookingConfirmed");
   }

   @End(root=true, beforeRedirect=true)
```

3

```
   public void cancel() {}

   @Destroy @Remove
   public void destroy() {}
}
```

1	Annotating an action with **`@End(root=true)`** ends the root conversation which effectively destroys the entire conversation stack. When any conversation is ended, its nested conversations are ended as well. As the root is the conversation that started it all, this is a simple way to destroy and release all state associated with a workspace once the booking is confirmed.
2	The **`roomSelection`** is only associated with the **`booking`** on user confirmation. While outjecting values to the nested conversation context will not impact the outer conversation, any objects injected from the outer conversation are injected by reference. This means that any changing to these objects will be reflected in the parent conversation as well as other concurrent nested conversations.

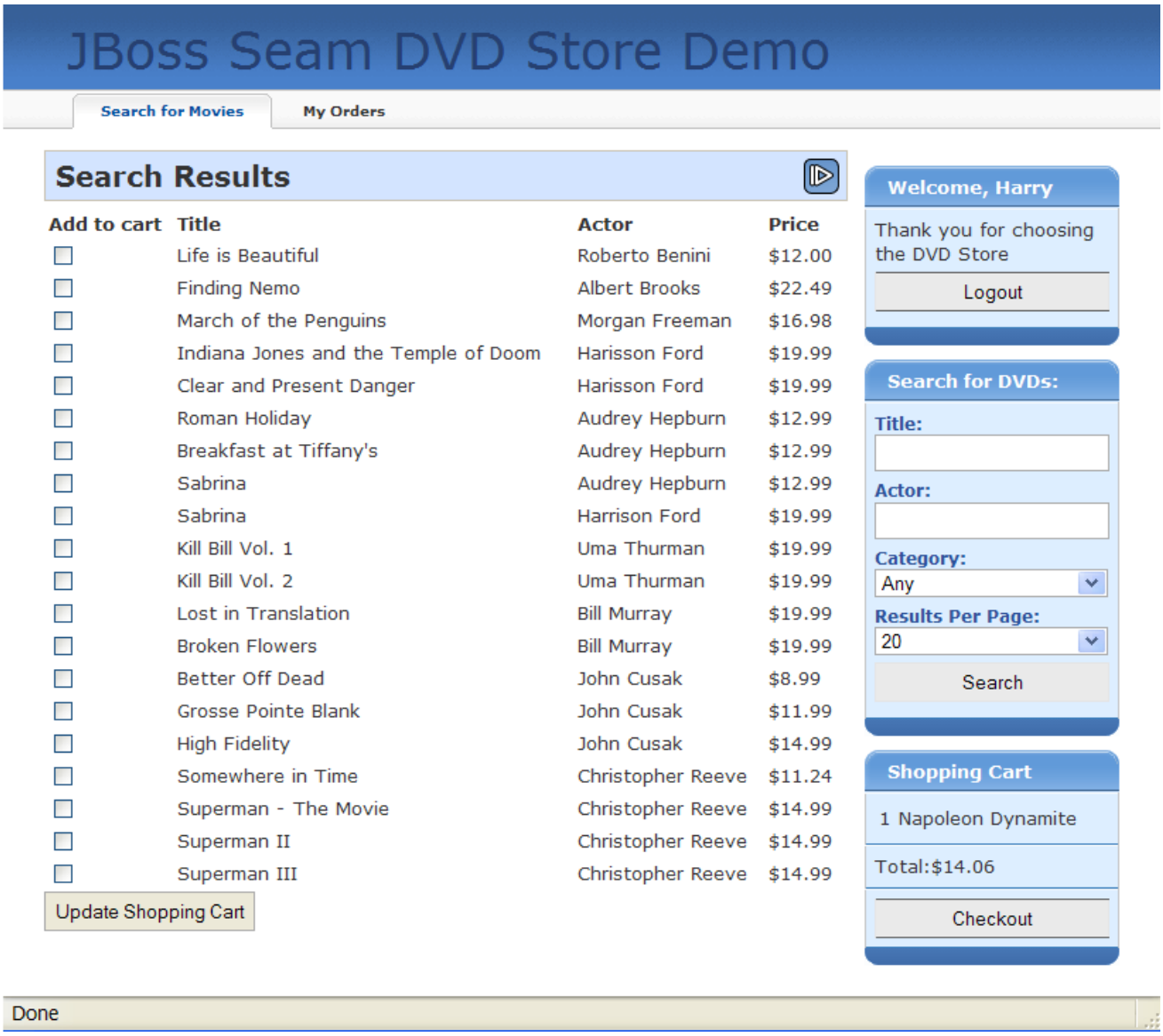
3 By simply annotating the cancellation action with **`@End(root=true, beforeRedirect=true)`** we can easily destroy and release all state associated with the workspace prior to redirecting the user back to the hotel selection view.

Feel free to deploy the application and test it yourself. Open many windows or tabs, and attempt combinations of various hotel and room preferences. Confirming a booking will always result in the correct hotel and room preference with the nested conversation model.

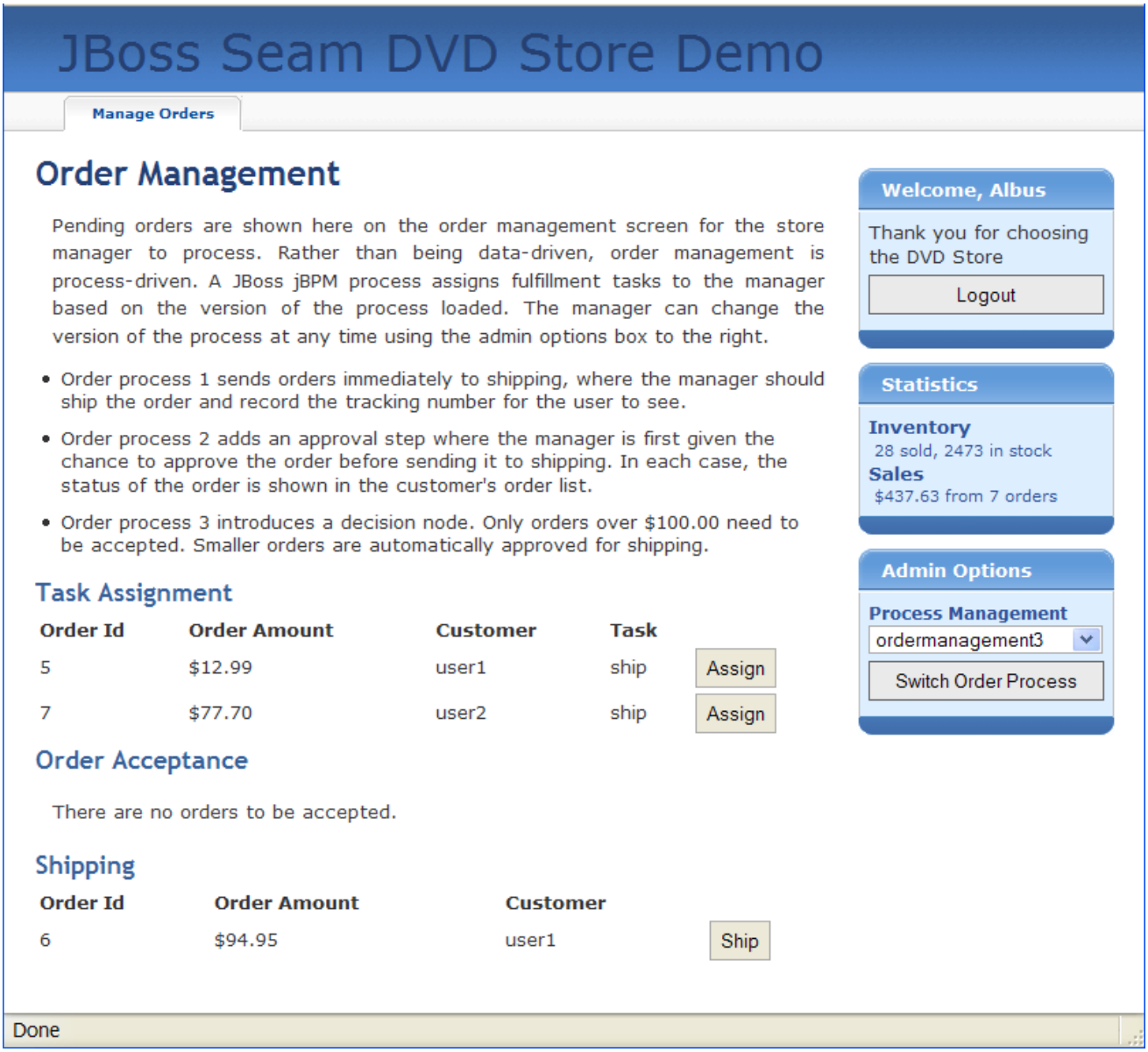
1.8. A COMPLETE APPLICATION FEATURING SEAM AND JBPM: THE DVD STORE EXAMPLE

The DVD Store demo application shows the practical usage of jBPM for both task management and pageflow.

The user screens take advantage of a jPDL pageflow to implement search and shopping cart functionality.



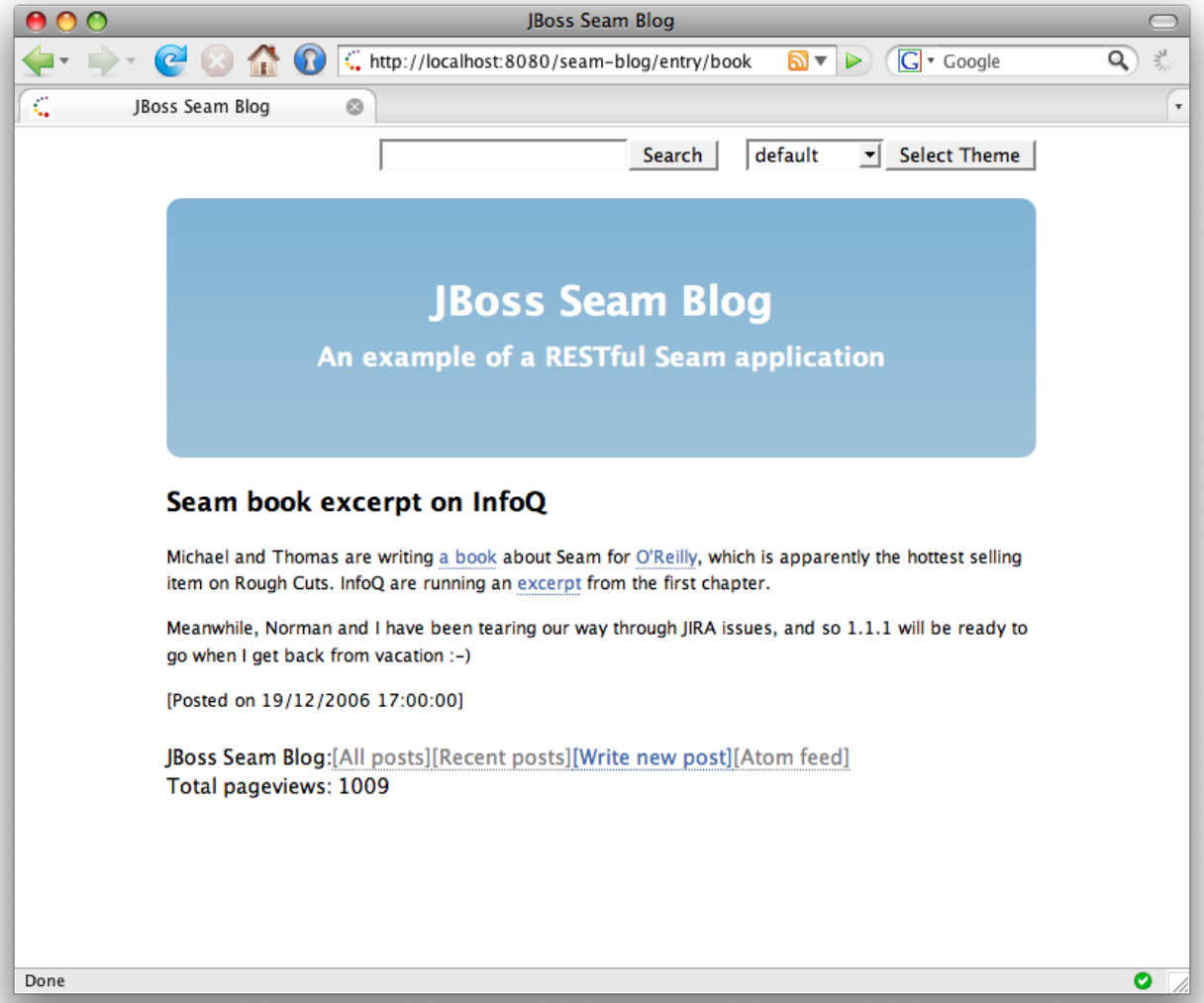
The administration screens use jBPM to manage the approval and shipping cycle for orders. The business process can even be changed dynamically by selecting a different process definition.



The Seam DVD Store demo can be run from the **`dvdstore`** directory, as with previous applications.

1.9. BOOKMARKABLE URLS WITH THE BLOG EXAMPLE

Seam makes it easy to implement applications which keep state on the server side. However, server-side state is not always appropriate, particularly for functionality that serves up content. For this, application state is often stored as part of the URL, so that any page can be accessed through a bookmark at any time. The blog example shows how to implement an application that supports bookmarking throughout, even on the search results page. This example demonstrates Seam's management of application state in the URL.



The blog example demonstrates the use of "pull"-style model view control (MVC), where the view pulls data from components as it is being rendered rather than using action listener methods to retrieve and prepare data for the view.

1.9.1. Using "pull"-style MVC

This snippet from the **`index.xhtml`** facelets page displays a list of recent blog entries:

```
<h:dataTable value="#{blog.recentBlogEntries}" var="blogEntry" rows="3">
  <h:column>
    <div class="blogEntry">
      <h3>#{blogEntry.title}</h3>
      <div>
        <s:formattedText value="#{blogEntry.excerpt==null ?
                                  blogEntry.body : blogEntry.excerpt}"/>
      </div>
      <p>
        <s:link view="/entry.xhtml" rendered="#{blogEntry.excerpt!=null}"
                propagation="none" value="Read more...">
          <f:param name="blogEntryId" value="#{blogEntry.id}"/>
        </s:link>
      </p>
      <p>
        [Posted on 
```

```
        <h:outputText value="#{blogEntry.date}">
          <f:convertDateTime timeZone="#{blog.timeZone}"
                             locale="#{blog.locale}" type="both"/>
          </h:outputText>]
           
          <s:link view="/entry.xhtml" propagation="none" value="[Link]">
            <f:param name="blogEntryId" value="#{blogEntry.id}"/>
          </s:link>
      </p>
    </div>
  </h:column>
</h:dataTable>
```

If we navigate to this page from a bookmark, the **`#{blog.recentBlogEntries}`** data used by the **`<h:dataTable>`** is retrieved lazily — "pulled" — when required, by a Seam component named **`blog`**. This flow of control is the reverse of that used in traditional action-based web frameworks like Struts.

Example 1.34.

```
 @Name("blog")
@Scope(ScopeType.STATELESS)
@AutoCreate
public class BlogService
{
   @In EntityManager entityManager;
```

(1)

```
   @Unwrap
```

(2)

```
   public Blog getBlog()
   {
      return (Blog) entityManager.createQuery("select distinct b from
Blog b left join fetch b.blogEntries")
         .setHint("org.hibernate.cacheable", true)
         .getSingleResult();
   }
}
```

(1)	This component uses a *seam-managed persistence context*. Unlike the other examples we've seen, this persistence context is managed by Seam, instead of by the EJB3 container. The persistence context spans the entire web request, allowing us to avoid any exceptions that occur when accessing unfetched associations in the view.
(2)	The **`@Unwrap`** annotation tells Seam to provide the return value of the method — the **`Blog`** — instead of the actual **`BlogService`** component to clients. This is the Seam *manager component pattern*.

This will store basic view content, but to bookmark form submission results like a search results page, there are several other required definitions.

1.9.2. Bookmarkable search results page

The blog example has a small form at the top right of each page that allows the user to search for blog entries. This is defined in **menu.xhtml**, which is included by the Facelets template **template.xhtml**:

```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}"/>
    <h:commandButton value="Search" action="/search.xhtml"/>
  </h:form>
</div>
```

To implement a bookmarkable search results page, after the search form submission is processed, we must perform a browser redirect. Because the JSF view ID is used as the action outcome, Seam automatically redirects to the view ID when the form is submitted. We could also have defined a navigation rule as follows:

```
<navigation-rule>
  <navigation-case>
    <from-outcome>searchResults</from-outcome>
    <to-view-id>/search.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

In that case, the form would have looked like this:

```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}"/>
    <h:commandButton value="Search" action="searchResults"/>
  </h:form>
</div>
```

However, to get a bookmarkable URL like **http://localhost:8080/seam-blog/search/**, the values submitted with the form must be included in the URL. There is no easy way to do this with JSF, but with Seam, only two features are required: *page parameters* and *URL rewriting*. Both are defined here in **WEB-INF/pages.xml**:

```
 <pages>
  <page view-id="/search.xhtml">
    <rewrite pattern="/search/{searchPattern}"/>
    <rewrite pattern="/search"/>

    <param name="searchPattern" value="#{searchService.searchPattern}"/>

  </page>
  ...
</pages>
```

The page parameter instructs Seam to link the **searchPattern** request parameter to the value held by **#{searchService.searchPattern}**, whenever the search page is requested, and whenever a link to the search page is generated. Seam takes responsibility for maintaining the link between URL state and application state.

The URL for a search on the term ***book*** would ordinarily be **http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book**. Seam can simplify this URL by using a rewrite rule. The first rewrite rule, for the pattern **/search/{searchPattern}**, states that whenever a URL for search.xhtml contains a searchPattern request parameter, that URL can be compressed into a simplified URL. So, the earlier URL (**http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern= book**) can instead be written as **http://localhost:8080/seam-blog/search/book**.

As with page parameters, rewriting URLs is bidirectional. This means that Seam forwards requests for the simplified URL to the correct view, and it automatically generates the simplified view — users need not construct URLs. The entire process is handled transparently. The only requirement for rewriting URLs is to enable the rewrite filter in **components.xml**:

```
<web:rewrite-filter view-mapping="/seam/*" />
```

The redirect takes us to the **search.xhtml** page:

```
<h:dataTable value="#{searchResults}" var="blogEntry">
 <h:column>
   <div>
      <s:link view="/entry.xhtml" propagation="none"
              value="#{blogEntry.title}">
        <f:param name="blogEntryId" value="#{blogEntry.id}"/>
      </s:link>
      posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timeZone="#{blog.timeZone}"
                           locale="#{blog.locale}" type="both"/>
      </h:outputText>
    </div>
  </h:column>
</h:dataTable>
```

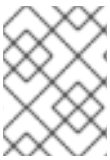
Again, this uses "pull"-style MVC to retrieve the search results with Hibernate Search.

```
@Name("searchService")
public class SearchService {

  @In
  private FullTextEntityManager entityManager;

  private String searchPattern;

  @Factory("searchResults")
  public List<BlogEntry> getSearchResults() {
  if (searchPattern==null || "".equals(searchPattern) )
    {
      searchPattern = null;
      return entityManager.createQuery(
                "select be from BlogEntry be order by date desc"
```

```
                ).getResultList();
    }
      else
        {
          Map<String,Float> boostPerField = new HashMap<String,Float>();
          boostPerField.put( "title", 4f );
          boostPerField.put( "body", 1f );
          String[] productFields = {"title", "body"};
          QueryParser parser = new MultiFieldQueryParser(productFields,
                               new StandardAnalyzer(), boostPerField);
          parser.setAllowLeadingWildcard(true);
          org.apache.lucene.search.Query luceneQuery;
          try
            {
              luceneQuery = parser.parse(searchPattern);
            }
          catch (ParseException e)
            {
              return null;
            }

          return entityManager
            .createFullTextQuery(luceneQuery, BlogEntry.class)
            .setMaxResults(100)
            .getResultList();
        }
    }

    public String getSearchPattern()
    {
      return searchPattern;
    }

    public void setSearchPattern(String searchPattern)
    {
      this.searchPattern = searchPattern;
    }

}
```

1.9.3. Using "push"-style MVC in a RESTful application

Push-style MVC is sometimes used to process RESTful pages, so Seam provides the notion of a *page action*. The blog example uses a page action for the blog entry page, **`entry.xhtml`**.



NOTE

We use push-style for the sake of an example, but this particular function would be simpler to implement with pull-style MVC.

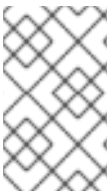
The **`entryAction`** component works much like an action class in a traditional push-MVC action-oriented framework like Struts.

```
@Name("entryAction")
```

```
@Scope(STATELESS)
public class EntryAction
{
    @In Blog blog;

    @Out BlogEntry blogEntry;

    public void loadBlogEntry(String id) throws EntryNotFoundException {
        blogEntry = blog.getBlogEntry(id);
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }

}
```

Page actions are also declared in **pages.xml**:

```
<pages>
  ...

  <page view-id="/entry.xhtml">
    <rewrite pattern="/entry/{blogEntryId}" />
    <rewrite pattern="/entry" />

    <param name="blogEntryId"
           value="#{blogEntry.id}"/>

    <action execute="#{entryAction.loadBlogEntry(blogEntry.id)}"/>
  </page>

  <page view-id="/post.xhtml" login-required="true">
    <rewrite pattern="/post" />

    <action execute="#{postAction.post}"
            if="#{validation.succeeded}"/>

    <action execute="#{postAction.invalid}"
            if="#{validation.failed}"/>

    <navigation from-action="#{postAction.post}">
      <redirect view-id="/index.xhtml"/>
    </navigation>
  </page>

  <page view-id="*">
    <action execute="#{blog.hitCount.hit}"/>
  </page>

</pages>
```



NOTE

Note that the example uses page actions for post validation and the pageview counter. Also note the use of a parameter in the page action method binding. This is not a standard JSF EL feature, but Seam allows it both here and in JSF method bindings.

When the **entry.xhtml** page is requested, Seam first binds the **blogEntryId** page parameter to the model. Remember that, because of URL rewriting, the blogEntryId parameter name won't appear in the URL. Seam then runs the page action, which retrieves the required data — the **blogEntry** — and places it in the Seam event context. Finally, it renders the following:

```
<div class="blogEntry">
  <h3>#{blogEntry.title}</h3>
  <div>
    <s:formattedText value="#{blogEntry.body}"/>
  </div>
  <p>
  [Posted on 
  <h:outputText value="#{blogEntry.date}">
     <f:convertDateTime timeZone="#{blog.timeZone}" locale="#
{blog.locale}" type="both"/>
  </h:outputText>]
  </p>
</div>
```

If the blog entry is not found in the database, the **EntryNotFoundException** exception is thrown. We want this exception to result in a 404 error, not a 505, so we annotate the exception class:

```
@ApplicationException(rollback=true)
@HttpError(errorCode=HttpServletResponse.SC_NOT_FOUND)
public class EntryNotFoundException extends Exception {
    EntryNotFoundException(String id) {
        super("entry not found: " + id);
    }
}
```

An alternative implementation of the example does not use the parameter in the method binding:

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction {
    @In(create=true)
        private Blog blog;

    @In @Out
    private BlogEntry blogEntry;

    public void loadBlogEntry() throws EntryNotFoundException {
        blogEntry = blog.getBlogEntry( blogEntry.getId() );
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }
}
```

```
<pages>
  ...
  <page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry}">
  <param name="blogEntryId" value="#{blogEntry.id}"/>
</page>
  ...
</pages>
```

The implementation used depends entirely upon personal preference.

The blog example also demonstrates very simple password authentication, posting to the blog, page fragment caching and atom feed generation.

CHAPTER 2. MIGRATION

If you already have a previous version of Seam installed, you will need to follow the instructions in this chapter to migrate to latest version (2.2.0), which is available with the JBoss Enterprise Application Platform.

If you are already using Seam 2.0, you can skip directly to Section 2.2, “Migrating from Seam 2.0 to Seam 2.1 or 2.2”. If you are currently using Seam 1.2.x, follow the instructions in both Section 2.1, “Migrating from Seam 1.2.x to Seam 2.0” and Section 2.2, “Migrating from Seam 2.0 to Seam 2.1 or 2.2”.

2.1. MIGRATING FROM SEAM 1.2.X TO SEAM 2.0

In this section, we show you how to migrate from Seam 1.2.x to Seam 2.0. We also list the changes to Seam components between versions.

2.1.1. Migrating to JavaServer Faces 1.2

Seam 2.0 requires JSF 1.2 to work correctly. We recommend Sun's JSF Reference Implementation (RI), which ships with most Java EE 5 application servers, including JBoss 4.2. To switch to the JSF RI, you will need to make the following changes to your **`web.xml`**:

- Remove the MyFaces **`StartupServletContextListener`**.
- Remove the AJAX4JSF filter, mappings, and **`org.ajax4jsf.VIEW_HANDLERS`** context parameter.
- Rename **`org.jboss.seam.web.SeamFilter`** as **`org.jboss.seam.servlet.SeamFilter`**.
- Rename **`org.jboss.seam.servlet.ResourceServlet`** as **`org.jboss.seam.servlet.SeamResourceServlet`**.
- Change the **`web-app`** version from **`2.4`** to **`2.5`**. In the namespace URL, change **`j2ee`** to **`javaee`**. For example:

```
<web-app 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-app_2_5.xsd" version="2.5">
  ...
</web-app>
```

As of Seam 1.2, you can declare **`SeamFilter`** in **`web.xml`** instead of explicitly declaring **`SeamExceptionFilter`** and **`SeamRedirectFilter`** in **`web.xml`**.

Client-side state saving is not required with the JSF RI, and can be removed. (Client-side state saving is defined by the **`javax.faces.STATE_SAVING_METHOD`** context parameter.

You will also need to make the following changes to **`faces-config.xml`**:

- Remove the **`TransactionalSeamPhaseListener`** or **`SeamPhaseListener`** declaration, if in use.

- Remove the **`SeamELResolver`** declaration, if in use.
- Change the **`SeamFaceletViewHandler`** declaration to the standard **`com.sun.facelets.FaceletViewHandler`**, and ensure it is enabled.
- Remove the Document Type Declaration (DTD) from the document and add the XML Schema declarations to the **`<faces-config>`** root tag, like so:

```
<faces-config version="1.2"
   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">
   ...
</faces-config>
```

2.1.2. Code Migration

Seam's built-in components have been reorganized to make them easier to learn and to isolate particular technology dependencies into specific packages.

- Persistence-related components have been moved to **`org.jboss.seam.persistence`**.
- jBPM-related components have been moved to **`org.jboss.seam.bpm`**.
- JSF-related components, most notably **`org.jboss.seam.faces.FacesMessages`**, have been moved to **`org.jboss.seam.faces`**.
- Servlet-related components have been moved to **`org.jboss.seam.web`**.
- Components related to asynchronicity have been moved to **`org.jboss.seam.async`**.
- Internationalization-related components have been moved to **`org.jboss.seam.international`**.
- The Pageflow component has been moved to **`org.jboss.seam.pageflow`**.
- The Pages component has been moved to **`org.jboss.seam.navigation`**.

Any code that depends upon these APIs will need to be altered to reflect the new Java package names.

Annotations have also been reorganized:

- BPM-related annotations are now included in the **`org.jboss.seam.annotations.bpm`** package.
- JSF-related annotations are now included in the **`org.jboss.seam.annotations.faces`** package.
- Interceptor annotations are now included in the **`org.jboss.seam.annotations.intercept`** package.
- Annotations related to asynchronicity are now included in the **`org.jboss.seam.annotations.async`** package.

- **@RequestParameter** is now included in the **org.jboss.seam.annotations.web** package.
- **@WebRemote** is now included in the **org.jboss.seam.annotations.remoting** package.
- **@Restrict** is now included in the **org.jboss.seam.annotations.security** package.
- Exception handling annotations are now included in the **org.jboss.seam.annotations.exception** package.
- Use **@BypassInterceptors** instead of **@Intercept(NEVER)**.

2.1.3. Migrating components.xml

The new packaging system outlined in the previous section means that you must update **components.xml** with the new schemas and namespaces.

Namespaces originally took the form **org.jboss.seam.foobar**. The new namespace format is **http://jboss.com/products/seam/foobar**, and the schema is of the form **http://jboss.com/products/seam/foobar-2.0.xsd**. You will need to update the format of the namespaces and schemas in your **components.xml** file so that the URLs correspond to the version of Seam that you wish to migrate to (2.0 or 2.1).

The following declarations must have their locations corrected, or be removed entirely:

- Replace **<core:managed-persistence-context>** with **<persistence:managed-persistence-context>** .
- Replace **<core:entity-manager-factory>** with **<persistence:entity-manager-factory>** .
- Remove **conversation-is-long-running** parameter from **<core:manager/>** element.
- Remove **<core:ejb/>** .
- Remove **<core:microcontainer/>** .
- Replace **<core:transaction-listener/>** with **<transaction:ejb-transaction/>** .
- Replace **<core:resource-bundle/>** with **<core:resource-loader/>** .

Example 2.1. components.xml Notes

Seam transaction management is now enabled by default. It is now controlled by **components.xml** instead of a JSF phase listener declaration in **faces-config.xml**. To disable Seam-managed transactions, use the following:

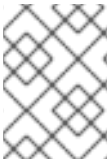
```
<core:init transaction-management-enabled="false"/>
```

The **expression** attribute on **event actions** has been deprecated in favor of **execute**. For example:

```
<event type="org.jboss.seam.security.notLoggedIn">
```

```
    <action execute="#{redirect.captureCurrentView}"/>
</event>
<event type="org.jboss.seam.loginSuccessful">
    <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

In Seam 2.2, security events use the **`org.jboss.seam.security>`** prefix instead of **`org.jboss.seam`** (for example, **`org.jboss.seam.security.notLoggedIn`**.)



NOTE

Instead of the **`org.jboss.seam.postAuthenticate`** event, use the **`org.jboss.seam.security.loginSuccessful`** event to return to the captured view.

2.1.4. Migrating to Embedded JBoss

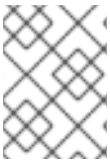
Embedded JBoss no longer supports the use of JBoss Embeddable EJB3 or JBoss Microcontainer. Instead, the new Embedded JBoss distribution provides a complete set of Java EE-compatible APIs with simplified deployment.

For testing, you will need to include the following in your classpath:

- the **`jar`**s in Seam's **`lib/`** directory
- the **`bootstrap/`** directory

Remove any references or artifacts relating to JBoss Embeddable EJB3, such as the **`embeddded-ejb`** directory and **`jboss-beans.xml`**. (You can use the Seam examples as reference.)

There are no longer any special configuration or packaging requirements for Tomcat deployment. To deploy with Tomcat, follow the instructions in the User Guide.



NOTE

Embedded JBoss can bootstrap a datasource from a **`-ds.xml`** file, so the **`jboss-beans.xml`** file is no longer required.

2.1.5. Migrating to jBPM 3.2

If you use jBPM for business processes as well as pageflows, you must add the **`tx`** service to **`jbpm.cfg.xml`**:

```
<service name="tx" factory="org.jbpm.tx.TxServiceFactory" />
```

2.1.6. Migrating to RichFaces 3.1

There has been a major reorganization of the codebase for both RichFaces and AJAX4JSF. The **`ajax4jsf.jar`** and **`richfaces.jar`** **`jar`**s have been replaced with the **`richfaces-api.jar`** (to be placed in the EAR **`lib/`** directory) and the **`richfaces-impl.jar`** and **`richfaces-ui.jar`** (to be placed in **`WEB-INF/lib`**).

<s:selectDate> has been deprecated in favor of **<rich:calendar>** . There will be no further development of **<s:selectDate>** . The styles associated with the data picker should be removed from your style sheet to reduce bandwidth use.

Check the RichFaces documentation for more information about changes to namespace and parameter names.

2.1.7. Changes to Components

Packaging Changes

All dependencies that were previously declared as modules in **application.xml** should now be placed in the **lib/** directory of your EAR, *except* **jboss-seam.jar**, which should be declared as an EJB module in **application.xml**.

Changes to the Seam User Interface

<s:decorate> has become a naming container. Client IDs have therefore changed from **fooForm:fooInput** to **fooForm:foo:fooInput**, assuming that the following has been declared:

```
<h:form id="fooForm">
  <s:decorate id="foo">
     <h:inputText id="fooInput" value="#{bean.property}"/>
   </s:decorate>
</h:form>
```

If you do not provide an ID for **<s:decorate>** , JSF will generate an ID automatically.

Changes to seam-gen

Since Seam 2.0.0.CR2, there have been changes to the organization of generated classes in **seam-gen** when **generate-entities** is executed.

Originally, classes were generated as follows:

```
        src/model/com/domain/projectname/model/EntityName.java
        src/action/com/domain/projectname/model/EntityNameHome.java
        src/action/com/domain/projectname/model/EntityNameList.java
```

Now, they are generated like this:

```
        src/model/com/domain/projectname/model/EntityName.java
        src/action/com/domain/projectname/action/EntityNameHome.java
        src/action/com/domain/projectname/action/EntityNameList.java
```

Home and Query objects are *action* components, not *model* components, and are therefore placed in the **action** package. This makes **generate-entities** conventions consistent with those of the **new-entity** command.

Model classes are listed separately because they cannot be hot-reloaded.

Since the testing system has changed from JBoss Embeddable EJB3 to Embedded JBoss, we recommend that you generate a project with **seam-gen** in Seam 2.x, and use its **build.xml** file as a base for new projects. If you have made extensive changes to the **build.xml**, you can focus on migrating only test-related targets.

For tests to work under Embedded JBoss, you need to change the value of the **<datasource>** element in **resources/META-INF/persistence-test.xml** (or **persistence-test-war.xml**) to **java:/DefaultDS**. Alternatively, you can deploy a **-ds.xml** file to the **bootstrap/deploy** folder and use the JNDI name defined in that file.

If you use the Seam 2.x **build.xml** as described, you will also require the **deployed-*.list** files, which define the **jar** files that are packaged in the EAR or WAR archive. These were introduced to remove the **jar** set from the **build.xml** file.

Add the following CSS to your style sheet to allow your style to accommodate a change in the RichFaces panel. Failing to add this code will mean that, in any page created by **generate-entities**, the *search criteria* block will bleed into the *results table*.

```
.rich-stglpanel-body {
        overflow: auto;
        }
```

2.2. MIGRATING FROM SEAM 2.0 TO SEAM 2.1 OR 2.2

This section describes the changes between Seam 2.0 and Seam 2.1 or 2.2. If you are trying to migrate from Seam 1.2.x, you will need to read the previous section, Section 2.1, “Migrating from Seam 1.2.x to Seam 2.0”, before following any steps outlined in this section.

2.2.1. Changes to dependency jar names

Refer to Table 2.1, “Included JARs” and Table 2.2, “Removed JARs” for a list of JARs which are included, and other that have been removed from the Seam framework.

Table 2.1. Included JARs

File Name	Description
lib/commons-codec.jar	Apache Commons codec library.
lib/commons-httpclient.jar	Apache Commons HTTP client library.
lib/drools-api.jar	Drools API.
lib/drools-decisiontables.jar	Drools decision tables.
lib/drools-templates.jar	Drools templates.
lib/ecj.jar	Eclipse compiler for Java (required for Drools Compiler).
lib/ehcache.jar	EH Cache (alternative cache provider)
lib/emma.jar	Emma Code Coverage library.
lib/gen/ant.jar	Ant build tool.

File Name	Description
lib/gen/cglib.jar	Code generation library.
lib/gen/common.jar	Eclipse Common library.
lib/gen/core.jar	Eclipse Core compiler.
lib/gen/darkX.jar	Richfaces DarkX skin library.
lib/gen/freemarker.jar	Freemarker templating engine.
lib/gen/glassX.jar	Richfaces GlassX skin library.
lib/gen/hibernate-tools.jar	Hibernate tools.
lib/gen/jboss-seam-gen.jar	JBoss Seam generator.
lib/gen/jtidy.jar	JTidy library.
lib/gen/laguna.jar	Richfaces Laguna skin library.
lib/gen/runtime.jar	Eclipse Runtime.
lib/gen/text.jar	Eclipse text library.
lib/guice.jar	Google dependency injection framework.
lib/hibernate-core.jar	Hibernate core.
lib/httpclient.jar	Apache HTTP client library.
lib/httpcore.jar	Apache HTTP commons library.
lib/itext-rtf.jar	IText RTF support for IText library (PDF generation)
lib/itext.jar	IText library (PDF generation)
lib/jaxrs-api.jar	JAVA Restful API for Web services
lib/jboss-seam-excel.jar	JBoss Seam Excel.
lib/jboss-seam-resteasy.jar	JBoss Seam Resteasy.
lib/jboss-transaction-api.jar	Java transaction API.
lib/jbosscache-core.jar	JBoss Cache provider.

File Name	Description
lib/jcip-annotations.jar	Java Concurrency In Practice annotations.
lib/jcl-over-slf4j.jar	Java Commons Logging bridge over SLF4J library.
lib/jettison.jar	Jettison provider for processing JSON.
lib/jxl.jar	Exel API library for Java.
lib/mvel2.jar	Expressions Language library.
lib/nekohtml.jar	NekoHTML library for parsing HTML.
lib/openid4java-nodeps.jar	OpenID library for Java.
lib/resteasy-atom-provider.jar	Resteasy Atom provider.
lib/resteasy-jaxb-provider.jar	Resteasy JAXB provider.
lib/resteasy-jaxrs.jar	Resteasy JAXRS API
lib/resteasy-jettison-provider.jar	Resteasy Jettison provider.
lib/slf4j-api.jar	SLF4J API.
lib/slf4j-log4j12.jar	SLF4J Log4j binding library 1.2.
lib/testng-jdk15.jar	TestNG test framework.

Many of the removed JARs have been excluded for several versions of the Platform. This list is mainly included for historical purposes.

Table 2.2. Removed JARs

JAR	Reason for Removal
activation.jar	Activation is bundled with Java 6, so it can be removed from the distribution.
commons-lang.jar	Commons Lang Library is no longer required.
geronimo-jms_1.1_spec.jar	
geronimo-jtaB_spec-1.0.1.jar	
hibernate3.jar	

JAR	Reason for Removal
`jboss-cache-jdk50.jar`	
`jboss-jmx.jar`	
`jboss-system.jar`	
`mvel.jar`	
`testng.jar`	

2.2.2. Changes to Components

Testing

`SeamTest` now boots Seam at the start of each suite, instead of the start of each class. This improves speed. Check the reference guide if you wish to alter the default.

Changes to DTD and Schema Format

Document Type Declarations (DTDs) for Seam XML files are no longer supported. XML Schema Declarations (XSDs) should be used for validation instead. Any file that uses Seam 2.0 XSDs should be updated to refer to the Seam 2.1 XSDs instead.

Changes to Exception Handling

Caught exceptions are now available in EL as **`#{org.jboss.seam.caughtException}`**. They are no longer available in **`#{org.jboss.seam.exception}`** form.

Changes to EntityConverter Configuration

You can now configure the entity manager used from the **`entity-loader`** component. For further details, see the documentation.

Changes in Managed Hibernate Sessions

Several aspects of Seam, including the Seam Application Framework, rely upon the existence of a common naming convention between the Seam-managed Persistence Context (JPA) and the Hibernate Session. In versions earlier than Seam 2.1, the name of the managed Hibernate Session was assumed to be **`session`**. Since **`session`** is an overloaded term in Seam and the Java Servlet API, the default has been changed to **`hibernateSession`** to reduce ambiguity. This means that, when you inject or resolve the Hibernate Session, it is much easier to identify the appropriate session.

You can use either of these approaches to inject the Hibernate Session:

```
@In private Session hibernateSession;
```

```
@In(name = "hibernateSession") private Session session;
```

If your Seam-managed Hibernate Session is still named **`session`**, you can inject the reference explicitly with the **`session`** property:

```
<framework:hibernate-entity-home session="#{session}".../>
      <transaction:entity-transaction session="#{session}".../>
```

Alternatively, you can override the **`getPersistenceContextName()`** method on any persistence controller in the Seam Application Framework with the following:

```
public String getPersistenceContextName() {
"session";
}
```

Changes to Security

The configuration for security rules in **`components.xml`** has changed for projects that use rule-based security. Previously, rules were configured as a property of the **`identity`** component:

```
<security:identity security-rules="#{securityRules}"
      authenticate-method="#{authenticator.authenticate}"/>
```

Seam 2.1 uses the **`ruleBasedPermissionResolver`** component for its rule-based permission checks. You must activate this component and register the security rules with it instead of with the **`identity`** component:

```
<security:rule-based-permission-resolver
      security-rules="#{securityRules}"/>
```

IMPORTANT

The definition of a *permission* has changed. Prior to Seam 2.1, a permission consisted of three elements:

- name
- action
- contextual object (optional)

The *name* would typically be the Seam component's name, entity class, or view ID. The *action* would be the method name, the JSF phase (restore or render), or an assigned term representing the intent of the activity. Optionally, one or more contextual objects can be inserted directly into the working memory to assist in decision-making. Typically, this would be the target of the activity. For example:

```
s:hasPermission('userManager', 'edit', user)
```

In Seam 2.1, permissions have been simplified so that they contain two elements:

- target
- action

The *target* replaces the *name* element, becoming the focus of the permission. The *action* still communicates the intent of the activity to be secured. Within the rules file, most checking now revolves around the *target* object. For example:

```
s:hasPermission(user, 'edit')
```

This change means that the rules can be applied more broadly, and lets Seam consult a persistent permission resolver (ACL) as well as the rule-based resolver.

Additionally, keep in mind that existing rules may behave oddly. This is because, given the following permission check format:

```
s:hasPermission('userManager', 'edit', user)
```

Seam transposes the following to apply the new permission format:

```
s:hasPemrission(user, 'edit')
```

Read the Security chapter for a complete overview of the new design.

Changes to Identity.isLoggedIn()

This method will no longer attempt to perform an authentication check if credentials have been set. Instead, it will return **`true`** if the user is currently unauthenticated. To make use of the previous behavior, use **`Identity.tryLogin()`** instead.

If you use Seam's token-based *Remember-Me* feature, you must add the following section to **`components.xml`** to ensure that the user is logged in automatically when the application is first accessed:

```
<event type="org.jboss.seam.security.notLoggedIn">
      <action execute="#{redirect.captureCurrentView}"/>
      <action execute="#{identity.tryLogin}"/>
      </event>
      <event type="org.jboss.seam.security.loginSuccessful">
      <action execute="#{redirect.returnToCapturedView}"/>
      </event>
```

Changes to iText (PDF)

The **documentStore** component has been moved from the external **pdf/itext** module into Seam itself. Any references to **pdf:document-store** in **components.xml** should therefore be replaced with **document:document-store**. Similarly, if your **web.xml** references **org.jboss.seam.pdf.DocumentStoreServlet**, you should change the reference to **org.jboss.seam.document.DocumentStoreServlet**.

Changes to Clustering

Seam's **ManagedEntityInterceptor** (previously **ManagedEntityIdentityInterceptor**) is now disabled by default. If you need the **ManagedEntityInterceptor** for clustered conversation failover, you can enable it in **components.xml** with the following:

```
<core:init>
      <core:interceptors>
      <value>org.jboss.seam.core.SynchronizationInterceptor</value>
      <value>org.jboss.seam.async.AsynchronousInterceptor</value>
      <value>org.jboss.seam.ejb.RemoveInterceptor</value>

<value>org.jboss.seam.persistence.HibernateSessionProxyInterceptor</value>

<value>org.jboss.seam.persistence.EntityManagerProxyInterceptor</value>
      <value>org.jboss.seam.core.MethodContextInterceptor</value>
      <value>org.jboss.seam.core.EventInterceptor</value>
      <value>org.jboss.seam.core.ConversationalInterceptor</value>
      <value>org.jboss.seam.bpm.BusinessProcessInterceptor</value>
      <value>org.jboss.seam.core.ConversationInterceptor</value>
      <value>org.jboss.seam.core.BijectionInterceptor</value>
      <value>org.jboss.seam.transaction.RollbackInterceptor</value>
      <value>org.jboss.seam.transaction.TransactionInterceptor</value>
      <value>org.jboss.seam.webservice.WSSecurityInterceptor</value>
      <value>org.jboss.seam.security.SecurityInterceptor</value>
      <value>org.jboss.seam.persistence.ManagedEntityInterceptor</value>
      </core:interceptors>
      </core:init>
```

Changes to Asynchronous Exception Handling

All asynchronous invocations are now wrapped by exception handling. By default, any exceptions that propagate out of an asynchronous call are caught and logged at the error level. You will find further information in Chapter 21, *Asynchronicity and messaging*.

Changes to Redeploy Events

The **org.jboss.seam.postInitialization** event is no longer called upon redeployment. **org.jboss.seam.postReInitialization** is called instead.

Changes to Cache Support

Cache support in Seam has been rewritten to support JBoss Cache 3.2, JBoss Cache 2 and Ehcache. Further information is available in Chapter 22, *Caching*.

The **<s:cache />** has not changed, but the **pojoCache** component can no longer be injected.

The **CacheProvider** provides a Map-like interface. The **getDelegate()** method can then be used to retrieve the underlying cache.

Changes to Maven Dependencies

The provided platform is now JBoss Enterprise Application Platform 5.1.0, so **javaassist:javaassist** and **dom4j:dom4j** are now marked as *provided*.

Changes to the Seam Application Framework

A number of properties now expect value expressions:

- **entityHome.createdMessage**
- **entityHome.updatedMessage**
- **entityHome.deletedMessage**
- **entityQuery.restrictions**

If you configure these objects with **components.xml**, no changes are necessary. If you configure the objects with JavaScript, you must create a value expression as follows:

```
public ValueExpression getCreatedMessage() {
return createValueExpression("New person #{person.firstName}
#{person.lastName} created");
}
```

CHAPTER 3. GETTING STARTED WITH SEAM-GEN

Seam includes a command line utility that makes it easy to set up an Eclipse project, generate some simple Seam skeleton code, and reverse-engineer an application from a preexisting database. This is an easy way to familiarize yourself with Seam.

In this release, **seam-gen** works best in conjunction with JBoss Enterprise Application Platform.

seam-gen can be used without Eclipse, but this tutorial focuses on using **seam-gen** with Eclipse for debugging and integration testing. If you would prefer not to use Eclipse, you can still follow this tutorial — all steps can be performed from the command line.

3.1. BEFORE YOU START

JBoss Enterprise Application Platform has sophisticated support for hot redeployment of **WAR**s and **EAR**s. Unfortunately, due to bugs in JVM, repeat redeployment of an EAR (common during development) uses all of the JVM's perm gen space. Therefore, we recommend running JBoss in a JVM with a large perm gen space during development.

If you are running JBoss from JBoss IDE, you can configure this in the server launch configuration, under "VM arguments". We suggest the following values:

```
-Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m
```

The minimum recommended values are:

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256m
```

If you are running EAP from the command line, you can configure the JVM options in **bin/run.conf**.

3.2. SETTING UP A NEW PROJECT

Task

Create a Seam project using the **seam-gen** command line tool.

Prerequisites:

- JDK 6 (see Section 37.1, “Java Development Kit Dependencies” for details)
- JBoss Enterprise Application Platform 5
- Ant 1.7.0
- A recent version of Eclipse.
- A recent version of the JBoss IDE plug-in for Eclipse.
- Add your EAP instance to the **JBoss Server View** in Eclipse.
- The TestNG plug-in for Eclipse.
- An EAP server running in debug mode.

- A command prompt in the directory where you unzipped the Seam distribution.

Procedure 3.1. Create a Seam project with seam-gen

1. Configure **seam-gen** for your environment by typing the following in your command line interface:

```
cd seam_distribution_dir; ./seam setup
```

The following output will be displayed:

```
Buildfile: build.xml

init:

setup:

    [echo] Welcome to seam-gen 2.2.2.EAP5 :-)

    [echo] Answer each question or hit ENTER to accept the default
(in brackets)

    [echo]
```

You will then receive the following prompts for required information:

1.
```
[input] Enter the directory where you want the project to be
created (should not contain spaces) [/home/<username>/projects]
[/home/<username>/projects]
```

Pressing **Enter** will accept the shown default (/home/<username>/projects).

Alternatively you can enter a different value.

2.
```
[input] Enter your JBoss AS home directory [/var/lib/jbossas]
[/var/lib/jbossas]
```

Again, press **Enter** to accept the default or enter a new value:

```
/home/<username>/path/to/jboss-as
```

3.
```
[input] Enter your JBoss AS domain [default] [default]
```

If you use a custom domain, enter it here.

4.
```
[input] Enter the project name [myproject] [myproject]
```

Enter the name of your project. Not entering a name will default to naming your project **myproject**.

For this example, the project will be called **helloworld**.

```
[echo] Accepted project name as: helloworld
```

5.
```
[input] Select a RichFaces skin [glassX] (blueSky, classic,
darkX, deepMarine, DEFAULT, emeraldTown, [glassX], japanCherry,
laguna, ruby, wine)
```

You can choose a skin for your project from the list presented. You can see that **`glassX`** is the default.

6.
```
[input] Is this project deployed as an EAR (with EJB components)
or a WAR (with no EJB support)? [war] (ear, [war])
```

Enter the appropriate archive format.

EAR projects support Enterprise JavaBeans 3.0 (EJB3) and require Java EE 5. **WAR** projects do not support EJB3, but can be deployed to a J2EE environment, and their packaging is simpler.

If you have an EJB3-ready application server like JBoss installed, choose **`ear`**. Otherwise, choose **`war`**. This tutorial assumes you are using an **EAR** deployment, but you can follow these steps even if your project is **WAR**-deployed.

7.
```
[input] Enter the base package name for your Java classes
[com.mydomain.helloworld] [com.mydomain.helloworld]

[input] Enter the Java package name for your session beans
[com.mydomain.helloworld.action] [com.mydomain.helloworld.action]

[input] Enter the Java package name for your entity beans
[com.mydomain.helloworld.model] [com.mydomain.helloworld.model]

[input] Enter the Java package name for your test cases
[com.mydomain.helloworld.test] [com.mydomain.helloworld.test]
```

You will be prompted to nominate the package names for your Java classes, session beans, entity beans and test cases.

8.
```
[input] What kind of database are you using? [hsql] ([hsql],
mysql, derby, oracle, postgres, mssql, db2, sybase, enterprisedb,
h2)
```

Nominate your database, **`mysql`**, for example. The default is **`hsql`**.

9.
```
[input] Enter the filesystem path to the JDBC driver jar [] []
```

Enter the path to your system's JDBC driver; **`/usr/share/java/mysql.jar`** for example.

You may see the following output:

```
[input] skipping input as property driver.license.jar.new has
already been set.
```

10.
```
[input] Enter the Hibernate dialect for your database
[org.hibernate.dialect.MySQLDialect]
[org.hibernate.dialect.MySQLDialect]
```

Nominate the appropriate Hibernate dialect.

11.
```
[input] Enter the JDBC DataSource class for your database
[com.mysql.jdbc.jdbc2.optional.MysqlDataSource]
[com.mysql.jdbc.jdbc2.optional.MysqlDataSource]

[input] Enter the JDBC driver class for your database
[com.mysql.jdbc.Driver] [com.mysql.jdbc.Driver]
```

You will be prompted to nominate the JDBC classes for your instance.

12.
```
[input] Enter the JDBC URL for your database [jdbc:mysql:///test]
[jdbc:mysql:///test]
```

The default URL will differ based on your response to the earlier database question.

13.
```
[input] Enter the database username [sa] [sa]
```

Nominate the username for access to the database, for example **root**.

14.
```
[input] Enter the database password [] []
```

Set the password for the above username.

You may see the following message:

```
[input] skipping input as property
hibernate.default_schema.entered has already been set.
```

15.
```
[input] Enter the database schema name (Enter '-' to clear
previous value) [] []

[input] Enter the database catalog name (Enter '-' to clear
previous value) [] []
```

Name the database schema and catalogue as prompted.

16.
```
[input] Are you working with tables that already exist in the
database? [n] (y, [n])
```

If you are working with an existing data model, make sure to tell **seam-gen** that tables already exist in the database.

17.
```
[input] Do you want to recreate the database tables and execute
import.sql each time you deploy? [n] (y, [n])
```

Again, answer as appropriate.

18. You will see the following output. Some details will differ in your instance, based on answers you provided to the above questions:

```
[propertyfile] Creating new property file: /home/path/to/jboss-
as/seam/seam-gen/build.properties

    [echo] Installing JDBC driver jar to JBoss AS

    [copy] Copying 1 file to /home/path/to/jboss-
as/server/default/lib

init:

init-properties:

    [echo] /home/path/to/jboss-as/

validate-workspace:

validate-project:

settings:

    [echo] JBoss AS home: /home/path/to/jboss-as/

    [echo] Project name: helloworld

    [echo] Project location: /home/path/to/projects/helloworld

    [echo] Project type: ear

    [echo] Action package: com.mydomain.helloworld.action

    [echo] Model package: com.mydomain.helloworld.model

    [echo] Test package: com.mydomain.helloworld.test

    [echo] JDBC driver class: com.mysql.jdbc.Driver

    [echo] JDBC DataSource class:
com.mysql.jdbc.jdbc2.optional.MysqlDataSource

    [echo] Hibernate dialect: org.hibernate.dialect.MySQLDialect

    [echo] JDBC URL: jdbc:mysql:///test

    [echo] Database username: root

    [echo] Database password:

    [echo]

    [echo] Type './seam create-project' to create the new project

BUILD SUCCESSFUL
Total time: 2 minutes 14 seconds
```

The time shown in this example is non-indicative of the actual time it will take to complete this task.

2. Create a new project in our Eclipse workspace directory by entering **one** of the following commands:

```
./seam create-project
```

Or

```
./seam new-project
```

You will see the following output:

```
Buildfile: build.xml
...
new-project:
  [echo] A new Seam project named 'helloworld' was created in the
C:\Projects directory
  [echo] Type 'seam explode' and go to
http://localhost:8080/helloworld
  [echo] Eclipse Users: Add the project into Eclipse using File >
New > Project and select General > Project (not Java Project)
  [echo] NetBeans Users: Open the project in NetBeans
BUILD SUCCESSFUL Total time: 7 seconds
C:\Projects\jboss-seam>
```

This copies the Seam **JAR**s, dependent **JAR**s and the JDBC driver **JAR** to a new Eclipse project. It generates all required resources and configuration files, a Facelets template file and stylesheet, along with Eclipse metadata and an Ant build script. The Eclipse project will be automatically deployed to an exploded directory structure in JBoss as soon as you add the project. To add the project, go to **New → Project... → General → Project → Next**, type the **`Project name`** (in this case, **`helloworld`**), and then click **`Finish`**. Do not select **`Java Project`** from the New Project wizard.

If your default JDK in Eclipse is not Java SE 6 JDK, you will need to select a compliant JDK. Go to **Project → Properties → Java Compiler**.

Alternatively, you can deploy the project from outside Eclipse by typing **`seam explode`**.

The welcome page can be found at **`http://localhost:8080/helloworld`**. This is a Facelets page (**`view/home.xhtml`**) created using the template found at **`view/layout/template.xhtml`**. You can edit the welcome page or the template in Eclipse, and see the results immediately by refreshing your browser.

XML configuration documents will be generated in the project directory. These may appear complicated, but they are comprised primarily of standard Java EE, and require little alteration, even between Seam projects.

There are three database and persistence configurations in the generated project. **`persistence-test.xml`** and **`import-test.sql`** are used while running TestNG unit tests against HSQLDB. The database schema and test data in **`import-test.sql`** is always exported to the database before tests are run. **`myproject-dev-ds.xml`**, **`persistence-dev.xml`** and **`import-dev.sql`** are used during application deployment to your development database. If you told **seam-gen** that you were working with

an existing database, the schema may be exported automatically upon deployment. **`myproject-prod-ds.xml`**, **`persistence-prod.xml`** and **`import-prod.sql`** are used during application deployment to your production database. The schema will not be exported automatically upon deployment.

3.3. CREATING A NEW ACTION

Task:

This task will show you how to create a simple web page with a stateless action method.

Prerequisites:

- JDK 6 (see Section 37.1, “Java Development Kit Dependencies” for details)
- JBoss Enterprise Application Platform 5.
- Ant 1.7.0.
- Recent versions of Eclipse, the JBoss IDE plug-in and the TestNG plug-in correctly installed.
- Add your JBoss installation to the Server View in Eclipse.
- JBoss Enterprise Application Platform running in debug mode.
- A command prompt in the directory where you unzipped the Seam distribution.

Procedure 3.2.

1. Execute the command:

```
seam new-action
```

2. Seam prompts for some information, and generates a new Facelets page and Seam component for your project.

```
Buildfile: build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
ping
    [input] Enter the local interface name [Ping]

    [input] Enter the bean class name [PingBean]

    [input] Enter the action method name [ping]

    [input] Enter the page name [ping]


setup-filters:
```

```
new-action:
     [echo] Creating a new stateless session bean component with an
action method
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
     [copy] Copying 1 file to C:\Projects\helloworld\view
     [echo] Type 'seam restart' and go to
http://localhost:8080/helloworld/ping.seam

BUILD SUCCESSFUL
Total time: 13 seconds
C:\Projects\jboss-seam>
```

3. Since we have added a new Seam component, it is necessary to restart the exploded directory deployment. You can do this by typing **`seam restart`**, or by running the **`restart`** target in the generated project's **`build.xml`** file from within Eclipse. Alternatively, you can edit the **`resources/META-INF/application.xml`** file in Eclipse.

   You do not need to restart JBoss each time you change the application.

4. Now go to **`http://localhost:8080/helloworld/ping.seam`** and click the button. The code behind this action is in the project **`src`** directory. Add a breakpoint to the **`ping()`** method, and click the button again.

5. Finally, locate the **`PingTest.xml`** file in the test package, and run the integration tests with the TestNG plug-in for Eclipse. You can also run the tests with **`seam test`** or the **`test`** target of the generated build.

3.4. CREATING A FORM WITH AN ACTION

The next step is to create a form. Type: **`seam new-form`**

```
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
hello
    [input] Enter the local interface name [Hello]

    [input] Enter the bean class name [HelloBean]

    [input] Enter the action method name [hello]

    [input] Enter the page name [hello]
```

```
setup-filters:

new-form:
     [echo] Creating a new stateful session bean component with an action
method
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\view
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello\test
     [echo] Type 'seam restart' and go to
http://localhost:8080/hello/hello.seam

BUILD SUCCESSFUL
Total time: 5 seconds
C:\Projects\jboss-seam>
```

Restart the application again, and go to **`http://localhost:8080/helloworld/hello.seam`**. Look at the generated code. Run the test. Experiment with adding new fields to the form and Seam component. (Remember to restart the deployment each time you alter the Java code.)

3.5. GENERATING AN APPLICATION FROM AN EXISTING DATABASE

Manually create tables in your database. (To switch to a different database, run **`seam setup`** again.) Now type:**`seam generate-entities`**

Restart the deployment, and go to **`http://localhost:8080/helloworld`**. You can browse the database, edit existing objects, and create new objects. The code generated here is very simple. Seam was designed so that data access code is easy to write by hand, even without the assistance of **seam-gen**.

3.6. GENERATING AN APPLICATION FROM EXISTING JPA/EJB3 ENTITIES

Place your existing, valid entity classes inside the **`src/main`** directory. Now, type: **`seam generate-ui`**

Restart the deployment, and go to **`http://localhost:8080/helloworld`**.

3.7. DEPLOYING THE APPLICATION AS AN EAR

Several changes are required before we can deploy the application with standard Java EE 5 packaging. First, remove the exploded directory by running **`seam unexplode`**. To deploy the EAR, either type **`seam deploy`** at the command prompt, or run the **`deploy`** target of the generated project build script. To undeploy, use **`seam undeploy`** or the **`undeploy`** target.

By default, the application deploys with the *dev profile*. The EAR includes the **`persistence-dev.xml`** and **`import-dev.sql`** files, and deploys **`myproject-dev-ds.xml`**. You can change the profile to *prod profile* by typing: **`seam -Dprofile=prod deploy`**

You can also define new deployment profiles for your application. Just add appropriately named files to your project — for example, **persistence-staging.xml**, **import-staging.sql** and **myproject-staging-ds.xml** — and select the name of the profile with **-Dprofile=staging**.

3.8. SEAM AND INCREMENTAL HOT DEPLOYMENT

Some support for incremental hot deployment is included during development when you deploy your Seam application as an exploded directory. Add the following line to **components.xml** to enable debug mode in Seam and Facelets:

```
<core:init debug="true">
```

WARNING

Hot deployment of Facelets does not work if the hot deployment scanner is not enabled for the server profile.

The following files may now be redeployed without requiring a full restart of the web application:

- any Facelets page
- any **pages.xml** file

If you want to change any Java code, you will still need to do a full restart of the application. In JBoss, this can be accomplished by touching the top-level deployment descriptor: **application.xml** for an EAR deployment, or **web.xml** for a WAR deployment.

Seam supports incremental redeployment of JavaBean components for a fast edit/compile/test cycle. To make use of this, the JavaBean components must be deployed into the **WEB-INF/dev** directory. Here, they will be loaded by a special Seam classloader instead of the WAR or EAR classloader.

This function has some limitations:

- The components must be JavaBean components — they cannot be EJB3 beans. (Seam is working to remove this limitation.)
- Entities can never be hot-deployed.
- Components deployed with **components.xml** cannot be hot-deployed.
- Hot-deployable components will not be visible to any classes deployed outside **WEB-INF/dev**.
- Seam debug mode must be enabled and **jboss-seam-debug.jar** must be included in **WEB-INF/lib**.
- The Seam filter must be installed in **web.xml**.
- You may see errors if the system is placed under any load and debug is enabled.

For WAR projects created with **seam-gen**, incremental hot deployment is available out of the box for classes in the `src/hot` source directory. However, **seam-gen** does not support incremental hot deployment for EAR projects.

CHAPTER 4. GETTING STARTED WITH JBOSS DEVELOPER STUDIO

JBoss Developer Studio is a collection of Eclipse plug-ins: a project creation wizard for Seam, a content assistant for the Unified Expression Language (EL) in both Facelets and Java, a graphical editor for jPDL, a graphical editor for Seam configuration files, support for running Seam integration tests from within Eclipse, and much more. For more information, review the *JBoss Developer Studio Getting Started Guide* available at http://docs.redhat.com/docs/en-US/JBoss_Developer_Studio/index.html.

Procedures to execute the following Seam tasks in **JBoss Developer Studio** are in the *Seam Developer Tools Reference Guide*:

- Setting up a Seam project.
- Creating a new Seam action.
- Creating a new form with an action.
- Generating an application from an existing database.

The *Seam Developer Tools Reference Guide* is available at http://docs.redhat.com/docs/en-US/JBoss_Developer_Studio/index.html.

4.1. HOT DEPLOYMENT WITH JBOSS DEVELOPER STUDIO

JBoss Developer Studio supports incremental hot deployment of any Facelets page and any **`pages.xml`** file out of the box. But if we want to change any Java code, we still need to do a full restart of the application by doing a **`Full Publish`**.

Seam supports incremental redeployment of JavaBean components for a fast edit/compile/test cycle. To make use of this, the JavaBean components must be deployed into the **`WEB-INF/dev`** directory. Here, they will be loaded by a special Seam classloader instead of the WAR or EAR classloader.

This function has some limitations:

- the components must be JavaBean components — they cannot be EJB3 beans. (Seam is working to remove this limitation.)
- entities can never be hot-deployed
- components deployed via **`components.xml`** may not be hot-deployed
- the hot-deployable components will not be visible to any classes deployed outside of **`WEB-INF/dev`**
- Seam debug mode must be enabled and **`jboss-seam-debug.jar`** must be in **`WEB-INF/lib`**
- the Seam filter must be installed in **`web.xml`**
- You may see errors if the system is placed under any load and debug is enabled.

For WAR projects created with JBoss Developer Studio, incremental hot deployment is available out of the box. However, JBoss Developer Studio does not support incremental hot deployment for EAR projects.

CHAPTER 5. THE CONTEXTUAL COMPONENT MODEL

Seam's two core concepts are the notions of a *context* and a *component*. Components are stateful objects, usually Enterprise JavaBeans (EJBs). An instance of a component is associated with a context, and assigned a name within that context. *Bijection* provides a mechanism for aliasing internal component names (instance variables) to contextual names, which allows component trees to be dynamically assembled and reassembled by Seam.

5.1. SEAM CONTEXTS

Seam has several built-in contexts, which are created and destroyed by the framework. The application does not control context demarcation via explicit Java API calls. Contexts are usually implicit. In some cases, however, contexts are demarcated with annotations.

There are a number of basic contexts:

- Stateless context
- Event (for instance, a request) context
- Page context
- Conversation context
- Session context
- Business process context
- Application context

Some of these contexts serve similar purposes in Servlet and related specifications. Two you may not have encountered previously are the *conversation context* and the *business process context*. One reason that state management in web applications is so fragile and error-prone is that the three built-in contexts (request, session, and application) are not especially meaningful for business logic. A user login session, for example, is an arbitrary construct in terms of the application workflow. Therefore, most Seam components are scoped to the conversation or business process contexts, since these are the most meaningful contexts in terms of the application.

5.1.1. Stateless context

Components which are truly stateless (primarily stateless session beans) always operate in the stateless context — the absence of a context, since the instance Seam resolves is not stored. Stateless components are arguably object-oriented, but they are developed regularly and thus form an important part of any Seam application.

5.1.2. Event context

The event context is the "narrowest" stateful context, and expands the notion of the web request to cover other event types. The event context associated with the life cycle of a JSF request is the most important example of an event context, and the one you will work with most often. Components associated with the event context are destroyed at the end of the request, but their state is available and well- defined for at least the life cycle of the request.

When you invoke a Seam component with RMI, or Seam Remoting, the event context is created and destroyed just for the invocation.

5.1.3. Page context

The page context allows you to associate state with a particular instance of a rendered page. You can initialize state in your event listener, or while rendering the page, and can then access it from any event that originates from that page. This is especially useful for functionality such as clickable lists, where the list is backed by changing data on the server side. The state is actually serialized to the client, so this construct is extremely robust with respect to multi-window operation and the back button.

5.1.4. Conversation context

The conversation context is a central concept to Seam. A *conversation* is a single unit of work from the user's perspective. In reality, it may span several interactions with a user — several requests and several data transactions. But to the user, a conversation solves a single problem. For example, the processes of booking a hotel, approving a contract, and creating an order are all conversations. It may help to think of a conversation as implementing a single "use case", although the relationship is not necessarily this exact.

A conversation holds state associated with the user's present task, in the current window. A single user may have multiple conversations in progress at any point in time, usually spanning multiple windows. The conversation context ensures that state from the different conversations does not collide and cause bugs.

Some conversations last only for a single request. Conversations that span multiple requests must be demarcated with annotations provided by Seam.

Some conversations are also *tasks*. A task is a conversation that is significant to a long-running business process, and can trigger a business process state transition when completed successfully. Seam provides a special set of annotations for task demarcation.

Conversations can be *nested*, with one conversation taking place inside a broader conversation. This is an advanced feature.

Between requests, conversation state is usually held in the Servlet session. Seam implements configurable *conversation timeout* to automatically destroy inactive conversations, which ensures that the state held by a single user login session does not continue to grow if a user abandons a conversation. In the same process, Seam serializes the processing of concurrent requests in the same long-running conversation context.

Alternatively, Seam can also be configured to store conversational state in the client browser.

5.1.5. Session context

A session context holds state associated with the user login session. There are some cases where it is useful for state to be shared between several conversations. However, session context should not usually hold components other than global information about the logged in user.

In a JSR-168 portal environment, the session context represents the portlet session.

5.1.6. Business process context

The business process context holds state associated with long-running business processes. This state is managed and made persistent by the BPM engine (in this case, JBoss jBPM). The business process spans multiple interactions with multiple users. State is shared between multiple users in a well-defined manner. The current task determines the current business process instance, and the business process life cycle is defined externally with *process definition language*, so there are no special annotations for business process demarcation.

5.1.7. Application context

The application context is the Servlet context from the Servlet specification. Application context is used primarily to hold static information such as configuration data, reference data or metamodels. For example, Seam stores its own configuration and metamodel in the application context.

5.1.8. Context variables

A context defines a namespace through a set of *context variables*. These work similarly to session or request attributes in the Servlet specification. Any value may be bound to a context variable, but they are usually bound to Seam component instances.

The context variable name identifies a component instance within a context. (The context variable name usually matches the component name.) You can programmatically access a named component instance in a particular scope with the **`Contexts`** class, which provides access to several thread-bound instances of the **`Context`** interface:

```
User user = (User) Contexts.getSessionContext().get("user");
```

You may also set or change the value associated with a name:

```
Contexts.getSessionContext().set("user", user);
```

However, components are usually obtained from a context via *injection*. Component instances are subsequently given to contexts via *outjection*.

5.1.9. Context search priority

Sometimes, component instances are obtained from a particular known scope. At other times, all stateful scopes are searched, in the following order of priority:

- Event context
- Page context
- Conversation context
- Session context
- Business process context
- Application context

You can perform a priority search by calling **`Contexts.lookupInStatefulContexts()`**. Whenever you access a component by name from a JSF page, a priority search occurs.

5.1.10. Concurrency model

Neither the Servlet, nor EJB specifications, define facilities for managing concurrent requests from the same client. The Servlet container lets all threads run concurrently, without ensuring thread-safeness. The EJB container allows concurrent access of stateless components, and throws an exception when multiple threads access a stateful session bean. This is sufficient for web applications based around fine-grained, synchronous requests. However, for modern applications, which frequently use asynchronous (AJAX) requests, concurrency support is vital. Therefore, Seam adds a concurrency management layer to its context model.

Session and application contexts are multi-threaded in Seam, allowing concurrent requests to be processed concurrently. Event and page contexts are single-threaded. Strictly, the business process context is multi-threaded, but concurrency here is rare, and can usually be disregarded. Seam serializes concurrent requests within a long-running conversation context in order to enforce a *single thread per conversation per process* model for the conversation context.

Because session context is multi-threaded and often contains volatile state, Seam always protects session-scoped components from concurrent access while Seam interceptors are enabled. If interceptors are disabled, any required thread safety must be implemented by the component itself. By default, Seam serializes requests to session-scoped session beans and JavaBeans, and detects and breaks any deadlocks that occur. However, this is not default behavior for application-scoped components, since they do not usually hold volatile state, and global synchronization is extremely expensive. Serialized threading models can be forced on any session bean or JavaBean component by adding the `@Synchronized` annotation.

This concurrency model means that AJAX clients can safely use volatile session and conversational state, without the need for any special work on the part of the developer.

5.2. SEAM COMPONENTS

Seam components are Plain Old Java Objects (POJOs). Specifically, they are JavaBeans, or Enterprise JavaBean 3.0 (EJB3) enterprise beans. While Seam does not require components to be EJBs, and can be used without an EJB3-compliant container, Seam was designed with EJB3 in mind, and includes deep integration with EJB3. Seam supports the following *component types*:

- EJB3 stateless session beans
- EJB3 stateful session beans
- EJB3 entity beans (for instance, JPA entity classes)
- JavaBeans
- EJB3 message-driven beans
- Spring beans (see Chapter 26, *Spring Framework integration*)

5.2.1. Stateless session beans

Stateless session bean components cannot hold state across multiple invocations, so they usually operate upon the state of other components in the various Seam contexts. They can be used as JSF action listeners, but cannot provide properties to JSF components for display.

Stateless session beans always exist in the stateless context. They can be accessed concurrently as a new instance is used for each request. The EJB3 container assigns instances to requests. (Normally, instances are allocated from a reuseable pool, so instance variables can retain data from previous uses of the bean.)

Seam stateless session bean components are instantiated with either `Component.getInstance()` or `@In(create=true)`. They should not be directly instantiated via JNDI look up or the `new` operator.

5.2.2. Stateful session beans

Stateful session bean components can not only hold state across multiple invocations of the bean, but also across multiple requests. Any application state that does not belong in the database should be held

by stateful session beans. This is a major difference between Seam and many other web application frameworks. Current conversation data should be stored in the instance variables of a stateful session bean bound to the conversation context, rather than in the **`HttpSession`**. This lets Seam manage state life cycle, and ensures there are no collisions between state relating to different concurrent conversations.

Stateful session beans are often used as JSF action listeners, and as backing beans to provide properties to JSF components for display or form submission.

By default, stateful session beans are bound to the conversation context. They may never be bound to the page, or to stateless contexts.

Seam serializes concurrent requests to session-scoped stateful session beans while Seam interceptors are enabled.

Seam stateful session bean components are instantiated with either **`Component.getInstance()`** or **`@In(create=true)`**. They should not be directly instantiated via JNDI look up or the**`new`** operator.

5.2.3. Entity beans

Entity beans can function as a Seam component when bound to a context variable. Because entities have a persistent identity in addition to their contextual identity, entity instances are usually bound explicitly in Java code, rather than being instantiated implicitly by Seam.

Entity bean components do not support bijection or context demarcation. Nor does invocation of an entity bean trigger validation.

Entity beans are not usually used as JSF action listeners, but often function as backing beans to provide properties to JSF components for display or form submission. They are commonly used as a backing bean coupled with a stateless session bean action listener to implement create/update/delete-type functionality.

By default, entity beans are bound to the conversation context, and can never be bound to the stateless context.

NOTE

In a clustered environment, it is less efficient to bind an entity bean directly to a conversation (or session-scoped Seam context variable) than it is to refer to the entity bean with a stateful session bean. Not all Seam applications define entity beans as Seam components for this reason.

Seam entity bean components are instantiated with **`Component.getInstance()`** or **`@In(create=true)`**, or directly instantiated with the **`new`** operator.

5.2.4. JavaBeans

JavaBeans are used similarly to stateless or stateful session beans. However, they do not provide functions such as declarative transaction demarcation, declarative security, efficient clustered state replication, EJB3 persistence, timeout methods, etc.

In a later chapter, we show you how to use Seam and Hibernate without an EJB container. In this case, components are JavaBeans rather than session beans.

NOTE

In a clustered environment, it is less efficient to cluster conversation- or session-scoped Seam JavaBean components than it is to cluster stateful session bean components.

By default, JavaBeans are bound to the event context.

Seam always serializes concurrent requests to session-scoped JavaBeans.

Seam JavaBean components are instantiated with **`Component.getInstance()`** or **`@In(create=true)`**. They should not be directly instantiated using the **`new`** operator.

5.2.5. Message-driven beans

Message-driven beans can function as Seam components. However, their call method differs from that of other Seam components — rather than being invoked with the context variable, they listen for messages sent to JMS queues or topics.

Message-driven beans cannot be bound to Seam contexts, nor can they access the session or conversation state of their *caller*. However, they do support bijection and some other Seam functionality.

Message-driven beans are never instantiated by the application; they are instantiated by the EJB container when a message is received.

5.2.6. Interception

To perform actions such as bijection, context demarcation, and validation, Seam must intercept component invocations. For JavaBeans, Seam controls component instantiation completely, and no special configuration is required. For entity beans, interception is not required, since bijection and context demarcation are not defined. For session beans, an EJB interceptor must be registered for the session bean component. This can be done with an annotation, as follows:

```
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login { ... }
```

However, it is better to define the interceptor in **`ejb-jar.xml`**:

```
  <interceptors>
  <interceptor>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor>
</interceptors>

<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

5.2.7. Component names

All Seam components require names. Assign a name with the **@Name** annotation:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login { ... }
```

This is the *Seam component name*, and does not relate to any other name defined by the EJB specification. However, Seam component names work like JSF managed bean names, and can be thought of in identical terms.

@Name is not the only way to define a component name, but the name must always be specified. No other Seam annotation will function if a name is not defined.

When Seam instantiates a component, it binds the new instance to a variable matching the component name in the component's configured scope. This is identical to JSF managed bean behavior, except that Seam lets you configure this mapping with annotations rather than XML. You can also programmatically bind a component to a context variable. This is useful if a particular component serves multiple roles within the system. For example, the current **User** might be bound to the **currentUser** session context variable, while a **User** that is the subject of some administration functionality might be bound to the **user** conversation context variable. Take care when binding programmatically, because it is possible to overwrite context variables that reference Seam components.

For very large applications, and for built-in Seam components, qualified component names are often used to avoid naming conflicts.

```
@Name("com.jboss.myapp.loginAction")
@Stateless
public class LoginAction implements Login { ... }
```

The qualified component name can be used both in Java code and in JSF's expression language:

```
<h:commandButton type="submit" value="Login"
   action="#{com.jboss.myapp.loginAction.login}"/>
```

Since this is noisy, Seam also provides a means of aliasing a qualified name to a simple name. Add a line like this to the **components.xml** file:

```
<factory name="loginAction" scope="STATELESS"
         value="#{com.jboss.myapp.loginAction}"/>
```

All built-in Seam components have qualified names, but can be accessed through their unqualified names with Seam's *namespace-import* feature. The **components.xml** file included in the Seam JAR defines the following namespaces:

```
<components xmlns="http://jboss.com/products/seam/components">
  <import>org.jboss.seam.core</import>
  <import>org.jboss.seam.cache</import>
  <import>org.jboss.seam.transaction</import>
  <import>org.jboss.seam.framework</import>
  <import>org.jboss.seam.web</import>
  <import>org.jboss.seam.faces</import>
  <import>org.jboss.seam.international</import>
```

```
    <import>org.jboss.seam.theme</import>
    <import>org.jboss.seam.pageflow</import>
    <import>org.jboss.seam.bpm</import>
    <import>org.jboss.seam.jms</import>
    <import>org.jboss.seam.mail</import>
    <import>org.jboss.seam.security</import>
    <import>org.jboss.seam.security.management</import>
    <import>org.jboss.seam.security.permission</import>
    <import>org.jboss.seam.captcha</import>
    <import>org.jboss.seam.excel.exporter</import>
    <!-- ... -->
</components>
```

When attempting to resolve an unqualified name, Seam will check each of these namespaces, in order. Additional application-specific namespaces can be included in your application's **components.xml** file.

5.2.8. Defining the component scope

The **@Scope** annotation lets us override the scope (context) of a component to define the context a component instance is bound to when instantiated by Seam.

```
@Name("user")
@Entity
@Scope(SESSION)
public class User { ... }
```

org.jboss.seam.ScopeType defines an enumeration of possible scopes.

5.2.9. Components with multiple roles

Some Seam component classes can fulfill multiple roles in the system. For example, the **User** class is usually a session-scoped component representing the current user, but in user administration screens becomes a conversation-scoped component. The **@Role** annotation lets us define an additional named role for a component, with a different scope — it lets us bind the same component class to different context variables. (Any Seam component *instance* can be bound to multiple context variables, but this lets us do it at the class level to take advantage of automatic instantiation.)

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Role(name="currentUser", scope=SESSION)
public class User { ... }
```

The **@Roles** annotation lets us specify additional roles as required.

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Roles({ @Role(name="currentUser", scope=SESSION),
         @Role(name="tempUser", scope=EVENT)})
public class User { ... }
```

5.2.10. Built-in components

Seam is implemented as a set of built-in interceptors and components. This makes it easy for applications to interact with built-in components at runtime, or to customize basic Seam functionality by replacing the built-in components with custom implementations. The built-in components are defined in the Seam namespace **`org.jboss.seam.core`**, and in the Java package of the same name.

The built-in components may be injected like any other Seam component, but they also provide convenient static **`instance()`** methods:

```
FacesMessages.instance().add("Welcome back, #{user.name}!");
```

5.3. BIJECTION

Dependency injection or *inversion of control* (IoC) allows one component to reference another by having the container "inject" the component into a setter method or instance variable. In previous dependency injection implementations, injection occurred at component construction, and the reference did not change for the lifetime of the component instance. This is reasonable for stateless components — from the client's perspective, all instances of a particular stateless component are interchangeable. However, Seam emphasizes the use of stateful components, so traditional dependency injection as a construct is less useful. Seam introduces the notion of *bijection* as a generalization of injection. In contrast to injection, bijection is:

contextual

Bijection is used to assemble stateful components from various different contexts. A component from a *wider* context can even refer to a component from a *narrower* context.

bidirectional

Values are injected from context variables into attributes of the invoked component, and returned (via outjection) to the context, allowing the invoked component to manipulate contextual variable values simply by setting its own instance variables.

dynamic

Since the value of contextual variables changes over time, and since Seam components are stateful, bijection takes place every time a component is invoked.

In essence, bijection lets you alias a context variable to a component instance variable, by specifying that the value of the instance variable is injected, outjected, or both. Annotations are used to enable bijection.

The **`@In`** annotation specifies that a value should be injected, either into an instance variable:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In User user;
  ...
}
```

or into a setter method:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
```

```
  User user;
  @In
  public void setUser(User user) { this.user=user; }
  ...
  }
```

By default, Seam performs a priority search of all contexts, using the name of the property or instance variable being injected. You may wish to specify the context variable name explicitly, using, for example, **`@In("currentUser")`**.

If you want Seam to create an instance of the component, where there is no existing component instance bound to the named context variable, you should specify **`@In(create=true)`**. If the value is optional (it can be null), specify **`@In(required=false)`**.

For some components, specifying **`@In(create=true)`** each time it is used can be repetitive. In such cases, annotate the component **`@AutoCreate`**. This way, it will always be created whenever required, even without the explicit use of **`create=true`**.

You can even inject the value of an expression:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In("#{user.username}") String username;
  ...
}
```

Injected values are disinjected (that is, set to **`null`**) immediately after method completion and outjection.

(More information about component life cycle and injection can be found in the next chapter.)

The **`@Out`** annotation specifies that an attribute should be outjected, either from an instance variable:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @Out User user;
  ...
}
```

or from a getter method:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  User user;

  @Out
  public User getUser() {
    return user;
  }
  ...
}
```

An attribute may be both injected and outjected:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In
  @Out User user;
  ...
}
```

or:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  User user;

  @In
  public void setUser(User user) {
    this.user=user;
  }

  @Out
  public User getUser() {
    return user; }
    ...
}
```

5.4. LIFE CYCLE METHODS

Session bean and entity bean Seam components support all common EJB3 life cycle callbacks (**`@PostConstruct`**, **`@PreDestroy`**, etc.), but Seam also supports the use of any of these callbacks with JavaBean components. However, since these annotations are not available in a J2EE environment, Seam defines two additional component life cycle callbacks, equivalent to **`@PostConstruct`** and **`@PreDestroy`**.

The **`@Create`** method is called after Seam instantiates a component. Components may define only one **`@Create`** method.

The **`@Destroy`** method is called when the context that the Seam component is bound to ends. Components may define only one **`@Destroy`** method.

In addition, stateful session bean components *must* define a method with no parameters, annotated **`@Remove`**. This method is called by Seam when the context ends.

Finally, the **`@Startup`** annotation can be applied to any application- or session-scoped component. The **`@Startup`** annotation tells Seam to instantiate the component immediately, when the context begins, instead of waiting until it is first referenced by a client. It is possible to control the order of instantiation of start up components by specifying **`@Startup(depends={...})`**.

5.5. CONDITIONAL INSTALLATION

The **`@Install`** annotation controls conditional installation of components that are required in some deployment scenarios and not in others. This is useful when you want to:

- mock out an infrastructural component in a test,
- change a component's implementation in certain deployment scenarios, or
- install some components only if their dependencies are available. (This is useful for framework authors.)

`@Install` lets you specify *precedence* and *dependencies*.

The precedence of a component is a number that Seam uses to decide which component to install when there are multiple classes with the same component name in the classpath. Seam will choose the component with the higher precedence. Some predefined precedence values are (in ascending order):

1. **`BUILT_IN`** — the lowest precedence components are the components built in to Seam.
2. **`FRAMEWORK`** — components defined by third-party frameworks may override built-in components, but are overridden by application components.
3. **`APPLICATION`** — the default precedence. This is appropriate for most application components.
4. **`DEPLOYMENT`** — for application components which are deployment-specific.
5. **`MOCK`** — for mock objects used in testing.

Suppose we have a component named **`messageSender`** that talks to a JMS queue.

```
@Name("messageSender")
public class MessageSender {

  public void sendMessage() {
    //do something with JMS
  }
}
```

In our unit tests, there is no available JMS queue, so we would like to stub out this method. We'll create a *mock* component that exists in the classpath when unit tests are running, but is never deployed with the application:

```
@Name("messageSender")
@Install(precedence=MOCK)
public class MockMessageSender extends MessageSender {
  public void sendMessage() {
    //do nothing!
  }
}
```

The **`precedence`** helps Seam decide which version to use when it finds both components in the classpath.

If we are able to control precisely which classes are in the classpath, this works well. But if we are writing a reuseable framework with many dependencies, we do not want to have to break that framework across multiple **`jar`**s. We want to be able to decide which components to install based on other installed components, and classes available in the classpath. The **`@Install`** annotation also controls this functionality. Seam uses this mechanism internally to enable the conditional installation of many built-in components.

5.6. LOGGING

Before Seam, even the simplest log message required verbose code:

```
private static final Log log = LogFactory.getLog(CreateOrderAction.class);

public Order createOrder(User user, Product product, int quantity) {
  if ( log.isDebugEnabled() ) {
    log.debug("Creating new order for user: " + user.username() +
              " product: " + product.name() + " quantity: " + quantity);
  }
  return new Order(user, product, quantity);
}
```

Seam provides a logging API that simplifies this code significantly:

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
  log.debug("Creating new order for user: #0 product: #1 quantity: #2",
            user.username(), product.name(), quantity);

  return new Order(user, product, quantity);
}
```

Except for entity bean components (which require the **log** variable to be static), this will work regardless of whether the **log** variable is declared static.

String concatenation occurs inside the **debug()** method, so the verbose **if (log.isDebugEnabled())** guard is unnecessary. Usually, we would not even need to explicitly specify the log category, since Seam knows where it is injecting the **log**.

If **User** and **Product** are Seam components available in the current contexts, the code is even more concise:

```
@Logger private Log log;
public Order createOrder(User user, Product product, int quantity) {
  log.debug("Creating new order for user: #{user.username}
            product: #{product.name} quantity: #0", quantity);
  return new Order(user, product, quantity);
}
```

Seam logging automatically chooses whether to send output to log4j or JDK logging — if log4j is in the classpath, it will be used; if not, Seam uses JDK logging.

5.7. THE MUTABLE INTERFACE AND @READONLY

Many application servers feature **HttpSession** clustering where changes to the state of mutable objects bound to the session are replicated only when **setAttribute** is called explicitly. This can lead to bugs that manifest only upon failover, which cannot be effectively tested during development. Further, the replication messages themselves are inefficient, since they contain the entire serialized object graph, bound to the session attribute.

EJB stateful session beans must perform automatic dirty checking (that is, they must automatically

detect object state changes to synchronize updated states with the database) and replicate mutable state. A sophisticated EJB container can introduce optimizations such as attribute-level replication. Unfortunately, not all Seam users will be working in an environment that supports EJB3. Therefore, Seam provides an extra layer of cluster-safe state management for session- and conversation-scoped JavaBean and entity bean components.

For session- or conversation-scoped JavaBean components, Seam automatically forces replication by calling **`setAttribute()`** once in every request where the component was invoked by the application. However, this strategy is inefficient for read-mostly components. Control this behavior by implementing the **`org.jboss.seam.core.Mutable`** interface, or by extending **`org.jboss.seam.core.AbstractMutable`** and writing your own dirty-checking logic inside the component. For example,

```
@Name("account")
public class Account extends AbstractMutable {
  private BigDecimal balance;
  public void setBalance(BigDecimal balance) {
    setDirty(this.balance, balance);
    this.balance = balance;
  }

  public BigDecimal getBalance() {
    return balance;
  }
  ...
}
```

Or, you can use the **`@ReadOnly`** annotation to achieve a similar effect:

```
@Name("account")
public class Account {
  private BigDecimal balance;
  public void setBalance(BigDecimal balance) {
    this.balance = balance;
  }

  @ReadOnly
  public BigDecimal getBalance() {
    return balance;
  }
  ...
}
```

For session- or conversation-scoped entity bean components, Seam automatically forces replication by calling **`setAttribute()`** once in every request, unless the (conversation-scoped) entity is currently associated with a Seam-managed persistence context, in which case replication is unnecessary. This strategy is not necessarily efficient, so session or conversation scope entity beans should be used with care. You can always write a stateful session bean or JavaBean component to "manage" the entity bean instance. For example:

```
@Stateful @Name("account")
public class AccountManager extends AbstractMutable {
  private Account account; // an entity bean
  @Unwrap
  public Account getAccount() {
```

```
    return account;
  }
  ...
}
```

Note that the **EntityHome** class in the Seam Application Framework is an excellent example of managing an entity bean instance using a Seam component.

5.8. FACTORY AND MANAGER COMPONENTS

It is often necessary to work with objects that are not Seam components, but we still prefer to be able to inject them into our components with **@In**, use them in value- and method-binding expressions, and tie them into the Seam context life cycle (**@Destroy**, for example). Therefore, Seam contexts can hold objects that are not Seam components, and Seam provides several features that simplify working with non-component objects bound to contexts.

The *factory component pattern* lets a Seam component act as the instantiator for a non-component object. A *factory method* will be called when a context variable is referenced but has no value bound to it. Factory methods are defined with the **@Factory** annotation. The factory method binds a value to the context variable, and determines the scope of the bound value. There are two styles of factory method. The first style returns a value, which is bound to the context by Seam:

```
@Factory(scope=CONVERSATION)
public List<Customer> getCustomerList() {
  return ... ;
}
```

The second style is a method of type **void**, which binds the value to the context variable itself:

```
@DataModel List<Customer> customerList;
@Factory("customerList")
public void initCustomerList() {
  customerList = ...  ;
}
```

In either case, the factory method is called when the **customerList** context variable is referenced, and its value is null. The factory method then has no further part in the life cycle of the value. The *manager component pattern* is an even more powerful pattern. In this case, a Seam component bound to a context variable manages the value of the context variable while remaining invisible to clients.

A manager component is any component with an **@Unwrap** method. This method returns the value that will be visible to clients, and is called every time a context variable is referenced.

```
@Name("customerList")
@Scope(CONVERSATION)
public class CustomerListManager {
  ...
  @Unwrap
  public List<Customer> getCustomerList() {
    return ... ;
  }
}
```

The manager component pattern is especially useful where more control is required over component life

cycle. For example, if you have a heavyweight object that needs a cleanup operation when the context ends, you could **@Unwrap** the object, and perform cleanup in the **@Destroy** method of the manager component.

```
@Name("hens")
@Scope(APPLICATION)
public class HenHouse {
  Set<Hen> hens;

  @In(required=false) Hen hen;

  @Unwrap
  public List<Hen> getHens()
  {
    if (hens == null) {
      // Setup our hens }
    return hens;
  }

  @Observer({"chickBorn", "chickenBoughtAtMarket"})
  public addHen() {
    hens.add(hen);
  }

  @Observer("chickenSoldAtMarket")
  public removeHen() {
    hens.remove(hen);
  }

  @Observer("foxGetsIn")
  public removeAllHens() {
    hens.clear();
  }

  ...
}
```

Here, the managed component observes many events that change the underlying object. The component manages these actions itself, and because the object is unwrapped each time it is accessed, a consistent view is provided.

CHAPTER 6. CONFIGURING SEAM COMPONENTS

Seam aims to minimize the need for XML-based configuration. However, there are various reasons you might want to configure Seam components with XML: to isolate deployment-specific information from the Java code, to enable the creation of reuseable frameworks, to configure Seam's built-in functionality, etc. Seam provides two basic approaches to component configuration: via property settings in a properties file or **web.xml**, and via **components.xml**.

6.1. CONFIGURING COMPONENTS VIA PROPERTY SETTINGS

You can provide Seam with configuration properties with either servlet context parameters (in system properties), or with a properties file named **seam.properties** in the root of the classpath.

The configurable Seam component must expose JavaBean-style property setter methods for the configurable attributes. That is, if a Seam component named **com.jboss.myapp.settings** has a setter method named **setLocale()**, we can provide either:

- a property named **com.jboss.myapp.settings.locale** in the **seam.properties** file,
- a system property named **org.jboss.seam.properties.com.jboss.myapp.settings.locale** via **-D** at start up, or
- the same system property as a Servlet context parameter.

Any of these will set the value of the **locale** attribute in the root of the class path.

The same mechanism is used to configure Seam itself. For example, to set conversation timeout, we provide a value for **org.jboss.seam.core.manager.conversationTimeout** in **web.xml**, **seam.properties**, or via a system property prefixed with **org.jboss.seam.properties**. (There is a built-in Seam component named **org.jboss.seam.core.manager** with a setter method named **setConversationTimeout()**.)

6.2. CONFIGURING COMPONENTS VIA COMPONENTS.XML

The **components.xml** file is more powerful than property settings. It lets you:

- configure components that have been installed automatically, including built-in components, and application components that have been annotated with **@Name** and picked up by Seam's deployment scanner.
- install classes with no **@Name** annotation as Seam components. This is most useful for infrastructural components which can be installed multiple times with different names (for example, Seam-managed persistence contexts).
- install components that do have a **@Name** annotation but are not installed by default because of an **@Install** annotation that indicates the component should not be installed.
- override the scope of a component.

A **components.xml** file appears in one of three locations:

- The **WEB-INF** directory of a **WAR**.

- The **META-INF** directory of a **JAR**.
- Any **JAR** directory containing classes with a **@Name** annotation.

Seam components are installed when the deployment scanner discovers a class with a **@Name** annotation in an archive with a **seam.properties** file, or a **META-INF/components.xml** file, unless the component also has an **@Install** annotation indicating that it should not be installed by default. The **components.xml** file handles special cases where the annotations must be overridden.

For example, the following **components.xml** file installs jBPM:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:bpm="http://jboss.com/products/seam/bpm">
  <bpm:jbpm/>
</components>
```

The following example also installs jBPM:

```
<components>
  <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

This example installs and configures two different Seam-managed persistence contexts:

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

    <persistence:managed-persistence-context name="customerDatabase"
         persistence-unit-jndi-name="java:/customerEntityManagerFactory"/>

    <persistence:managed-persistence-context name="accountingDatabase"
       persistence-unit-jndi-name="java:/accountingEntityManagerFactory"/>

</components>
```

This example also installs and configures two different Seam-managed persistence contexts:

```
<components>
  <component name="customerDatabase"

class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/customerEntityManagerFactory
    </property>
  </component>

  <component name="accountingDatabase"

class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/accountingEntityManagerFactory
```

```
    </property>
  </component>
</components>
```

The following examples create a session-scoped Seam-managed persistence context. (This is not recommended in practice.)

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context
          name="productDatabase" scope="session"
          persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase" scope="session"

class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/productEntityManagerFactory
    </property>
  </component>

</components>
```

The **auto-create** option is commonly used for infrastructural objects such as persistence contexts, removing the need to specify **create=true** explicitly when using the **@In** annotation.

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context
          name="productDatabase" auto-create="true"
          persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase"
    auto-create="true"
          class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/productEntityManagerFactory
    </property>
  </component>

</components>
```

The **<factory>** declaration specifies a value- or method-binding expression that will initialize the value of a context variable when it is first referenced.

```
<components>
  <factory name="contact" method="#{contactManager.loadContact}"
           scope="CONVERSATION"/>
</components>
```

You can create an *alias* (a second name) for a Seam component like so:

```
<components>
  <factory name="user" value="#{actor}" scope="STATELESS"/>
</components>
```

You can even create an alias for a commonly used expression:

```
<components>
  <factory name="contact" value="#{contactManager.contact}"
           scope="STATELESS"/>
</components>
```

auto-create="true" is often used with the **<factory>** declaration:

```
<components>
  <factory name="session" value="#{entityManager.delegate}"
           scope="STATELESS" auto-create="true"/>
</components>
```

The same **components.xml** file is sometimes used (with minor changes) during both deployment and testing. Seam allows wildcards of the form **@wildcard@** to be placed in **components.xml**, which can be replaced at deployment time by either your Ant build script, or providing a file named **components.properties** in the classpath. (The latter approach appears in the Seam examples.)

6.3. FINE-GRAINED CONFIGURATION FILES

If a large number of components require XML configuration, it is sensible to split **components.xml** into several smaller files. With Seam, configuration for a class named **com.helloworld.Hello** can be placed in a resource named **com/helloworld/Hello.component.xml**. (This pattern is also used in Hibernate.) The root element of the file may either be a **<components>** or **<component>** element.

<components> lets you define multiple components in the file:

```
<components>

  <component class="com.helloworld.Hello" name="hello">
    <property name="name">#{user.name}</property>
  </component>
  <factory name="message" value="#{hello.message}"/>

</components>
```

<component> only lets you configure one component, but is less verbose:

```
<component name="hello">
  <property name="name">#{user.name}</property>
</component>
```

The class name in the latter is implied by the file in which the component definition appears.

Alternatively, you may put configuration for all classes in the **com.helloworld** package in **com/helloworld/components.xml**.

6.4. CONFIGURABLE PROPERTY TYPES

Properties of string, primitive or primitive wrapper type are configured as follows:

- org.jboss.seam.core.manager.conversationTimeout 60000
- <core:manager conversation-timeout="60000"/>

```
        <component name="org.jboss.seam.core.manager">
        <property name="conversationTimeout">60000</property>
        </component>
```

Arrays, sets, and lists of strings or primitives are also supported:

```
org.jboss.seam.bpm.jbpm.processDefinitions
order.jpdl.xml,
return.jpdl.xml,
inventory.jpdl.xml
```

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>order.jpdl.xml</value>
    <value>return.jpdl.xml</value>
    <value>inventory.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

```
<component name="org.jboss.seam.bpm.jbpm">

  <property name="processDefinitions">
    <value>order.jpdl.xml</value>
    <value>return.jpdl.xml</value>
    <value>inventory.jpdl.xml</value>
  </property>

</component>
```

Even maps with String-valued keys and string or primitive values are supported:

```
<component name="issueEditor">

  <property name="issueStatuses">
    <key>open</key> <value>open issue</value>
    <key>resolved</key> <value>issue resolved by developer</value>
    <key>closed</key> <value>resolution accepted by user</value>
```

```
  </property>

</component>
```

When configuring multi-valued properties, Seam preserves the order of attributes set out in **components.xml** by default, unless **SortedSet**/**SortedMap** are used, in which case Seam refers to **TreeMap**/**TreeSet**. If the property has a concrete type (**LinkedList**, for example) Seam will use that type.

You can also override the type by specifying a fully qualified class name:

```
<component name="issueEditor">

  <property name="issueStatusOptions" type="java.util.LinkedHashMap">
    <key>open</key> <value>open issue</value>
    <key>resolved</key> <value>issue resolved by developer</value>
    <key>closed</key> <value>resolution accepted by user</value>
  </property>

</component>
```

Finally, you can link components with a value-binding expression. Note that since this occurs upon component instantiation, not invocation, this is quite different to injection with **@In**. It is more similar to the dependency injection facilities offered by traditional Inversion of Control containers such as JavaServer Faces (JSF) or Spring.

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
                               rule-base="#{policyPricingRules}"/>
```

```
<component name="policyPricingWorkingMemory"
           class="org.jboss.seam.drools.ManagedWorkingMemory">
  <property name="ruleBase">#{policyPricingRules}</property>
</component>
```

Seam also resolves EL expression strings prior to assigning the initial value to the bean property of the component, so some contextual data can be injected into components:

```
<component name="greeter" class="com.example.action.Greeter">
  <property name="message">
    Nice to see you, #{identity.username}!
  </property>
</component>
```

However, there is one important exception: if the initial value is assigned to either a Seam **ValueExpression** or **MethodExpression**, then the evaluation of the EL is deferred, and the appropriate expression wrapper is created and assigned to the property. The message templates on the **Home** component of the Seam Application Framework are a good example of this:

```
<framework:entity-home name="myEntityHome"
                       class="com.example.action.MyEntityHome"
                       entity-class="com.example.model.MyEntity"
                       created-message="'#{myEntityHome.instance.name}'
                       has been successfully added."/>
```

Within the component, you can access the expression string by calling **`getExpressionString()`** on either **`ValueExpression`** or **`MethodExpression`**. If the property is a **`ValueExpression`**, resolve the value with **`getValue()`**. If the property is a **`MethodExpression`**, invoke the method with **`invoke({Object arguments})`**. To assign a value to a **`MethodExpression`** property, the entire initial value must be a single EL expression.

6.5. USING XML NAMESPACES

Previous examples have alternated between two component declaration methods: with and without using XML namespaces. The following shows a typical **`components.xml`** file that does not use namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd">

  <component class="org.jboss.seam.core.init">
    <property name="debug">true</property>
    <property name="jndiPattern">@jndiPattern@</property>
  </component>

</components>
```

As you can see, this code is verbose. More importantly, the component and attribute names cannot be validated at development time.

Using namespaces gives us:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core
               http://jboss.com/products/seam/core-2.2.xsd
               http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd">

  <core:init debug="true" jndi-pattern="@jndiPattern@"/>

</components>
```

Although the schema declarations are verbose, the XML content itself is lean and easy to understand. The schemas provide detailed information about each component and the available attributes, allowing XML editors to offer intelligent auto-completion. Using namespaced elements makes it easier to generate and maintain correct **`components.xml`** files.

This works well for built-in Seam components, but for user components there are two available options. First, Seam supports mixing both models, allowing the use of generic **`<component>`** declarations for user components, and namespaced declarations for built-in components. More importantly, Seam lets you quickly declare namespaces for your own components.

Any Java package can be associated with an XML namespace by annotating the package with **@Namespace**. (Package-level annotations are declared in a file named **package-info.java** in the package directory.) An example of this from the seampay demo is:

```
@Namespace(value="http://jboss.com/products/seam/examples/ seampay")
package org.jboss.seam.example.seampay; import
org.jboss.seam.annotations.Namespace;
```

Using the namespaced style in **components.xml** is that simple. Now we can write:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >

  <pay:payment-home new-instance="#{newPayment}"
       created-message="Created a new payment to #{newPayment.payee}" />

  <pay:payment name="newPayment"
               payee="Somebody"
               account="#{selectedAccount}"
               payment-date="#{currentDatetime}"
               created-date="#{currentDatetime}" />
     ...
</components>
```

Or:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >

  <pay:payment-home>
    <pay:new-instance>"#{newPayment}"</pay:new-instance>
    <pay:created-message>
      Created a new payment to #{newPayment.payee}
    </pay:created-message>
  </pay:payment-home>

  <pay:payment name="newPayment">
    <pay:payee>Somebody"</pay:payee>
    <pay:account>#{selectedAccount}</pay:account>
    <pay:payment-date>#{currentDatetime}</pay:payment-date>
    <pay:created-date>#{currentDatetime}</pay:created-date>
   </pay:payment>
    ...
</components>
```

The previous examples illustrate the two usage models of a namespaced element. In the first declaration, **<pay:payment-home>** references the **paymentHome** component:

```
package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController extends EntityHome<Payment> {
  ...
```

```
}
```

The element name is the hyphenated form of the component name. The attributes of the element are the hyphenated forms of the property names.

In the second declaration, the **<pay:payment>** element refers to the **Payment** class in the **org.jboss.seam.example.seampay** package. In this case, **Payment** is an entity that is being declared as a Seam component:

```
package org.jboss.seam.example.seampay;
...
@Entity
public class Payment implements Serializable {
  ...
}
```

A schema is required for validation and auto-completion to work for user-defined components. Seam cannot yet generate a schema for a set of components automatically, so schema must be manually created. You can use schema definitions for standard Seam packages for guidance.

The following are the namespaces used by Seam:

- components — **http://jboss.com/products/seam/components**
- core — **http://jboss.com/products/seam/core**
- drools — **http://jboss.com/products/seam/drools**
- framework — **http://jboss.com/products/seam/framework**
- jms — **http://jboss.com/products/seam/jms**
- remoting — **http://jboss.com/products/seam/remoting**
- theme — **http://jboss.com/products/seam/theme**
- security — **http://jboss.com/products/seam/security**
- mail — **http://jboss.com/products/seam/mail**
- web — **http://jboss.com/products/seam/web**
- pdf — **http://jboss.com/products/seam/pdf**
- spring — **http://jboss.com/products/seam/spring**

CHAPTER 7. EVENTS, INTERCEPTORS AND EXCEPTION HANDLING

To complement the contextual component model, there are two further basic concepts that facilitate the extremely loose coupling distinctive of Seam applications. The first is a strong event model, where events are mapped to event listeners with method-binding expressions like those in JavaServer Faces (JSF). The second is the pervasive use of annotations and interceptors to apply cross-cutting concerns to components that implement business logic.

7.1. SEAM EVENTS

The Seam component model was developed for use with *event-driven applications*, specifically to enable the development of fine-grained, loosely-coupled components in a fine-grained eventing model. There are several event types in Seam:

- JSF events
- jBPM transition events
- Seam page actions
- Seam component-driven events
- Seam contextual events

Each of these events is mapped to Seam components with JSF EL method-binding expressions. For a JSF event, this is defined in the JSF template:

```
<h:commandButton value="Click me!" action="#{helloWorld.sayHello}"/>
```

For a jBPM transition event, it is specified in the jBPM process definition or pageflow definition:

```
<start-page name="hello" view-id="/hello.jsp">
    <transition to="hello">
    <action expression="#{helloWorld.sayHello}"/>
    </transition>
    </start-page>
```

More information about JSF events and jBPM events is available elsewhere. For now, we will concentrate upon the two additional event types defined by Seam.

7.2. PAGE ACTIONS

A Seam page action is an event occurring immediately before a JSF request is rendered. Declare page actions in **`WEB-INF/pages.xml`**. We can define a page action for a particular JSF view ID:

```
<pages>
    <page view-id="/hello.jsp" action="#{helloWorld.sayHello}"/>
    </pages>
```

Or we can use a **`*`** wildcard as a suffix to the **`view-id`** to specify an action that applies to all view IDs that match that pattern:

```
<pages>
    <page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
    </pages>
```

NOTE

If the **<page>** element is defined in a fine-grained page descriptor, the **view-id** attribute can be omitted, as it is already implied.

If multiple wildcarded page actions match the current view-id, Seam will call all the actions, in order of least-specific to most-specific.

The page action method can return a JSF outcome. If the outcome is not null, Seam uses the defined navigation rules to navigate to a view.

The view ID mentioned in the **<page>** element need not correspond to a real JSP or Facelets page. This way, we can reproduce the functionality of a traditional action-oriented framework like Struts or WebWork using page actions. This is useful for performing complex actions in response to non-Faces requests like HTTP GET.

Multiple or conditional page actions can be specified with the **<action>** tag:

```
<pages>
  <page view-id="/hello.jsp">
    <action execute="#{helloWorld.sayHello}"
            if="#{not validation.failed}"/>
    <action execute="#{hitCount.increment}"/>
  </page>
</pages>
```

Page actions are executed on both an initial (non-Faces) request and a postback (Faces) request. If you use the page action to load data, it may conflict with the standard JSF actions being executed on a postback. One way to disable the page action is to set up a condition that resolves to **true** only upon an initial request.

```
<pages>
  <page view-id="/dashboard.xhtml">
    <action execute="#{dashboard.loadData}"
            if="#{not FacesContext.renderKit.responseStateManager
                .isPostback(FacesContext)}"/>
  </page>
</pages>
```

This condition consults the **ResponseStateManager#isPostback(FacesContext)** to determine if the request is a postback. The ResponseStateManager is accessed using **FacesContext.getCurrentInstance().getRenderKit(). getResponseStateManager()**.

Seam offers a built-in condition that accomplishes this result less verbosely. You can disable a page action on a postback by setting the **on-postback** attribute to **false**:

```
<pages>
    <page view-id="/dashboard.xhtml">
    <action execute="#{dashboard.loadData}" on-postback="false"/>
```

```
    </page>
    </pages>
```

The **on-postback** attribute defaults to **true** to maintain backwards compatibility. However, you are more likely to use **false** more often.

7.3. PAGE PARAMETERS

A Faces request (a JSF form submission) encapsulates both an *action* (a method binding) and *parameters* (input value bindings). A page action can also require parameters.

Since non-Faces (GET) requests can be bookmarked, page parameters are passed as human-readable request parameters.

You can use page parameters with or without an action method.

7.3.1. Mapping request parameters to the model

Seam lets us provide a value binding that maps a named request parameter to an attribute of a model object.

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" value="#{person.firstName}"/>
    <param name="lastName" value="#{person.lastName}"/>
  </page>
</pages>
```

The **<param>** declaration is bidirectional, as with value bindings for JSF input:

- When a non-Faces (GET) request for the view ID occurs, Seam sets the value of the named request parameter to the model object, after performing appropriate type conversions.
- Any **<s:link>** or **<s:button>** includes the request parameter transparently. The parameter value is determined by evaluating the value binding during the render phase (when the **<s:link>** is rendered).
- Any navigation rule with a **<redirect/>** to the view ID includes the request parameter transparently. The parameter value is determined by evaluating the value binding at the end of the invoke application phase.
- The value is transparently propagated with any JSF form submission for the page with the given view ID. This means that view parameters behave like **PAGE**-scoped context variables for Faces requests.

However we arrive at **/hello.jsp**, the value of the model attribute referenced in the value binding is held in memory, without the need for a conversation (or other server-side state).

7.4. PROPAGATING REQUEST PARAMETERS

If only the **name** attribute is specified, the request parameter is propagated with the **PAGE** context (that is, it is not mapped to model property).

```
<pages>
```

```
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" />
    <param name="lastName" />
  </page>
</pages>
```

Page parameter propagation is especially useful when building multi-layered master-detail CRUD pages. You can use it to "remember" your view (for example, when pressing the Save button), and which entity you were editing.

- Any **<s:link>** or **<s:button>** transparently propagates the request parameter if that parameter is listed as a page parameter for the view.
- The value is transparently propagated with any JSF form submission for the page with the given view ID. (This means that view parameters behave like **PAGE**-scoped context variables for Faces requests.

Although this is fairly complex, it is definitely worthwhile to dedicate time to an understanding of page parameters. They are the most elegant method of propagating state across non-Faces requests. They are particularly useful in certain situations. For example, if we have search screens with bookmarkable results pages, page parameters let us write handling for both POST and GET requests in the same code. Page parameters eliminate repetitive request parameter-listing in the view definition, and simplify redirect code.

7.5. URL REWRITING WITH PAGE PARAMETERS

Rewriting occurs based on patterns found for views in **pages.xml**. Seam URL rewriting performs both incoming and outgoing URL rewriting based on the same pattern. A simple pattern for this process is:

```
<page view-id="/home.xhtml">
  <rewrite pattern="/home" />
</page>
```

In this case, any incoming request for **/home** will be sent to **/home.xhtml**. Any link generated that would normally point to **/home.seam** will instead be rewritten as **/home**. Rewrite patterns only match the portion of the URL before the query parameters, so **/home.seam?conversationId=13** and **/home.seam?color=red** will both be matched by this rewrite rule.

Rewrite rules can take query parameters into consideration, as shown with the following rules.

```
<page view-id="/home.xhtml">
  <rewrite pattern="/home/{color}" />
  <rewrite pattern="/home" />
</page>
```

In this case, an incoming request for **/home/red** will be served as if it were a request for **/home.seam?color=red**. Similarly, if color is a page parameter, an outgoing URL that would normally show as **/home.seam?color=blue** would instead be output as **/home/blue**. Rules are processed in order, so it is important to list more specific rules before more general rules.

Default Seam query parameters can also be mapped with URL rewriting, further concealing Seam's fingerprints. In the following example, **/search.seam?conversationId=13** would be written as **/search-13**.

```
<page view-id="/search.xhtml">
  <rewrite pattern="/search-{conversationId}" />
  <rewrite pattern="/search" />
</page>
```

Seam URL rewriting provides simple, bidirectional rewriting on a per-view basis. For more complex rewriting rules that cover non-Seam components, Seam applications can continue to use the **`org.tuckey.URLRewriteFilter`**, or apply rewriting rules at the web server.

To use URL rewriting, the Seam *rewrite filter* must be enabled. Rewrite filter configuration is discussed in Section 28.1.4.3, “URL rewriting”.

7.6. CONVERSION AND VALIDATION

You can specify a JSF converter for complex model properties, in either of the following ways:

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converterId="com.my.calculator.OperatorConverter"
           value="#{calculator.op}"/>
  </page>
</pages>
```

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converter="#{operatorConverter}"
           value="#{calculator.op}"/>
  </page>
</pages>
```

JSF validators, and **`required="true"`** may also be used, in either of the following ways:

```
<pages>
  <page view-id="/blog.xhtml">
    <param name="date" value="#{blog.date}"
           validatorId="com.my.blog.PastDate" required="true"/>
  </page>
</pages>
```

```
<pages>
  <page view-id="/blog.xhtml">
    <param name="date" value="#{blog.date}"
           validator="#{pastDateValidator}" required="true"/>
  </page>
</pages>
```

Model-based Hibernate validator annotations are automatically recognized and validated. Seam also provides a default date converter to convert a string parameter value to a date and back.

When type conversion or validation fails, a global **FacesMessage** is added to the **FacesContext**.

7.7. NAVIGATION

You can use standard JSF navigation rules defined in **faces-config.xml** in a Seam application. However, these rules have several limitations:

- It is not possible to specify that request parameters are used when redirecting.
- It is not possible to begin or end conversations from a rule.
- Rules work by evaluating the return value of the action method; it is not possible to evaluate an arbitrary EL expression.

Another problem is that "orchestration" logic is scattered between **pages.xml** and **faces-config.xml**. It is better to unify this logic under **pages.xml**.

This JSF navigation rule:

```
<navigation-rule>
  <from-view-id>/editDocument.xhtml</from-view-id>
  <navigation-case>
    <from-action>#{documentEditor.update}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/viewDocument.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

Can be rewritten as follows:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if-outcome="success">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

However, this method pollutes **DocumentEditor** with string-valued return values (the JSF outcomes). Instead, Seam lets us write:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}"
              evaluate="#{documentEditor.errors.size}">
    <rule if-outcome="0">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

Or even:

```
<page view-id="/editDocument.xhtml">
```

```
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

The first form evaluates a value binding to determine the outcome value used by the subsequent rules. The second approach ignores the outcome and evaluates a value binding for each possible rule.

When an update succeeds, we probably want to end the current conversation, like so:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

Since the conversation has ended, any subsequent requests will not know which document we are interested in. We can pass the document ID as a request parameter, which also makes the view bookmarkable:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml">
        <param name="documentId" value="#{documentEditor.documentId}"/>
      </redirect>
    </rule>
  </navigation>
</page>
```

Null outcomes are a special case in JSF, and are interpreted as instructions to redisplay the page. The following navigation rule matches any non-null outcome, but *not* the null outcome:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule>
      <render view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

To perform navigation when a null outcome occurs, use the following:

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <render view-id="/viewDocument.xhtml"/>
  </navigation>
</page>
```

The view ID can be given as a JSF EL expression:

```
<page view-id="/editDocument.xhtml">
  <navigation>
    <rule if-outcome="success">
      <redirect view-id="/#{userAgent}/displayDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

7.8. FINE-GRAINED FILES FOR DEFINING NAVIGATION, PAGE ACTIONS AND PARAMETERS

If you have a large number of different page actions and parameters — or even just a large number of navigation rules — it is sensible to split the declarations into several smaller files. You can define actions and parameters for a page with the view ID **`/calc/calculator.jsp`** in a resource named **`calc/calculator.page.xml`**. In this case, **`<page>`** is the root element, and the view ID is implied:

```
<page action="#{calculator.calculate}">
  <param name="x" value="#{calculator.lhs}"/>
  <param name="y" value="#{calculator.rhs}"/>
  <param name="op" converter="#{operatorConverter}" value="#
{calculator.op}"/>
</page>
```

7.9. COMPONENT-DRIVEN EVENTS

Seam components interact by calling each other's methods. Stateful components can even implement the observer/observable pattern. However, to enable more loosely-coupled interaction, Seam provides *component-driven events*.

We specify event listeners (observers) in **`components.xml`**.

```
<components>
  <event type="hello">
    <action execute="#{helloListener.sayHelloBack}"/>
    <action execute="#{logger.logHello}"/>
  </event>
</components>
```

Here, the *event type* is an arbitrary string.

When an event occurs, the actions registered for that event will be called in the order they appear in **`components.xml`**. Seam provides a built-in component to raise events.

```
@Name("helloWorld")
public class HelloWorld {
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
        Events.instance().raiseEvent("hello");
    }
}
```

You can also use an annotation, like so:

```
@Name("helloWorld")
public class HelloWorld {
    @RaiseEvent("hello")
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
    }
}
```

This event producer is not dependent upon event consumers. The event listener can now be implemented with absolutely no dependency upon the producer:

```
@Name("helloListener")
public class HelloListener {
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

The method binding defined above in **`components.xml`** maps the event to the consumer. If you prefer, you can also do this with annotations:

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

If you are familiar with component-driven events, you may be wondering about event objects. In Seam, event objects do not need to propagate state between the event producer and listener. State is held in the Seam contexts, and shared between components. However, if you do want to pass an event object, you can do so:

```
@Name("helloWorld")
public class HelloWorld {
    private String name;
    public void sayHello() {
        FacesMessages.instance().add("Hello World, my name is #0.", name);
        Events.instance().raiseEvent("hello", name);
    }
}
```

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack(String name) {
        FacesMessages.instance().add("Hello #0!", name);
    }
}
```

7.10. CONTEXTUAL EVENTS

Seam defines a number of built-in events that the application uses for certain kinds of framework integration. The events are:

Table 7.1. Contextual Events

Event	Description
org.jboss.seam.validationFailed	Called when JSFvalidation fails.
org.jboss.seam.noConversation	Called when there is nolong-running conversation and a long-running conversation is required.
org.jboss.seam.preSetVariable.<name>	called when the context variable <name> is set.
org.jboss.seam.postSetVariable.<name>	called when the context variable <name> is set.
org.jboss.seam.preRemoveVariable.<name>	Called when the context variable <name> is unset.
org.jboss.seam.postRemoveVariable.<name>	Called when the context variable <name> is unset.
org.jboss.seam.preDestroyContext.<SCOPE>	Called before the <SCOPE> context is destroyed.
org.jboss.seam.postDestroyContext.<SCOPE>	Called after the <SCOPE> context is destroyed.
org.jboss.seam.beginConversation	Called whenever along-running conversation begins.
org.jboss.seam.endConversation	Called whenever a long-running conversation ends.
org.jboss.seam.conversationTimeout	Called when a conversation timeout occurs. The conversation ID is passed as a parameter.
org.jboss.seam.beginPageflow	Called when a pageflowbegins.
org.jboss.seam.beginPageflow.<name>	Called when the pageflow <name> begins.
org.jboss.seam.endPageflow	Called when a pageflowends.
org.jboss.seam.endPageflow.<name>	Called when the pageflow <name> ends.

Event	Description
org.jboss.seam.createProcess.<name>	called when the process <name> is created.
org.jboss.seam.endProcess.<name>	called when the process <name> ends.
org.jboss.seam.initProcess.<name>	called when the process <name> is associated with the conversation.
org.jboss.seam.initTask.<name>	Called when the task <name> is associated with the conversation.
org.jboss.seam.startTask.<name>	Called when the task <name> is started.
org.jboss.seam.endTask.<name>	Called when the task <name> is ended.
org.jboss.seam.postCreate.<name>	Called when the component <name> is created.
org.jboss.seam.preDestroy.<name>	Called when the component <name> is destroyed.
org.jboss.seam.beforePhase	Called before the start of a JSF phase.
org.jboss.seam.afterPhase	Called after the end of a JSF phase.
org.jboss.seam.postInitialization	Called when Seam has initialized and started up all components.
org.jboss.seam.postReInitialization	Called when Seam has re-initialized and started up all components after a redeploy.
org.jboss.seam.exceptionHandled.<type>	Called when an uncaught exception is handled by Seam.
org.jboss.seam.exceptionHandled	Called when an uncaught exception is handled by Seam.
org.jboss.seam.exceptionNotHandled	Called when there was no handler for an uncaught exception.
org.jboss.seam.afterTransactionSuccess	Called when a transaction succeeds in the Seam Application Framework.
org.jboss.seam.afterTransactionSuccess.<name>	Called when a transaction succeeds in the Seam Application Framework managing the entity **<name>** .
org.jboss.seam.security.loggedOut	Called when a user logs out.

Event	Description
org.jboss.seam.security.loginFailed	Called when a user authentication attempt fails.
org.jboss.seam.security.loginSuccessful	Called when a user is successfully authenticated.
org.jboss.seam.security.notAuthorized	Called when an authorization check fails.
org.jboss.seam.security.notLoggedIn	Called when there is no authenticated user and authentication is required.
org.jboss.seam.security.postAuthenticate	Called after a user is authenticated.
org.jboss.seam.security.preAuthenticate	Called before attempting to authenticate a user.

Seam components observe these events just as they observe any other component-driven event.

7.11. SEAM INTERCEPTORS

EJB3 introduced a standard interceptor model for session bean components. To add an interceptor to a bean, you need to write a class with a method annotated **@AroundInvoke** and annotate the bean with an **@Interceptors** annotation that specifies the name of the interceptor class. For example, the following interceptor checks that the user is logged in before allowing invoking an action listener method:

```
public class LoggedInInterceptor {
    @AroundInvoke
    public Object checkLoggedIn(InvocationContext invocation)
        throws Exception {
        boolean isLoggedIn = Contexts.getSessionContext()
            .get("loggedIn")!=null;
        if (isLoggedIn) {
            //the user is already logged in return invocation.proceed();
        } else {
            //the user is not logged in, fwd to login page return "login";
        }
    }
}
```

To apply this interceptor to a session bean acting as an action listener, we must annotate the session bean **@Interceptors(LoggedInInterceptor.class)**. However, Seam builds upon the interceptor framework in EJB3 by allowing you to use **@Interceptors** as a meta-annotation for class level interceptors (those annotated **@Target(TYPE)**). In this example, we would create an **@LoggedIn** annotation, as follows:

```
@Target(TYPE)
@Retention(RUNTIME)
@Interceptors(LoggedInInterceptor.class)
    public @interface LoggedIn {}
```

We can now annotate our action listener bean with **@LoggedIn** to apply the interceptor.

```
@Stateless
@Name("changePasswordAction")
@LoggedIn
@Interceptors(SeamInterceptor.class)
public class ChangePasswordAction implements ChangePassword {
    ...
        public String changePassword() {
            ...
        }
}
```

Where interceptor order is important, add **@Interceptor** annotations to your interceptor classes to specify a particular order of interceptors.

```
@Interceptor(around={BijectionInterceptor.class,
                     ValidationInterceptor.class,
                     ConversationInterceptor.class},
    within=RemoveInterceptor.class)
public class LoggedInInterceptor {
    ...
}
```

You can even have a client-side interceptor, for built-in EJB3 functions:

```
@Interceptor(type=CLIENT)
public class LoggedInInterceptor {
    ...
}
```

EJB interceptors are stateful, and their life cycles match that of the component they intercept. For interceptors that do not need to maintain state, Seam allows performance optimization where **@Interceptor(stateless=true)** is specified.

Much of Seam's functionality is implemented as a set of built-in Seam interceptors, including the interceptors named in the previous example. These interceptors exist for all interceptable Seam components; you need not specify them explicitly through annotation.

Seam interceptors can also be used with JavaBean components.

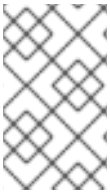
EJB defines interception not only for business methods (using **@AroundInvoke**), but also for the life cycle methods **@PostConstruct**, **@PreDestroy**, **@PrePassivate** and **@PostActive**. Seam supports these life cycle methods on both component and interceptor, not only for EJB3 beans, but also for JavaBean components (except **@PreDestroy**, which is not meaningful for JavaBean components).

7.12. MANAGING EXCEPTIONS

JSF has a limited ability to handle exceptions. To work around this problem, Seam lets you define treatment of an exception class through annotation, or through declaration in an XML file. This combines with the EJB3-standard **@ApplicationException** annotation, which specifies whether the exception should cause a transaction rollback.

7.12.1. Exceptions and transactions

EJB specifies well-defined rules to control whether an exception immediately marks the current transaction for rollback, when thrown by a business method of the bean. *System exceptions* always cause a transaction rollback. *Application exceptions* do not cause a rollback by default, but they will cause a rollback if **`@ApplicationException(rollback=true)`** is specified. (An application exception is any checked exception, or any unchecked exception annotated **`@ApplicationException`**. A system exception is any unchecked exception without an **`@ApplicationException`** annotation.)



NOTE

Marking a transaction for rollback is not the same as actually rolling back the transaction. The exception rules say that the transaction should be marked rollback only, but it may still be active after the exception is thrown.

Seam also applies the EJB3 exception rollback rules to Seam JavaBean components.

These rules apply only in the Seam component layer. When an exception occurs outside the Seam component layer, Seam rolls back any active transaction.

7.12.2. Enabling Seam exception handling

To enable Seam's exception handling, the master Servlet filter must be declared in **`web.xml`**:

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>*.seam</url-pattern>
</filter-mapping>
```

For the exception handlers to fire, you must disable Facelets development mode in **`web.xml`** and Seam debug mode in **`components.xml`**.

7.12.3. Using annotations for exception handling

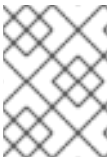
The following exception results in a HTTP 404 error whenever it propagates outside the Seam component layer. It does not roll back the current transaction immediately when thrown, but the transaction will be rolled back if it the exception is not caught by another Seam component.

```
@HttpError(errorCode=404)
public class ApplicationException extends Exception {
    ...
}
```

This exception results in a browser redirect whenever it propagates outside the Seam component layer. It also ends the current conversation. It causes an immediate rollback of the current transaction.

```
@Redirect(viewId="/failure.xhtml", end=true)
```

```
@ApplicationException(rollback=true)
public class UnrecoverableApplicationException extends RuntimeException {
    ...
}
```



NOTE

Seam cannot handle exceptions that occur during JSF's **RENDER_RESPONSE** phase, as it is not possible to perform a redirect once writing to the response has begun.

You can also use EL to specify the **viewId** to redirect to.

When this exception propagates outside the Seam component layer, it results in a redirect and a message to the user. It also immediately rolls back the current transaction.

```
@Redirect(viewId="/error.xhtml", message="Unexpected error")
public class SystemException extends RuntimeException {
    ...
}
```

7.12.4. Using XML for exception handling

Since annotations cannot be added to all exception classes, Seam also lets us specify this functionality in **pages.xml**.

```
<pages>
  <exception class="javax.persistence.EntityNotFoundException">
    <http-error error-code="404"/>
  </exception>

  <exception class="javax.persistence.PersistenceException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>Database access failed</message>
    </redirect>
  </exception>

  <exception>
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>Unexpected failure</message>
    </redirect>
  </exception>
</pages>
```

The final **<exception>** declaration does not specify a class, and acts as catch-all for any exception without specified handling via annotations or in **pages.xml**.

You can also use EL to specify the **view-id** to redirect to.

You can also access the handled exception instance through EL. Seam places it in the conversation context. For example, to access the exception message:

```
...
  throw new AuthorizationException("You are not allowed to do this!");

<pages>
  <exception class="org.jboss.seam.security.AuthorizationException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message severity="WARN">
        #{org.jboss.seam.handledException.message}
      </message>
    </redirect>
  </exception>
</pages>
```

`org.jboss.seam.handledException` holds the nested exception that was handled by an exception handler. The outermost (wrapper) exception is also available as **`org.jboss.seam.caughtException`**.

7.12.4.1. Suppressing exception logging

For the exception handlers defined in **`pages.xml`**, it is possible to declare the level at which the exception will be logged, or to suppress exception logging altogether. The **`log`** and **`log-level`** attributes are used to control exception logging. No log message will be generated when the specified exception occurs when **`log="false"`** is set, as shown here:

```
<exception class="org.jboss.seam.security.NotLoggedInException"
           log="false">
  <redirect view-id="/register.xhtml">
    <message severity="warn">
      You must be a member to use this feature
    </message>
  </redirect>
</exception>
```

If the **`log`** attribute is not specified, then it defaults to **`true`** — that is, the exception will be logged. Alternatively, you can specify the **`log-level`** to control the level at which the exception will be logged:

```
<exception class="org.jboss.seam.security.NotLoggedInException"
           log-level="info">
  <redirect view-id="/register.xhtml">
    <message severity="warn">
      You must be a member to use this feature
    </message>
  </redirect>
</exception>
```

The acceptable values for **`log-level`** are: **`fatal, error, warn, info, debug`**, and **`trace`**. If the **`log-level`** is not specified, or if an invalid value is configured, **`log-level`** will default to **`error`**.

7.12.5. Some common exceptions

If you are using JPA:

```
<exception class="javax.persistence.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message>Not found</message>
  </redirect>
</exception>

<exception class="javax.persistence.OptimisticLockException">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message>
      Another user changed the same data, please try again
    </message>
  </redirect>
</exception>
```

If you are using the Seam Application Framework:

```
<exception class="org.jboss.seam.framework.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message>Not found</message>
  </redirect>
</exception>
```

If you are using Seam Security:

```
<exception class="org.jboss.seam.security.AuthorizationException">
  <redirect>
    <message>You do not have permission to do this</message>
  </redirect>
</exception>

<exception class="org.jboss.seam.security.NotLoggedInException">
  <redirect view-id="/login.xhtml">
    <message>Please log in first</message>
  </redirect>
</exception>
```

And, for JSF:

```
<exception class="javax.Faces.application.ViewExpiredException">
  <redirect view-id="/error.xhtml">
    <message>Your session has timed out, please try again</message>
  </redirect>
</exception>
```

A **`ViewExpiredException`** occurs when the user posts to a page after their session has expired. The **`conversation-required`** and **`no-conversation-view-id`** settings in the Seam page descriptor, discussed in Section 8.4, “Requiring a long-running conversation”, allow finer-grained control over session expiration while accessing a page used within a conversation.

CHAPTER 8. CONVERSATIONS AND WORKSPACE MANAGEMENT

Now we will take you through Seam's conversation model in finer detail.

The notion of a Seam *conversation* came about as a combination of three separate concepts:

- the concept of a *workspace*, and effective workspace management.
- the concept of an *application transaction* with optimistic semantics. Existing frameworks, based around a stateless architecture, were unable to provide effective management of extended persistence contexts.
- the concept of a workflow *task*.

By unifying these ideas and providing deep support in the framework, we have created a powerful construct that allows for richer and more efficient applications, using less verbose code.

8.1. SEAM'S CONVERSATION MODEL

All examples so far operate under a simple conversation model with the following rules:

- A conversation context is always active during the apply request values, process validation, update model values, invoke application and render response phases of the JSF request life cycle.
- At the end of the restore view phase of the JSF request life cycle, Seam attempts to restore any previous long-running conversation context. If none exists, Seam creates a new temporary conversation context.
- When a **`@Begin`** method is encountered, the temporary conversation context is promoted to a long-running conversation.
- When an **`@End`** method is encountered, any long-running conversation context is demoted to a temporary conversation.
- At the end of the render response phase of the JSF request life cycle, Seam either stores the contents of a long-running conversation context, or destroys the contents of a temporary conversation context.
- Any Faces request (a JSF postback) will propagate the conversation context. By default, non-Faces requests (GET requests, for example) do not propagate the conversation context.
- If the JSF request life cycle is foreshortened by a redirect, Seam transparently stores and restores the current conversation context, unless the conversation was already ended via **`@End(beforeRedirect=true)`**.

Seam transparently propagates the conversation context (including the temporary conversation context) across JSF postbacks and redirects. Without special additions, a *non-Faces request* (a GET request, for example) will not propagate the conversation context, and will be processed in a new temporary conversation. This is usually — but not always — the desired behavior.

To propagate a Seam conversation across a non-Faces request, the Seam *conversation ID* must be explicitly coded as a request parameter:

```
<a href="main.jsf?#{manager.conversationIdParameter}=#{conversation.id}">
  Continue
</a>
```

Or, for JSF:

```
<h:outputLink value="main.jsf">
  <f:param name="#{manager.conversationIdParameter}"
     value="#{conversation.id}"/>
  <h:outputText value="Continue"/>
</h:outputLink>
```

If you use the Seam tag library, this is equivalent:

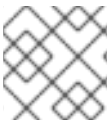
```
<h:outputLink value="main.jsf">
  <s:conversationId/>
  <h:outputText value="Continue"/>
</h:outputLink>
```

The code to disable propagation of the conversation context for a postback is similar:

```
<h:commandLink action="main" value="Exit">
  <f:param name="conversationPropagation" value="none"/>
</h:commandLink>
```

The equivalent for the Seam tag library is:

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="none"/>
</h:commandLink>
```



NOTE

Disabling conversation context propagation is *not* the same as ending the conversation.

The **`conversationPropagation`** request parameter or **`<s:conversationPropagation>`** tag can also be used to begin and end conversations, or to begin a nested conversation.

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="end"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Child">
  <s:conversationPropagation type="nested"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="begin"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="join"/>
</h:commandLink>
```

This conversation model makes it easy to build applications which behave correctly with respect to multi-window operation. For many applications, this is all that is required. Some complex applications have one or both of the following additional requirements:

- A conversation spans many smaller units of user interaction, which execute serially or even concurrently. The smaller *nested conversations* have their own isolated set of conversation state, and have access to the state of the outer conversation.
- The user can switch between many conversations within the same browser window. This feature is called *workspace management*.

8.2. NESTED CONVERSATIONS

A nested conversation is created by invoking a method marked **`@Begin(nested=true)`** within the scope of an existing conversation. A nested conversation has its own conversation context, but can read values from the outer conversation's context. The outer conversation's context is read-only within a nested conversation, but because objects are obtained by reference, changes to the objects themselves will be reflected in the outer context.

- Nesting a conversation initializes a context that is stacked on the context of the original, or outer, conversation. The outer conversation is considered the parent.
- Any values outjected or set directly into the nested conversation’s context do not affect the objects accessible in the parent conversation’s context.
- Injection, or a context look up from the conversation context, will first look up the value in the current conversation context. If no value is found, look up will continue down the conversation stack, if the conversation is nested. This behavior can be overridden.

When an **`@End`** is subsequently encountered, the nested conversation will be destroyed, and the outer conversation will resume, *popping* the conversation stack. Conversations may be nested to any arbitrary depth.

Certain user activities (workspace management, or the back button) can cause the outer conversation to be resumed before the inner conversation ends. In this case, it is possible to have multiple concurrent nested conversations belonging to the same outer conversation. If the outer conversation ends before a nested conversation ends, Seam destroys all nested conversation contexts along with the outer context.

The conversation at the bottom of the conversation stack is the root conversation. Destroying this conversation will always destroy all descendant conversations. You can achieve this declaratively by specifying **`@End(root=true)`**.

A conversation can be thought of as a *continuable state*. Nested conversations allow the application to capture a consistent continuable state at various points in a user interaction, thus ensuring truly correct behavior in the face of backbuttoning and workspace management.

As mentioned previously, if a component exists in a parent conversation of the current nested conversation, the nested conversation will use the same instance. Occasionally, it is useful to have a different instance in each nested conversation, so that the component instance that of the parent conversation is invisible to its child conversations. You can achieve this behavior by annotating the component **`@PerNestedConversation`**.

8.3. STARTING CONVERSATIONS WITH GET REQUESTS

JSF does not define any action listener triggered when a page is accessed via a non-Faces request (a HTTP GET request, for example). This can occur when a user bookmarks the page, or navigates to the page via an **<h:outputLink>**.

Sometimes we want a conversation to begin immediately the page is accessed. Since there is no JSF action method, we cannot annotate the action with **@Begin**.

Further problems arise when the page requires state to be fetched into a context variable. We have already seen two methods of solving this problem. If the state is held in a Seam component, we can fetch the state in a **@Create** method. If not, we can define a **@Factory** method for the context variable.

If neither option works for you, Seam lets you define a *page action* in the **pages.xml** file.

```
<pages>
  <page view-id="/messageList.jsp" action="#{messageManager.list}"/>
  ...
</pages>
```

This action method is called at the beginning of the render response phase — that is, any time the page is about to be rendered. If a page action returns a non-null outcome, Seam will process any appropriate JSF and Seam navigation rules. This can result in a completely different page rendering.

If beginning a conversation is *all* you want to do before rendering the page, you can use a built-in action method:

```
<pages>
  <page view-id="/messageList.jsp" action="#{conversation.begin}"/>
  ...
</pages>
```

You can also call this built-in action from a JSF control, and that **#{conversation.end}** similarly ends conversations.

The **<begin-conversation>** element can be used as follows for further control over joining existing conversations, or beginning a nested conversation, a pageflow, or an atomic conversation.

```
<pages>
  <page view-id="/messageList.jsp">
    <begin-conversation nested="true" pageflow="AddItem"/>
  <page>
  ...
</pages>
```

There is also an **<end-conversation>** element.

```
<pages>
  <page view-id="/home.jsp">
    <end-conversation/>
  <page>
  ...
</pages>
```

We now have five options to begin a conversation immediately the page is accessed:

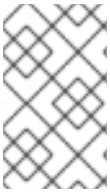
- Annotate the **`@Create`** method with **`@Begin`**
- Annotate the **`@Factory`** method with **`@Begin`**
- Annotate the Seam page action method with **`@Begin`**
- Use **`<begin-conversation>`** in **`pages.xml`**.
- Use **`#{conversation.begin}`** as the Seam page action method

8.4. REQUIRING A LONG-RUNNING CONVERSATION

Certain pages are only relevant in the context of a long-running conversation. One way to restrict access to such a page is to make the existence of a long-running conversation a prerequisite to the page being rendered.

Seam's page descriptor has a **`conversation-required`** attribute, which lets you indicate that the current conversation must be long-running (or nested) in order for a page to be rendered, like so:

```
<page view-id="/book.xhtml" conversation-required="true"/>
```



NOTE

At present, you cannot indicate which long-running conversation is required. However, you can build on the basic authorization by checking whether a specific value is also present in the conversation within a page action.

When Seam determines that the page has been requested while no long-running conversation is present, it performs the following actions:

- raises a contextual event called **`org.jboss.seam.noConversation`**
- registers a warning status message with the bundle key, **`org.jboss.seam.NoConversation`**
- redirects the user to an alternative page, if defined in the **`no-conversation-view-id`** attribute, like so:

  ```
  <pages no-conversation-view-id="/main.xhtml"/>
  ```

  This page will be used across the entire application; at present, multiple alternative pages cannot be defined.

8.5. USING `<S:LINK>` AND `<S:BUTTON>`

JSF command links always perform a form submission with JavaScript, which causes problems with the web browser's "open in new window" or "open in new tab" feature. If you require this functionality in plain JSF, you need to use an **`<h:outputLink>`**, but there are two major limitations to this method:

- JSF provides no way to attach an action listener to an **`<h:outputLink>`**, and

- JSF does not propagate the selected row of a **`DataModel`**, since there is no actual form submission.

To solve the first problem, Seam implements the notion of a *page action*, but this does not solve the second. It is possible to work around this by passing a request parameter and requerying for the selected object on the server-side, and in some cases (like the Seam blog example application), this is the best approach. Since it is RESTful and does not require server-side state, bookmarking is supported. In other cases, where bookmarking is unnecessary, **`@DataModel`** and **`@DataModelSelection`** are transparent and convenient.

To replace this missing functionality, and to simplify conversation propagation further, Seam provides the **`<s:link>`** JSF tag.

The link can specify only the JSF ID:

```
<s:link view="/login.xhtml" value="Login"/>
```

It can also specify an action method, in which case the action outcome determines the page that results:

```
<s:link action="#{login.logout}" value="Logout"/>
```

If both a JSF view ID and an action method are specified, the view will be used unless the action method returns a non-null outcome:

```
<s:link view="/loggedOut.xhtml"  action="#{login.logout}" value="Logout"/>
```

The link automatically propagates the selected row of a **`DataModel`** inside **`<h:dataTable>`**:

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}"
   value="#{hotel.name}"/>
```

You can leave the scope of an existing conversation:

```
<s:link view="/main.xhtml" propagation="none"/>
```

You can begin, end, or nest conversations:

```
<s:link action="#{issueEditor.viewComment}" propagation="nest"/>
```

If the link begins a conversation, you can specify the use of a particular pageflow:

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
   pageflow="EditDocument"/>
```

The **`taskInstance`** attribute is for use in jBPM task lists, as follows. See Section 1.8, “A complete application featuring Seam and jBPM: the DVD Store example” for an example.

```
<s:link action="#{documentApproval.approveOrReject}"
   taskInstance="#{task}"/>
```

Finally, use **`<s:button>`** if you want the "link" rendered as a button:

```
<s:button action="#{login.logout}" value="Logout"/>
```

8.6. SUCCESS MESSAGES

Messages are commonly displayed to the user to indicate the success or failure of an action. A JSF **`FacesMessage`** is convenient for this function. However, a successful action often requires a browser redirect. Since JSF does not propagate Faces messages across redirects, it is difficult to display success messages in plain JSF.

The built-in conversation-scoped Seam component named **`facesMessages`** solves this problem. (This requires the Seam redirect filter.)

```
@Name("editDocumentAction")
@Stateless
    public class EditDocumentBean implements EditDocument {
    @In EntityManager em;
    @In Document document;
    @In FacesMessages facesMessages;

    public String update() {
        em.merge(document);
        facesMessages.add("Document updated");
    }
}
```

When a message is added to **`facesMessages`**, it is used in the nextg render response phase for the current conversation. Since Seam preserves even temporary conversation contexts across redirects, this works even without a long-running conversation.

You can even include JSF EL expressions in a Faces message summary:

```
facesMessages.add("Document #{document.title} was updated");
```

Messages are displayed as usual:

```
<h:messages globalOnly="true"/>
```

8.7. NATURAL CONVERSATION IDS

When working with conversations that deal with persistent objects, there are several reasons to use the natural business key of the object instead of the standard, "surrogate" conversation ID.

Easy redirect to existing conversation

If the user requests the same operation twice, it can be useful to redirect to an existing conversation. Take the following situation, for example:

You are on Ebay, halfway through paying for an item you won as a Christmas present for your parents. You want to send it straight to them, but once you have entered your payment details, you cannot remember your parents' address. While you find the address, you accidentally reuse the same browser window, but now you need to return to payment for the item.

With a natural conversation, the user can easily rejoin the previous conversation and pick up where they left off. In this case, they can rejoin the payForItem conversation with the itemId as the conversation ID.

User-friendly URLs

A user-friendly URL is meaningful (refers to page contents plainly, without using ID numbers), and has a navigable heirarchy (that is, the user can navigate by editing the URL).

With a natural conversation, applications can generate long, complex URLs, but display simple, memorable URLs to users by using URLRewrite. In the case of our hotel booking example, **`http://seam-hotels/book.seam?hotel=BestWesternAntwerpen`** is rewritten as **`http://seam-hotels/book/BestWesternAntwerpen`** — much clearer. Note that URLRewrite relies upon parameters: **`hotel`** in the previous example must map to a unique parameter on the domain model.

8.8. CREATING A NATURAL CONVERSATION

Natural conversations are defined in **`pages.xml`**:

```
<conversation name="PlaceBid" parameter-name="auctionId"
              parameter-value="#{auction.auctionId}"/>
```

The first thing to note in the above definition is the conversation name, in this case **`PlaceBid`**. The conversation name identifies this particular named conversation uniquely, and is used by the **`page`** definition to identify a named conversation in which to participate.

The **`parameter-name`** attribute defines the request parameter that will hold the natural conversation ID, and replace the default conversation ID parameter. In this case, **`parameter-name`** is **`auctionId`**. This means that the URL of your page will contain **`auctionId=765432`** instead of a conversation parameter like **`cid=123`**.

The final attribute, **`parameter-value`**, defines an EL expression to evaluate the value of the natural business key to use as the conversation ID. In this example, the conversation ID will be the primary key value of the **`auction`** instance currently in scope.

Next, we define the pages participating in the named conversation. This is done by specifying the **`conversation`** attribute for a **`page`** definition:

```
<page view-id="/bid.xhtml" conversation="PlaceBid" login-required="true">
  <navigation from-action="#{bidAction.confirmBid}">
    <rule if-outcome="success">
      <redirect view-id="/auction.xhtml">
        <param name="id" value="#{bidAction.bid.auction.auctionId}"/>
      </redirect>
    </rule>
  </navigation>
</page>
```

8.9. REDIRECTING TO A NATURAL CONVERSATION

When initiating or redirecting to a natural conversation, there are several ways to specify the natural conversation name. We will start with the following page definition:

```
<page view-id="/auction.xhtml">
  <param name="id" value="#{auctionDetail.selectedAuctionId}"/>
  <navigation from-action="#{bidAction.placeBid}">
    <redirect view-id="/bid.xhtml"/>
  </navigation>
</page>
```

Here we see that invoking **#{bidAction.placeBid}** redirects us to **/bid.xhtml**, which is configured with the natural conversation ID **PlaceBid**. Our action method declaration looks like this:

```
@Begin(join = true)
public void placeBid()
```

When named conversations are specified in the **<page/>** element, redirection to the named conversation occurs as part of navigation rules following the invocation of the action method. This can cause problems when redirecting to an existing conversation, since redirection needs to occur before the action method is invoked. Therefore, the conversation name must be specified before the action is invoked. One method of doing this uses the **<s:conversationName>** tag:

```
<h:commandButton id="placeBidWithAmount" styleClass="placeBid"
   action="#{bidAction.placeBid}">
  <s:conversationName value="PlaceBid"/>
</h:commandButton>
```

You can also specify the **conversationName** attribute for either the **s:link** or **s:button**:

```
<s:link value="Place Bid" action="#{bidAction.placeBid}"
   conversationName="PlaceBid"/>
```

8.10. WORKSPACE MANAGEMENT

Workspace management is the ability to "switch" conversations in a single window. Seam workspace management is completely transparent at the Java level. To enable workspace management:

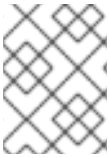
- Provide *description* text for each view ID (when using JSF or Seam navigation rules) or page node (when using jPDL pageflows). Workspace switchers display this description text to the user.
- Include one or more workspace switcher JSP or Facelets fragments in your page. Standard fragments support workspace management via a drop-down menu and a list of conversations, or "breadcrumbs".

8.10.1. Workspace management and JSF navigation

With JSF or Seam navigation rules in place, Seam switches to a conversation by restoring the current **view-id** for that conversation. The descriptive text for the workspace is defined in a file called **pages.xml**, which Seam expects to find in the **WEB-INF** directory alongside **faces-config.xml**:

```
<pages>
  <page view-id="/main.xhtml">
    <description>Search hotels: #{hotelBooking.searchString}</description>
  </page>
  <page view-id="/hotel.xhtml">
```

```
    <description>View hotel: #{hotel.name}</description>
  </page>
  <page view-id="/book.xhtml">
    <description>Book hotel: #{hotel.name}</description>
  </page>
  <page view-id="/confirm.xhtml">
    <description>Confirm: #{booking.description}</description>
  </page>
</pages>
```



NOTE

The Seam application will still work if this file is not present. However, workplace switching will not be available.

8.10.2. Workspace management and jPDL pageflow

When a jPDL pageflow definition is in place, Seam switches to a particular conversation by restoring the current jBPM process state. This is a more flexible model, since it allows the same **`view-id`** to have different descriptions depending on the current **<page>** node. The description text is defined by the **<page>** node:

```
<pageflow-definition name="shopping">
  <start-state name="start">
    <transition to="browse"/>
  </start-state>
  <page name="browse" view-id="/browse.xhtml">
    <description>DVD Search: #{search.searchPattern}</description>
    <transition to="browse"/>
    <transition name="checkout" to="checkout"/>
  </page>
  <page name="checkout" view-id="/checkout.xhtml">
    <description>Purchase: $#{cart.total}</description>
    <transition to="checkout"/>
    <transition name="complete" to="complete"/>
  </page>
  <page name="complete" view-id="/complete.xhtml">
    <end-conversation />
  </page>
</pageflow-definition>
```

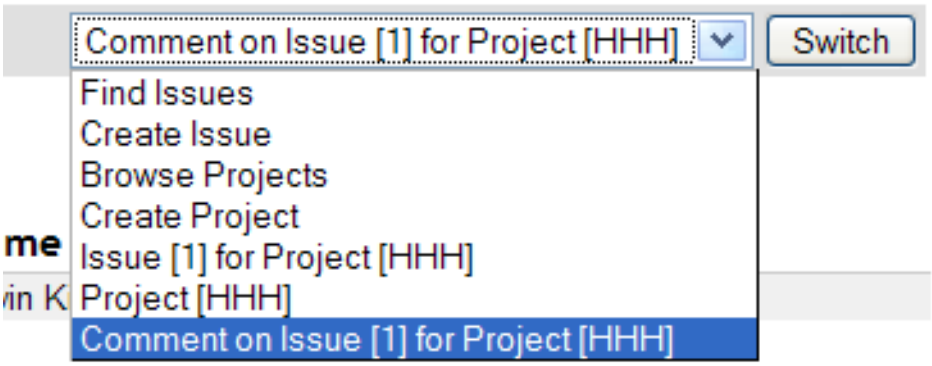
8.10.3. The conversation switcher

Including the following fragment in your JSP or Facelets page will include a drop-down menu that lets you switch to any current conversation, or any other page of the application:

```
<h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
  <f:selectItem itemLabel="Find Issues" itemValue="findIssue"/>
  <f:selectItem itemLabel="Create Issue" itemValue="editIssue"/>
  <f:selectItems value="#{switcher.selectItems}"/>
</h:selectOneMenu>
<h:commandButton action="#{switcher.select}" value="Switch"/>
```

This example includes a menu that contains an item for each conversation, plus two additional items that let the user begin an additional conversation.

Only conversations with a description (specified in **`pages.xml`**) will be included in the drop-down menu.



8.10.4. The conversation list

The conversation list is similar to the conversation switcher, except that it is displayed as a table:

```
<h:dataTable value="#{conversationList}" var="entry"
   rendered="#{not empty conversationList}">
  <h:column>
    <f:facet name="header">Workspace</f:facet>
    <h:commandLink action="#{entry.select}" value="#{entry.description}"/>
    <h:outputText value="[current]" rendered="#{entry.current}"/>
  </h:column>
  <h:column>
    <f:facet name="header">Activity</f:facet>
    <h:outputText value="#{entry.startDatetime}">
      <f:convertDateTime type="time" pattern="hh:mm a"/>
    </h:outputText>
    <h:outputText value=" - "/>
    <h:outputText value="#{entry.lastDatetime}">
      <f:convertDateTime type="time" pattern="hh:mm a"/>
    </h:outputText>
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <h:commandButton action="#{entry.select}" value="#{msg.Switch}"/>
    <h:commandButton action="#{entry.destroy}" value="#{msg.Destroy}"/>
  </h:column>
</h:dataTable>
```

This can be customized for your own applications.

Workspace	Workspace activity	Action
Comment on Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy

Only conversations with a description will be included in the list.

Note that the conversation list also lets the user destroy workspaces.

8.10.5. Breadcrumbs

Breadcrumbs are a list of links to conversations in the current conversation stack. They are useful for applications with a nested conversation model:

```
<ui:repeat value="#{conversationStack}" var="entry">
  <h:outputText value=" | "/>
  <h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</ui:repeat>
```

8.11. CONVERSATIONAL COMPONENTS AND JSF COMPONENT BINDINGS

Conversational components have one minor limitation: they cannot be used to hold bindings to JSF components. (Generally we recommend avoiding this feature of JSF unless absolutely necessary, since it creates a hard dependency from application logic to the view.) On a postback request, component bindings are updated during the Restore View phase, before the Seam conversation context has been restored.

You can work around this by storing component bindings with an event-scoped component, and injecting this into the requiring conversation-scoped component.

```
@Name("grid")
@Scope(ScopeType.EVENT)
public class Grid {
    private HtmlPanelGrid htmlPanelGrid; // getters and setters
    ...
}
```

```
@Name("gridEditor")
@Scope(ScopeType.CONVERSATION)
public class GridEditor {
    @In(required=false)
    private Grid grid;
    ...
}
```

You are also limited in that a conversation-scoped component cannot be injected into an event-scoped component with a JSF control bound to it. This includes Seam built-in components like **`facesMessages`**.

You can also access the JSF component tree with the implicit **`uiComponent`** handle. The following example accesses the **`getRowIndex()`** of the **`UIData`** component that backs the data table during iteration, and prints the current row number:

```
<h:dataTable id="lineItemTable" var="lineItem"
   value="#{orderHome.lineItems}">
  <h:column>
    Row: #{uiComponent[&#39;lineItemTable&#39;].rowIndex}
  </h:column>
  ...
</h:dataTable>
```

In this map, JSF UI components are available with their client identifier.

8.12. CONCURRENT CALLS TO CONVERSATIONAL COMPONENTS

Section 5.1.10, “Concurrency model” contains a general discussion of concurrent calls to Seam components. In this section, we discuss the most common situation in which you will encounter concurrency — when accessing conversational components from AJAX requests. We will also look at the options provided by an AJAX client library, and RichFaces, to control events originating from the client.

Conversational components do not allow true concurrent access, so Seam queues each request for serial processing. This allows each request to be executed in a deterministic fashion. However, there are some limitations to a simple queue. If a method, for whatever reason, takes a long time to complete, running it whenever the client generates a request can lead to Denial of Service attacks. AJAX is also often used to provide quick status updates to users, so continuing to run an action after a long time is not useful.

Therefore, when you work inside a long-running conversation, Seam queues the action even for a period of time (the concurrent request timeout). If Seam cannot process the event before timeout, it creates a temporary conversation and prints a message for the user, informing them of the timeout. It is therefore important not to flood the server with AJAX events.

We can set a sensible default for the concurrent request timeout (in milliseconds) in components.xml:

```
<core:manager concurrent-request-timeout="500" />
```

The concurrent request timeout can also be adjusted on a page-by-page basis:

```
<page view-id="/book.xhtml" conversation-required="true"
      login-required="true" concurrent-request-timeout="2000" />
```

So far we have discussed AJAX requests that appear serial to the user, where the client tells the server than an event has occurred, and then rerenders part of the page based on the result. This approach is sufficient when the AJAX request is lightweight (the methods called are simple, for example, calculating the sum of a column of numbers), but complex computations require a different approach.

A poll-based approach is where the client sends an AJAX request to the server, causing actions to begin immediate asynchronous execution on the server. The client then polls the server for updates while the actions are executed. This is a sensible approach when it is important that no action in a long-running action sequence times out.

8.12.1. How should we design our conversational AJAX application?

The first question is whether to use the simpler "serial" request method, or a polling approach.

If you want to use *serial* requests, you must estimate the time required for your request to complete. You may need to alter the concurrent request timeout for this page, as discussed in the previous section. A

queue on the server side is probably necessary, to prevent requests from flooding the server. If the event occurs often (for example, a keystroke, or onblur of input fields) and immediate client update is not a priority, set a request delay on the client side. Remember to factor the possibility of server-side queueing into your request delay.

Finally, the client library may provide an option to abort unfinished duplicate requests in favor of the most recent.

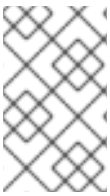
A polling approach requires less fine-tuning — simply mark your action method **@Asynchronous** and decide on a polling interval:

```
int total;

// This method is called when an event occurs on the client
// It takes a really long time to execute
@Asynchronous
public void calculateTotal() {
    total = someReallyComplicatedCalculation();
}

// This method is called as the result of the poll
// It's very quick to execute
public int getTotal() {
    return total;
}
```

8.12.2. Dealing with errors

However carefully your application is designed to queue concurrent requests to your conversational component, there is a risk that the server will overload. When overload occurs, not all requests will be processed before the **concurrent-request-timeout** period expires. In this case, Seam throws a **ConcurrentRequestTimeoutException**, which is handled in **pages.xml**. We recommend sending a HTTP 503 error:

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException"
           log-level="trace">
  <http-error error-code="503" />
</exception>
```



NOTE

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay.

Alternatively you could redirect to an error page:

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException"
           log-level="trace">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message>
      The server is too busy to process your request,
```

```
      please try again later
    </message>
  </redirect>
</exception>
```

ICEfaces, RichFaces AJAX and Seam Remoting can all handle HTTP error codes. Seam Remoting will pop up a dialog box showing the HTTP error. ICEfaces will indicate the error in its connection status component. RichFaces provides the most complete support for handling HTTP errors by providing a user definable callback. For example, to show the error message to the user:

```
<script type="text/javascript">
  A4J.AJAX.onError = function(req,status,message) {
    alert("An error occurred");
  };
</script>
```

If, rather than an error code, the server reports that the view has expired, perhaps because a session timed out, use a separate callback function in RichFaces:

```
<script type="text/javascript">
  A4J.AJAX.onExpired = function(loc,message) {
    alert("View expired");
  };
</script>
```

Alternatively, you can allow RichFaces to handle the error. In this case, the user will receive a prompt reading, "View state could not be restored — reload page?" This message can be globally customized by setting the following message key in an application resource bundle:

```
AJAX_VIEW_EXPIRED=View expired. Please reload the page.
```

8.12.3. RichFaces (Ajax4jsf)

RichFaces (Ajax4jsf) is the most common AJAX library used with Seam, and provides all of the controls discussed in the previous section.

`eventsQueue`

Provides a queue in which events are placed. All events are queued, and requests are sent to the server serially. This is useful if the request to the server can take some time to execute (for example, in heavy computation, retrieving information from a slow source) since it prevents server flooding.

`ignoreDupResponses`

Ignores the response produced by a request if a more recent "similar" request is already queued. **`ignoreDupResponses="true"`** does *not* cancel the processing of the request on the server side; it only prevents unnecessary updates on the client side.

With Seam conversations, this option should be used with care, since it allows multiple concurrent requests.

`requestDelay`

Defines the time in milliseconds that the request will remain on the queue. If, at this time, the request has not been processed, the request will either be sent (regardless of whether a response has been received), or discarded (if there is a more recent "similar" event queued).

With Seam conversations, this option should be used with care, as it allows multiple concurrent requests. The delay that you set (in combination with the concurrent request timeout) must be longer than the action will take to execute.

```
<a:poll reRender="total" interval="1000" />
```

Polls the server and rerenders an area, as required.

CHAPTER 9. PAGEFLOWS AND BUSINESS PROCESSES

JBoss jBPM is a business process management engine for any Java SE or EE environment. jBPM represents a business process or user interaction as a graph of nodes representing wait states, decisions, tasks, web pages, etc. The graph is defined with a simple, very readable XML dialect called jPDL, and can be edited and represented graphically using an Eclipse plug-in. jPDL is an extensible language, suitable for a range of problems, from defining web application pageflow, to managing traditional workflow, all the way up to orchestrating services in a SOA environment.

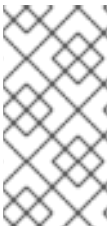
Seam applications use jBPM for two different problems: defining pageflow in complex user interactions, and defining the overarching business process.

For the former, a jPDL process definition defines the pageflow for a single conversation, whereas a Seam conversation is considered to be a relatively short-running interaction with a single user.

For the latter, the business process may span multiple conversations with multiple users. Its state is persistent in the jBPM database, so it is considered long-running. Coordinating the activities of multiple users is more complex than scripting interaction with a single user, so jBPM offers sophisticated facilities for managing tasks and multiple concurrent execution paths.

NOTE

Do not confuse pageflow with the overarching business process. They operate at different levels of granularity. Pageflows, conversations, and tasks are all single interactions with a single user. A business process spans many tasks. Furthermore, the two applications of jBPM are not dependent upon each other — you can use them together, independently, or not at all.



NOTE

It is not necessary to know jPDL to use Seam. If you prefer to define pageflow with JSF or Seam navigation rules, and your application is more data-driven than process-driven, jBPM is probably unnecessary. However, we find that thinking of user interaction in terms of a well-defined graphical representation helps us create more robust applications.

9.1. PAGEFLOW IN SEAM

There are two ways to define pageflow in Seam:

- Using JavaServer Faces (JSF) or Seam navigation rules — the *stateless navigation model*
- Using jPDL — the *stateful navigation model*

Simple applications will only require the stateless navigation model. Complex applications will use a combination of the two. Each model has its strengths and weaknesses, and should be implemented accordingly.

9.1.1. The two navigation models

The stateless model defines a mapping from a set of named, logical event outcomes directly to the resulting view page. The navigation rules ignore any state held by the application, other than the page from which the event originates. Therefore, action listener methods must sometimes make decisions about the pageflow, since only they have access to the current state of the application.

Here is an example pageflow definition using JSF navigation rules:

```
<navigation-rule>
  <from-view-id>/numberGuess.jsp</from-view-id>

  <navigation-case>
    <from-outcome>guess</from-outcome>
    <to-view-id>/numberGuess.jsp</to-view-id>
    <redirect/>
  </navigation-case>

  <navigation-case>
    <from-outcome>win</from-outcome>
    <to-view-id>/win.jsp</to-view-id>
    <redirect/>
  </navigation-case>

  <navigation-case>
    <from-outcome>lose</from-outcome>
    <to-view-id>/lose.jsp</to-view-id>
    <redirect/>
  </navigation-case>

</navigation-rule>
```

Here is the same example pageflow definition using Seam navigation rules:

```
<page view-id="/numberGuess.jsp">
  <navigation>

    <rule if-outcome="guess">
      <redirect view-id="/numberGuess.jsp"/>
    </rule>

    <rule if-outcome="win">
      <redirect view-id="/win.jsp"/>
    </rule>

    <rule if-outcome="lose">
      <redirect view-id="/lose.jsp"/>
    </rule>

  </navigation>
</page>
```

If you find navigation rules too verbose, you can return view IDs directly from your action listener methods:

```
public String guess() {
  if (guess==randomNumber) return "/win.jsp";
  if (++guessCount==maxGuesses) return "/lose.jsp";
  return null;
}
```

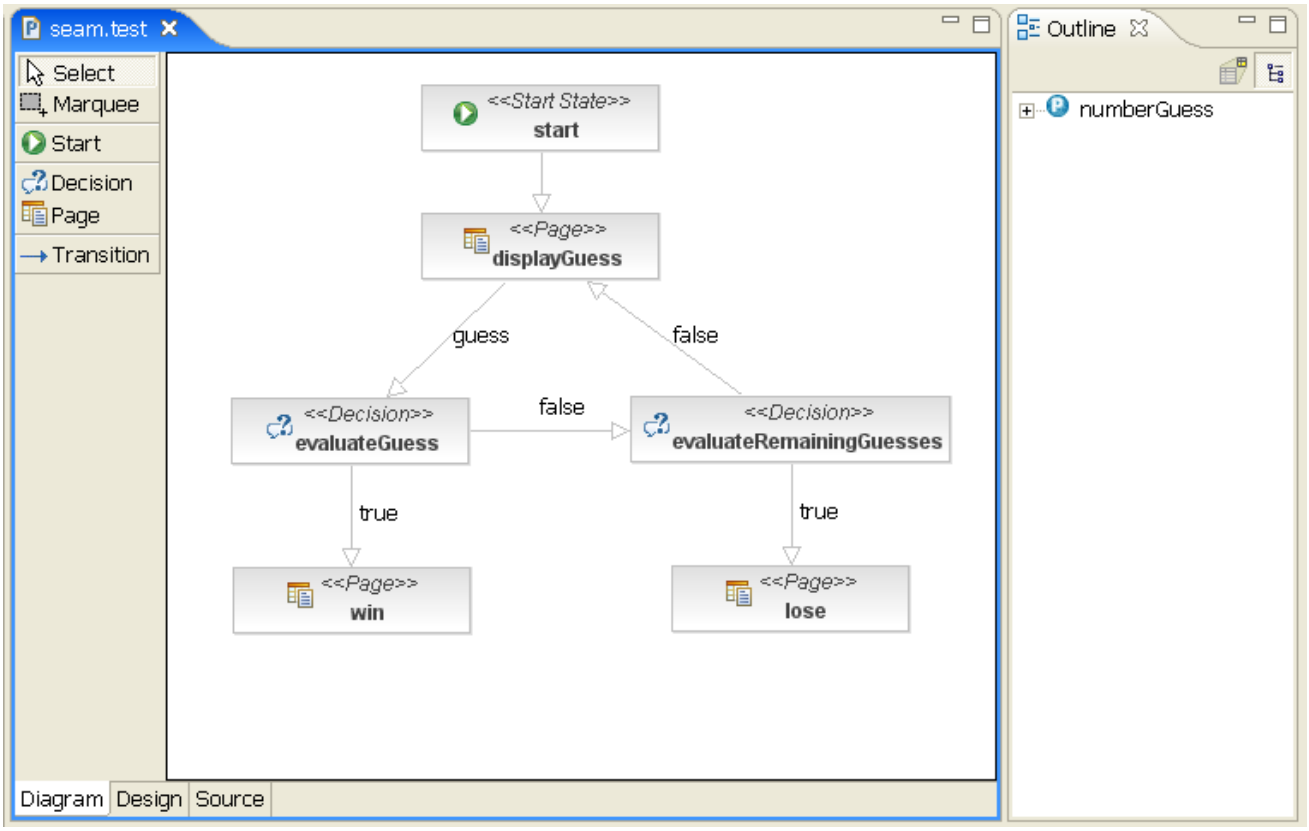
Note that this results in a redirect. You can also specify parameters to be used in the redirect:

```
public String search() {
  return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}
```

The stateful model defines a set of transitions between a set of named, logical application states. With this model, you can express the flow of any user interaction entirely in the jPDL pageflow definition, and write action listener methods that are completely unaware of the flow of the interaction.

Here is an example page flow definition using jPDL:

```
<pageflow-definition name="numberGuess">
  <start-page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition name="guess" to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
    </transition>
  </start-page>

  <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

  <decision name="evaluateRemainingGuesses"
            expression="#{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="win" view-id="/win.jsp">
    <redirect/>
    <end-conversation />
  </page>

  <page name="lose" view-id="/lose.jsp">
    <redirect/>
    <end-conversation />
  </page>

</pageflow-definition>
```



Here, we notice two things immediately:

- The JSF and Seam navigation rules are much simpler. (However, this obscures the fact that the underlying Java code is more complex.)
- The jPDL makes the user interaction immediately comprehensible, and removes the need to look at JSP or Java code.

In addition, the stateful model is more constrained. For each logical state (each step in the pageflow), there are a constrained set of possible transitions to other states. The stateless model is an *ad hoc* model, suitable to relatively unconstrained, freeform navigation where the user decides where he/she wants to go next, not the application.

The distinction between stateful and stateless navigation is similar to that between modal and modeless interaction. Seam applications are not usually modal in the simple sense of the word — we use conversations to avoid modal behavior. However, Seam applications can be, and often are, modal at the level of a particular conversation. Because user movements are not perfectly predictable, modal behavior is best avoided, but it has its place in the stateful model.

The biggest contrast between the two models is the back-button behavior.

9.1.2. Seam and the back button

With JSF or Seam navigation rules, the user can navigate freely with the back, forward and refresh buttons. The application is responsible for ensuring that conversational state remains internally consistent. Experience with web application frameworks and stateless component models has taught developers how difficult this is. It becomes far more straightforward in Seam, where it sits in a well-defined conversational model backed by stateful session beans. Usually, you need only combine a **`no-conversation-view-id`** with null checks at the beginning of action listener methods. Freeform navigation support is almost always desirable.

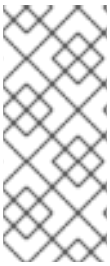
In this case, the **no-conversation-view-id** declaration goes in **pages.xml**. This tells Seam to redirect to a different page if a request originates from a page that was rendered during a conversation that no longer exists:

```
<page view-id="/checkout.xhtml" no-conversation-view-id="/main.xhtml"/>
```

On the other hand, in the stateful model, the back button is interpreted as an undefined transition back to a previous state. Since the stateful model enforces a defined set of transitions from the current state, the back button is, by default, not permitted in the stateful model. Seam transparently detects the use of the back button, and blocks any attempt to perform an action from a previous, "stale" page, redirecting the user to the "current" page (and displaying a Faces message). Although developers view this as a feature, it can be frustrating from the user's perspective. You can enable back button navigation from a particular page node by setting **back="enabled"**.

```
<page name="checkout" view-id="/checkout.xhtml" back="enabled">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

This allows navigation via the back button from the **checkout** state to any previous state.



NOTE

If a page is set to redirect after a transition, the back button cannot return a user to that page even when back is enabled on a page later in the flow. This is because Seam stores information about the pageflow in the page scope and the back button must result in a POST for that information to be restored (for example, through a Faces request). A redirect severs this link.

We must still define what happens if a request originates from a page rendered during a pageflow, and the conversation with the pageflow no longer exists. In this case, the **no-conversation-view-id** declaration goes into the pageflow definition:

```
<page name="checkout" view-id="/checkout.xhtml" back="enabled"
      no-conversation-view-id="/main.xhtml">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

In practice, both navigation models have their place, and you will quickly learn to recognize where one model is better-suited to a task.

9.2. USING JPDL PAGEFLOWS

9.2.1. Installing pageflows

We need to install the Seam jBPM-related components, and place the pageflow definitions (using the standard **.jpdl.xml** extension) inside a Seam archive (an archive containing a **seam.properties** file):

```
<bpm:jbpm />
```

We can also explicitly tell Seam where to find our pageflow definition. We specify this in **components.xml**:

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>pageflow.jpdl.xml</value>
  </bpm:pageflow-definitions>
</bpm:jbpm>
```

9.2.2. Starting pageflows

We "start" a jPDL-based pageflow by specifying the name of the process definition with a **@Begin**, **@BeginTask** or **@StartTask** annotation:

```
@Begin(pageflow="numberguess") public void begin() { ... }
```

Alternatively, we can start a pageflow using **pages.xml**:

```
<page>
  <begin-conversation pageflow="numberguess"/>
</page>
```

If we are beginning the pageflow during the **RENDER_RESPONSE** phase — during a **@Factory** or **@Create** method, for example — we consider ourselves already at the rendered page, and use a **<start-page>** node as the first node in the pageflow, as in the example above.

But if the pageflow is begun as the result of an action listener invocation, the outcome of the action listener determines the first page to be rendered. In this case, we use a **<start-state>** as the first node in the pageflow, and declare a transition for each possible outcome:

```
<pageflow-definition name="viewEditDocument">
  <start-state name="start">
    <transition name="documentFound" to="displayDocument"/>
    <transition name="documentNotFound" to="notFound"/>
  </start-state>

  <page name="displayDocument" view-id="/document.jsp">
    <transition name="edit" to="editDocument"/>
    <transition name="done" to="main"/>
  </page>

  ...

  <page name="notFound" view-id="/404.jsp">
    <end-conversation/>
  </page>

</pageflow-definition>
```

9.2.3. Page nodes and transitions

Each **<page>** node represents a state where the system is waiting for user input:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition name="guess" to="evaluateGuess">
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>
```

The **view-id** is the JSF view ID. The **<redirect/>** element has the same effect as **<redirect/>** in a JSF navigation rule — that is, a post-then-redirect behavior, to overcome problems with the browser's refresh button. (Note that Seam propagates conversation contexts across these browser redirects, so Seam does not require a Ruby on Rails-style *flash* construct.)

The transition name is the name of a JSF outcome triggered by clicking a command button or command link in **numberGuess.jsp**.

```
<h:commandButton type="submit" value="Guess" action="guess"/>
```

When clicking this button triggers the transition, jBPM activates the transition action by calling the **guess()** method of the **numberGuess** component. The syntax used for specifying actions in the jPDL is a familiar JSF EL expression, and the transition handler is a method of a Seam component in the current Seam contexts. Thus, we have the same event model for jBPM events as we have for JSF events. This is one of the guiding principles of Seam.

In the case of a null outcome (for example, a command button with no **action** defined), Seam signals the transition with no name (if one exists), or simply redisplay the page if all transitions are named. Therefore we could simplify this button and our pageflow like so:

```
<h:commandButton type="submit" value="Guess"/>
```

This would fire the following un-named transition:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition to="evaluateGuess">
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>
```

The button could also call an action method, in which case the action outcome determines the transition to be made:

```
<h:commandButton type="submit" value="Guess"
   action="#{numberGuess.guess}"/>
```

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <transition name="correctGuess" to="win"/>
  <transition name="incorrectGuess" to="evaluateGuess"/>
</page>
```

However, this style is considered inferior, since it shifts responsibility for flow control out of the pageflow definition and back into other components. It is much better to centralize this concern in the pageflow itself.

9.2.4. Controlling the flow

Usually, the more powerful features of jPDL are not required when defining pageflows. However, we do require the **<decision>** node:

```
<decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
  <transition name="true" to="win"/>
  <transition name="false" to="evaluateRemainingGuesses"/>
</decision>
```

A decision is made by evaluating a JSF EL expression within the Seam context.

9.2.5. Ending the flow

We end the conversation with **<end-conversation>** or **@End**. For the sake of readability, we encourage you to use both.

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-conversation/>
</page>
```

Optionally, we can end a task, or specify a jBPM **transition** name. In this case, Seam signals the end of the current task in the overarching business process.

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-task transition="success"/>
</page>
```

9.2.6. Pageflow composition

It is possible to compose pageflows so that one pageflow pauses while another pageflow executes. The **<process-state>** node pauses the outer pageflow, and begins execution of a named pageflow:

```
<process-state name="cheat">
  <sub-process name="cheat"/>
  <transition to="displayGuess"/>
</process-state>
```

The inner flow begins executing at a **<start-state>** node. When it reaches an **<end-state>** node, execution of the inner flow ends, and execution of the outer flow resumes with the transition defined by the **<process-state>** element.

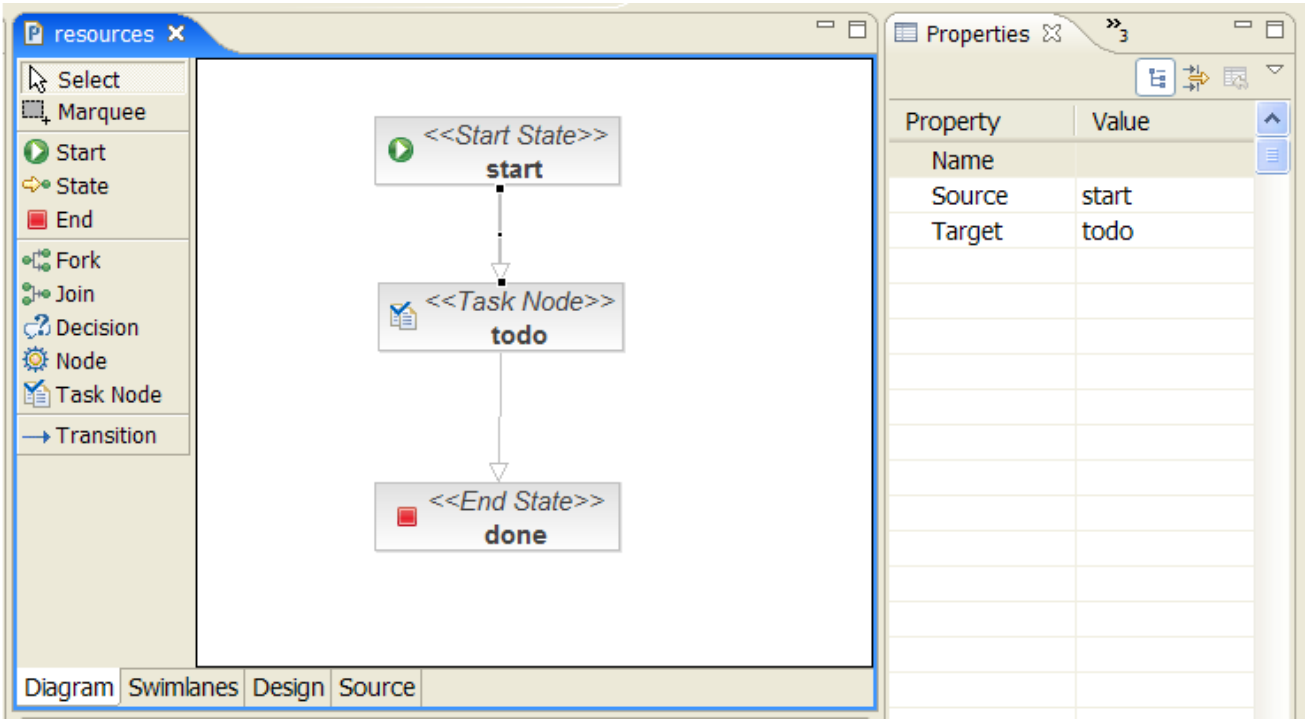
9.3. BUSINESS PROCESS MANAGEMENT IN SEAM

A business process is a set of tasks that must be performed by users or software systems according to well-defined rules regarding who can perform a certain task, and when that task should be performed.

Seam's jBPM integration makes it easy to let users view and manage their task lists. Seam also lets the application store state associated with the business process in the **BUSINESS_PROCESS** context, and makes that state persistent through jBPM variables.

A simple business process definition resembles a pageflow definition, except that instead of **<page>** nodes, we use **<task-node>** nodes. In a long-running business process, the wait state occurs where the system is waiting for some user to log in and perform a task.

```
<process-definition name="todo">
  <start-state name="start">
    <transition to="todo"/>
  </start-state>

  <task-node name="todo">
    <task name="todo" description="#{todoList.description}">
      <assignment actor-id="#{actor.id}"/>
    </task>
    <transition to="done"/>
  </task-node>

  <end-state name="done"/>

</process-definition>
```



jPDL business process definitions and jPDL pageflow definitions can be used in the same project. When this occurs, a single **<task>** in a business process corresponds to a whole pageflow **<pageflow-definition>**.

9.4. USING JPDL BUSINESS PROCESS DEFINITIONS

9.4.1. Installing process definitions

First, we must install jBPM and tell it where to find the business process definitions:

-

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>todo.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

Since jBPM processes persist across application restarts, when using Seam in a production environment, it is unnecessary to install the process definition each time the application starts. Therefore, the process must be deployed to jBPM outside Seam. It is only necessary to install process definitions from **`components.xml`** during application development.

9.4.2. Initializing actor IDs

We always need to know which user is currently logged in. jBPM recognizes users with their *actor ID* and *group actor ID*. We specify the current actor IDs with the built-in Seam component, **`actor`**:

```
@In Actor actor;

public String login() {
  ...
  actor.setId( user.getUserName() );
  actor.getGroupActorIds().addAll( user.getGroupNames() );
  ...
}
```

9.4.3. Initiating a business process

To initiate a business process instance, we use the **`@CreateProcess`** annotation:

```
@CreateProcess(definition="todo")
public void createTodo() { ... }
```

Alternatively we can initiate a business process using **`pages.xml`**:

```
<page>
  <create-process definition="todo" />
</page>
```

9.4.4. Task assignment

When a process reaches a task node, task instances are created. These must be assigned to users or user groups. We can either hard code our actor IDs, or delegate to a Seam component:

```
<task name="todo" description="#{todoList.description}">
  <assignment actor-id="#{actor.id}"/>
</task>
```

In this case, we have simply assigned the task to the current user. We can also assign tasks to a pool:

```
<task name="todo" description="#{todoList.description}">
  <assignment pooled-actors="employees"/>
</task>
```

9.4.5. Task lists

Several built-in Seam components make it easy to display task lists. The **pooledTaskInstanceList** is a list of pooled tasks that users may assign to themselves:

```
<h:dataTable value="#{pooledTaskInstanceList}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{pooledTask.assignToCurrentActor}"
            value="Assign" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

Note that instead of **<s:link>**, we can use a plain JSF **<h:commandLink>**:

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}">
  <f:param name="taskId" value="#{task.id}"/>
</h:commandLink>
```

The **pooledTask** component is a built-in component that simply assigns the task to the current user.

The **taskInstanceListForType** component includes tasks of a particular type that are assigned to the current user:

```
<h:dataTable value="#{taskInstanceListForType['todo']}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{todoList.start}"
            value="Start Work" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

9.4.6. Performing a task

To begin work on a task, we use either **@StartTask** or **@BeginTask** on the listener method:

```
@StartTask public String start() { ... }
```

Alternatively, we can begin work on a task with **pages.xml**:

```
<page>
  <start-task />
</page>
```

These annotations begin a special kind of conversation that is significant in terms of the overarching business process. Work done by this conversation has access to state held in the business process context.

If we end the conversation with **@EndTask**, Seam signals the completion of the task:

```
@EndTask(transition="completed")
public String completed() { ... }
```

Alternatively, we can use **pages.xml**:

```
<page>
  <end-task transition="completed" />
</page>
```

You can also use EL to specify the transition in **pages.xml**.

At this point, jBPM will continue to execute the business process definition. (In more complex processes, several tasks may need to be completed before process execution can resume.)

Please refer to the jBPM documentation for a more thorough overview of the sophisticated features that jBPM provides for managing complex business processes.

CHAPTER 10. SEAM AND OBJECT/RELATIONAL MAPPING

Seam provides extensive support for the two most popular persistence architectures for Java: Hibernate, and the Java Persistence API introduced with Enterprise JavaBeans 3.0 (EJB3). Seam's unique state-management architecture allows the most sophisticated ORM integration of any web application framework.

10.1. INTRODUCTION

Seam was created because of frustration with the statelessness typical of the previous generation of Java application architectures. Seam's state management architecture was originally designed to solve problems relating to persistence, particularly problems associated with *optimistic transaction processing*. Scalable online applications always use optimistic transactions. An atomic (database/JTA) level transaction should not span a user interaction unless the application is designed to support only a very small number of concurrent clients. But almost all work involves first displaying data to a user, and then updating that data. Hibernate was designed to support a persistence context that spanned an optimistic transaction.

Unfortunately, the "stateless" architectures that preceded Seam and EJB3 had no construct to represent an optimistic transaction. Instead, these architectures provided persistence contexts scoped to the atomic transaction. This resulted in many problems for users, and causes the number one user complaint: Hibernate's **`LazyInitializationException`**. A construct was required to represent an optimistic transaction in the application tier.

EJB3 recognizes this problem, and introduces the idea of a stateful component (a stateful session bean) with an *extended persistence context* scoped to the lifetime of the component. This is a partial solution to the problem (and is a useful construct in and of itself), but there are still two issues with this approach:

- The life cycle of the stateful session bean must be managed manually with code in the web tier.
- Propagation of the persistence context between stateful components in the same optimistic transaction is possible, but very complex.

Seam solves the first problem by providing conversations, and scoping stateful session bean components to the conversation. (Most conversations actually represent optimistic transactions in the data layer.) This is sufficient for many simple applications where persistence context propagation is not required, such as the Seam booking example application. For more complex applications, with many loosely-interacting components in each conversation, propagation of the persistence context across components becomes an important issue. Therefore, Seam extends the persistence context management model of EJB3, to provide conversation-scoped extended persistence contexts.

10.2. SEAM MANAGED TRANSACTIONS

EJB session beans feature declarative transaction management. The EJB container can start a transaction transparently when the bean is invoked, and end it when the invocation ends. If we write a session bean method that acts as a JSF action listener, all work associated with that action can be performed as one transaction, and committed or rolled back when the action is completely processed. This is a useful feature, and for some Seam applications, this is all that is required.

However, there is a problem with this approach: in a request from a single method call to a session bean, a Seam application may not perform all data access.

- when the request requires processing by several loosely-coupled components, with each component being called independently from the web layer. It is common to see multiple calls per request from the web layer to EJB components in Seam.

- when view rendering requires lazily-fetched associations.

The more transactions that exist per request, the more likely we are to encounter atomicity and isolation problems while our application processes many concurrent requests. All write operations should occur in the same transaction.

To work around this problem, Hibernate users developed the *open session in view* pattern. This is also important because some frameworks (Spring, for example) use transaction-scoped persistence contexts, which caused **`LazyInitializationException`**s when unfetched associations were accessed.

Open session in view is usually implemented as a single transaction that spans the entire request. The most serious problem with this implementation is that we cannot be certain that a transaction is successful until we commit it — but when the transaction commits, the view is fully rendered, and the rendered response may already be synchronized the client, so there is no way to notify the user that their transaction did not succeed.

Seam solves the problems with transaction isolation and association fetching, while working around the major flaw in *open session in view*, with two changes:

- Seam uses an extended persistence context that is scoped to the conversation instead of the transaction.
- Seam uses two transactions per request. The first spans from the beginning of the restore view phase until the end of the invoke application phase; the second spans the length of the render response phase. (In some applications, the first phase will begin later, at the beginning of the apply request values phase.)

The next section takes you through the setup of a conversation-scoped persistence context. Before this, we will enable Seam transaction management. You can use conversation-scoped persistence contexts without Seam transaction management, and Seam transaction management is useful even without Seam-managed persistence contexts, but they work most effectively together.

10.2.1. Disabling Seam-managed transactions

Seam transaction management is enabled by default for all JSF requests, but can be disabled in **`components.xml`**:

```
<core:init transaction-management-enabled="false"/>

<transaction:no-transaction />
```

10.2.2. Configuring a Seam transaction manager

Seam provides a transaction management abstraction for beginning, committing, rolling back, and synchronizing with transactions. By default, Seam uses a JTA transaction component to integrate with container-managed and programmatic EJB transactions. If you work in a Java EE 5 environment, install the EJB synchronization component in **`components.xml`**:

```
<transaction:ejb-transaction />
```

However, if you work in a non-EE 5 container, Seam attempts to auto-detect the correct transaction synchronization mechanism. If Seam is unable to detect the correct mechanism, you may need to configure one of the following:

- configure JPA RESOURCE_LOCAL managed transactions with the **`javax.persistence.EntityTransaction`** interface. **`EntityTransaction`** starts the transaction at the beginning of the apply request values phase.
- configure Hibernate managed transactions with the **`org.hibernate.Transaction`** interface. **`HibernateTransaction`** starts the transaction at the beginning of the apply request values phase.
- configure Spring managed transactions with the **`org.springframework.transaction.PlatformTransactionManager`** interface. The Spring **`PlatformTransactionManagement`** manager may begin the transaction at the beginning of the apply request values phase if the **`userConversationContext`** attribute is set.
- Explicitly disable Seam managed transactions

To configure JPA RESOURCE_LOCAL transaction management, add the following to your **`components.xml`**, where **`#{em}`** is the name of the **`persistence:managed-persistence-context`** component. If your managed persistence context is named **`entityManager`**, you may leave out the **`entity-manager`** attribute. (For further information, see Section 10.3, “Seam-managed persistence contexts”.)

```
<transaction:entity-transaction entity-manager="#{em}"/>
```

To configure Hibernate managed transactions, declare the following in your **`components.xml`**, where **`#{hibernateSession}`** is the name of the project's **`persistence:managed-hibernate-session`** component. If your managed hibernate session is named **`session`**, you can opt to leave out the **`session`** attribute. (For further information, see Section 10.3, “Seam-managed persistence contexts”.)

```
<transaction:hibernate-transaction session="#{hibernateSession}"/>
```

To explicitly disable Seam managed transactions, declare the following in your **`components.xml`**:

```
<transaction:no-transaction />
```

For information about configuring Spring-managed transactions see Section 26.5, “Using Spring PlatformTransactionManagement”.

10.2.3. Transaction synchronization

Transaction synchronization provides callbacks for transaction-related events such as **`beforeCompletion()`** and **`afterCompletion()`**. By default, Seam uses its own transaction synchronization component, which requires explicit use of the Seam transaction component when committing transactions so that synchronization callbacks are correctly executed. If you work in a Java EE 5 environment, declare **`<transaction:ejb-transaction/>`** in **`components.xml`** to ensure that Seam synchronization callbacks are called correctly if the container commits a transaction outside Seam.

10.3. SEAM-MANAGED PERSISTENCE CONTEXTS

If you use Seam outside a Java EE 5 environment, you cannot rely upon the container to manage the persistence context lifestyle. Even within EE 5 environments, propagating the persistence context between loosely-coupled components in a complex application can be difficult and error-prone.

In this case, you will need to use a *managed persistence context* (for JPA) or a *managed session* (for Hibernate) in your components. A Seam-managed persistence context is just a built-in Seam component that manages an instance of **`EntityManager`** or **`Session`** in the conversation context. You can inject it with **`@In`**.

Seam-managed persistence contexts are extremely efficient in a clustered environment. Seam can perform optimizations for container-managed persistence contexts that the EJB3 specification does not allow. Seam supports transparent failover of extended persistence contexts, without replicating any persistence context state between nodes. (We hope to add this support to the next revision of the EJB specification.)

10.3.1. Using a Seam-managed persistence context with JPA

Configuring a managed persistence context is easy. In **`components.xml`**, write:

```
<persistence:managed-persistence-context name="bookingDatabase"
   auto-create="true"
   persistence-unit-jndi-name="java:/EntityManagerFactories/bookingData"/>
```

This configuration creates a conversation-scoped Seam component named **`bookingDatabase`**, which manages the life cycle of **`EntityManager`** instances for the persistence unit (**`EntityManagerFactory`** instance) with JNDI name **`java:/EntityManagerFactories/bookingData`**.

You must bind the **`EntityManagerFactory`** into JNDI. In JBoss, you can do this by adding the following property setting to **`persistence.xml`**.

```
<property name="jboss.entity.manager.factory.jndi.name"
          value="java:/EntityManagerFactories/bookingData"/>
```

Now we can inject our **`EntityManager`** with:

```
@In EntityManager bookingDatabase;
```

If you use EJB3, and mark your class or method **`@TransactionAttribute(REQUIRES_NEW)`**, then the transaction and persistence context should not propagate to method calls on this object. However, since the Seam-managed persistence context propagates to any component within the conversation, it propagates to methods marked **`REQUIRES_NEW`**. Therefore, if you mark a method **`REQUIRES_NEW`**, you should access the entity manager with **`@PersistenceContext`**.

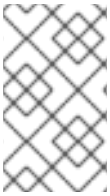
10.3.2. Using a Seam-managed Hibernate session

Seam-managed Hibernate sessions work in a similar fashion. In **`components.xml`**:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>

<persistence:managed-hibernate-session name="bookingDatabase"
            auto-create="true"
            session-factory-jndi-name="java:/bookingSessionFactory"/>
```

Here, **`java:/bookingSessionFactory`** is the name of the session factory specified in **`hibernate.cfg.xml`**.

```
<session-factory name="java:/bookingSessionFactory">
  <property name="transaction.flush_before_completion">true</property>
  <property name="connection.release_mode">after_statement</property>
  <property name="transaction.manager_lookup_class">
    org.hibernate.transaction.JBossTransactionManagerLookup
  </property>
  <property name="transaction.factory_class">
    org.hibernate.transaction.JTATransactionFactory
  </property>
  <property name="connection.datasource">
    java:/bookingDatasource
  </property>
  ...
</session-factory>
```



NOTE

Seam does not synchronize the session with the database, so always enable **`hibernate.transaction.flush_before_completion`** to ensure that the session is automatically synchronized before the JTA transaction commits.

We can now inject a managed Hibernate **`Session`** into our JavaBean components with the following code:

```
@In Session bookingDatabase;
```

10.3.3. Seam-managed persistence contexts and atomic conversations

Conversation-scoped persistence contexts let you program optimistic transactions spanning multiple server requests, without using **`merge()`**, reloading data at the beginning of each request, or wrestling with exceptions (**`LazyInitializationException`** or **`NonUniqueObjectException`**).

You can achieve transaction isolation and consistency by using optimistic locking. Both Hibernate and EJB3 make optimistic locking easy with the **`@Version`** annotation.

By default, the persistence context is synchronized with the database (flushed) at the end of each transaction. Sometimes this is desirable, but often we prefer all changes to be held in memory, and only written to the database when the conversation ends successfully. This allows for truly atomic conversations with EJB3 persistence. However, Hibernate provides this feature as a vendor extension to the **`FlushModeType`**s defined by the specification. We expect other vendors will soon provide a similar extension.

Seam lets you specify **`FlushModeType.MANUAL`** when beginning a conversation. Currently, this works only when Hibernate is the underlying persistence provider, but we plan to support other equivalent vendor extensions.

```
@In EntityManager em; //a Seam-managed persistence context
@Begin(flushMode=MANUAL)

public void beginClaimWizard() {
  claim = em.find(Claim.class, claimId);
}
```

Now, the **claim** object remains managed by the persistence context for the entire conversation. We can make changes to the claim:

```
public void addPartyToClaim() {
  Party party = ....;
  claim.addParty(party);
}
```

But these changes will not be flushed to the database until we explicitly force synchronization to occur:

```
@End public void commitClaim() {
  em.flush();
}
```

You can also set the **flushMode** to **MANUAL** from pages.xml, for example in a navigation rule:

```
<begin-conversation flush-mode="MANUAL" />
```

You can set any Seam-managed persistence context to use manual flush mode:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core">
  <core:manager conversation-timeout="120000"
        default-flush-mode="manual" />
</components>
```

10.4. USING THE JPA "DELEGATE"

The **EntityManager** interface lets you access a vendor-specific API with the **getDelegate()** method. We recommend using Hibernate as your vendor, and **org.hibernate.Session** as your delegate interface, but if you require a different JPA provider, see Section 28.2, “Using Alternate JPA Providers” for further information.

Regardless of your vendor, there are several approaches to using the delegate in your Seam components. One approach is:

```
@In EntityManager entityManager;
@Create public void init() {
  ((Session)entityManager.getDelegate() ).enableFilter("currentVersions");
}
```

If you, like most Java users, would rather avoid using typecasts, you can also access the delegate by adding the following line to **components.xml**:

```
<factory name="session" scope="STATELESS" auto-create="true"
         value="#{entityManager.delegate}"/>
```

The session can now be injected directly:

```
@In Session session;

@Create
```

```
public void init() {
  session.enableFilter("currentVersions");
}
```

10.5. USING EL IN EJB-QL/HQL

Seam proxies the **`EntityManager`** or **`Session`** object whenever you use a Seam-managed persistence context or inject a container-managed persistence context with **`@PersistenceContext`**. This lets you safely and efficiently use EL expressions in your query strings. For example, this:

```
User user = em.createQuery("from User where username=#{user.username}")
                          .getSingleResult();
```

is equivalent to:

```
User user = em.createQuery("from User where username=:username")
                          .setParameter("username", user.getUsername())
                          .getSingleResult();
```

WARNING

Do not use the format below, because it is vulnerable to SQL injection attacks, as well as being inefficient.

```
User user = em.createQuery("from User where username=" +
user.getUsername()).getSingleResult(); //BAD!
```

10.6. USING HIBERNATE FILTERS

Hibernate's most unique, useful feature is the *filter*. Filters provide a restricted view of the data in the database. You can find more information in the Hibernate documentation, but this section takes you through one easy, effective method of incorporating filters into Seam.

Seam-managed persistence contexts can have a list of filters defined, which will be enabled whenever an **`EntityManager`** or Hibernate **`Session`** is first created. (These can only be used when Hibernate is the underlying persistence provider.)

```
<persistence:filter name="regionFilter">
  <persistence:name>region</persistence:name>
  <persistence:parameters>
    <key>regionCode</key>
    <value>#{region.code}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:filter name="currentFilter">
  <persistence:name>current</persistence:name>
  <persistence:parameters>
```

```
    <key>date</key>
    <value>#{currentDate}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:managed-persistence-context name="personDatabase"
  persistence-unit-jndi-
name="java:/EntityManagerFactories/personDatabase">
  <persistence:filters>
    <value>#{regionFilter}</value>
    <value>#{currentFilter}</value>
  </persistence:filters>
</persistence:managed-persistence-context>
```

CHAPTER 11. JSF FORM VALIDATION IN SEAM

In plain JSF, validation is defined in the view:

```
<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <my:validateCountry/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <my:validateZip/>
    </h:inputText>
  </div>

  <h:commandButton/>
</h:form>
```

In practice, this approach usually violates DRY, since most "validation" actually enforces constraints that are part of the data model, and exist all the way down to the database schema definition. Seam provides support for model-based constraints defined with Hibernate Validator.

We will begin by defining our constraints, on our **`Location`** class:

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Length(max=30)
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @Length(max=6)
    @Pattern("^\d*$")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

In practice, it may be more elegant to use custom constraints rather than those built into Hibernate Validator:

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Country
```

```
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @ZipCode
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

Whichever method we choose, we no longer need specify the validation type to be used in the JSF page. Instead, we use **<s:validate>** to validate against the constraint defined on the model object.

```
<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <h:commandButton/>

</h:form>
```

NOTE

Specifying **@NotNull** on the model does *not* eliminate the need for **required="true"** to appear on the control. This is a limitation of the JSF validation architecture.

This approach defines constraints on the model, and presents constraint violations in the view.

The design is better, but not much less verbose than our initial design. Now, we will use **<s:validateAll>**:

```
<h:form>

  <h:messages/>

  <s:validateAll>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true"/>
  </div>
```

```
  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true"/>
  </div>

  <h:commandButton/>

  </s:validateAll>

</h:form>
```

This tag adds an **<s:validate>** to every input in the form. In a large form, this can save a lot of typing.

Next, we need to display feedback to the user when validation fails. Currently, all messages are displayed at the top of the form. To correlate the message with an input, you must define a label by using the standard **label** attribute on the input component.

```
<h:inputText value="#{location.zip}" required="true" label="Zip:">
  <s:validate/>
</h:inputText>
```

Inject this value into the message string with the placeholder {0} (the first and only parameter passed to a JSF message for a Hibernate Validator restriction). See the internationalization section for more information on where to define these messages.

NOTE

validator.length={0} length must be between {min} and {max}

We would prefer the message to be displayed beside the field with the error, highlight the field and label, and display an image next to the field. In plain JSF, only the first is possible. We also want to display a colored asterisk beside the label of each required form field.

This is a lot of functionality for each field. We do not want to specify highlighting and the layout of the image, message, and input field for every field on the form, so we specify the layout in a facelets template:

```
<ui:composition 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:s="http://jboss.com/products/seam/taglib">
  <div>

    <s:label styleClass="#{invalid?'error':''}">
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}">*</s:span>
    </s:label>

    <span class="#{invalid?'error':''}">
      <h:graphicImage value="/img/error.gif" rendered="#{invalid}"/>
      <s:validateAll>
        <ui:insert/>
      </s:validateAll>
    </span>
```

```
    <s:message styleClass="error"/>

  </div>

</ui:composition>
```

We can include this template for each of our form fields by using **<s:decorate>**:

```
<h:form>
  <h:messages globalOnly="true"/>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true"/>
    </s:decorate>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true"/>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

Finally, we can use RichFaces Ajax to display validation messages while the user navigates around the form:

```
<h:form>
  <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true">
            <a:support event="onblur" reRender="countryDecoration"
                       bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true">
            <a:support event="onblur" reRender="zipDecoration"
                       bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

Stylistically, it is better to define explicit IDs for important page controls, particularly if you want automated UI testing. If explicit IDs are not provided, JSF will generate its own — but they will not remain static if anything on the page is changed.

```
<h:form id="form">
  <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
    <ui:define name="label">Country:</ui:define>
      <h:inputText id="country" value="#{location.country}"
                   required="true">
        <a:support event="onblur" reRender="countryDecoration"
                   bypassUpdates="true"/>
      </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
      <ui:define name="label">Zip code:</ui:define>
      <h:inputText id="zip" value="#{location.zip}" required="true">
        <a:support event="onblur" reRender="zipDecoration"
                   bypassUpdates="true"/>
      </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

If you want to specify a different message to be displayed when validation fails, you can use the Seam message bundle with the Hibernate Validator:

```
public class Location {
    private String name;
    private String zip;

    // Getters and setters for name

    @NotNull
    @Length(max=6)
    @ZipCode(message="#{messages['location.zipCode.invalid']}")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

```
 location.zipCode.invalid = The zip code is not valid for #{location.name}
```

CHAPTER 12. GROOVY INTEGRATION

Seam has a great capability for Rapid Application Development (RAD). Seam allows you to utilize dynamic languages with your existing platform, while retaining compatibility with standard Java APIs. Static and dynamic languages are integrated, so there is no need for context-switching, and you can use the same annotations and APIs to write a dynamic Seam component as you would for a regular Seam component.

12.1. GROOVY INTRODUCTION

Groovy is an agile, Java-based dynamic language, with additional features inspired by Python, Ruby, and Smalltalk. Being Java-based, with Java objects and classes, Groovy is easy to learn and integrates seamlessly with existing Java libraries and frameworks.

12.2. WRITING SEAM APPLICATIONS IN GROOVY

Since Groovy objects are Java objects, any Seam component can be written and deployed with Groovy. You can also combine Groovy and Java classes in the same application.

12.2.1. Writing Groovy components

You will need to use Groovy 1.1 or higher to support annotations. The rest of this chapter shows how to use Groovy in a Seam application.

12.2.1.1. Entity

Example 12.1. Using Groovy in a Seam Application

```
@Entity
@Name("hotel")
class Hotel implements Serializable {
    @Id @GeneratedValue
    Long id

        @Length(max=50) @NotNull
        String name

        @Length(max=100) @NotNull
        String address

        @Length(max=40) @NotNull
        String city

        @Length(min=2, max=10) @NotNull
        String state

        @Length(min=4, max=6) @NotNull
        String zip

        @Length(min=2, max=40) @NotNull
        String country

        @Column(precision=6, scale=2)
        BigDecimal price
```

```
        @Override
        String toString(){
        return "Hotel(${name},${address},${city},${zip})"
    }
}
```

Since Groovy supports properties, there is no need to explicitly write verbose getters and setters. In the previous example, the hotel class can be accessed from Java as **`hotel.getCity()`** — the getters and setters are generated by the Groovy compiler. This makes the entity code very concise.

12.2.2. Seam component

You can write Seam components in Groovy exactly as you would in Java: annotations mark classes as Seam components.

Example 12.2. Writing Seam Components in Groovy

```
@Scope(ScopeType.SESSION)
@Name("bookingList")
class BookingListAction implements Serializable
{
    @In EntityManager em
    @In User user
    @DataModel List<Booking> bookings
    @DataModelSelection Booking booking
    @Logger Log log

    @Factory
    public void getBookings()
    {
        bookings = em.createQuery('''
        select b from Booking b
        where b.user.username = :username
        order by b.checkinDate''').
            setParameter("username", user.username).
            getResultList()
    }

    public void cancel()
    {
        log.info("Cancel booking: #{bookingList.booking.id}
                 for #{user.username}")
        Booking cancelled = em.find(Booking.class, booking.id)
        if (cancelled != null) em.remove( cancelled )
           getBookings()
           FacesMessages.instance().add("Booking cancelled for
confirmation
                                        number #
{bookingList.booking.id}",
                                        new Object[0])
    }
}
```

12.2.3. seam-gen

Seam-gen interacts transparently with Groovy. No additional infrastructure is required to include Groovy code in seam-gen-backed projects — when writing an entity, just place your **.groovy** files in **src/main**. When writing an action, place your **.groovy** files in **src/hot**.

12.3. DEPLOYMENT

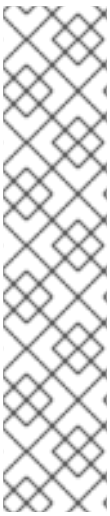
Deploying Groovy classes works like deploying Java classes. As with JavaBeans component classes, Seam can redeploy GroovyBeans component classes without restarting the application.

12.3.1. Deploying Groovy code

Groovy entities, session beans, and components all require compilation to deploy — use the **groovyc** ant task. Once compiled, a Groovy class is identical to a Java class, and the application server will treat them equally. This allows a seamless mix of Groovy and Java code.

12.3.2. Native .groovy file deployment at development time

Seam supports **.groovy** file hot deployment (deployment without compilation) in incremental hot deployment mode. This mode is development-only, and enables a fast edit/test cycle. Follow the configuration instructions at Section 3.8, "Seam and incremental hot deployment" to set up **.groovy** hot deployment. Deploy your Groovy code (**.groovy** files) into the **WEB-INF/dev** directory. The GroovyBean components will deploy incrementally, without needing to restart either application or application server.



NOTE

The native **.groovy** file deployment has the same limitations as the regular Seam hot deployment:

- components must be either JavaBeans or GroovyBeans — they cannot be EJB3 beans.
- entities cannot be hot deployed.
- hot-deployable components are not visible to any classes deployed outside **WEB-INF/dev**.
- Seam debug mode must be enabled.

12.3.3. seam-gen

Seam-gen transparently supports Groovy file deployment and compilation. This includes the native **.groovy** file hot deployment available during development. In WAR-type projects, Java and Groovy classes in **src/hot** are automatic candidates for incremental hot deployment. In production mode, Groovy files will be compiled prior to deployment.

There is a Booking demonstration, written completely in Groovy and supporting incremental hot deployment, in **examples/groovybooking**.

CHAPTER 13. THE SEAM APPLICATION FRAMEWORK

Seam makes it easy to create applications with annotated plain Java classes. We can make common programming tasks even easier by providing a set of pre-built components that are reusable with configuration or extension.

The Seam Application Framework can reduce the amount of code you need to write in a web application for basic database access with either Hibernate or JPA. The framework contains a handful of simple classes that are easy to understand and to extend where required.

13.1. INTRODUCTION

The components provided by the Seam Application Framework can be used in two separate approaches. The first approach is to install and configure an instance of the component in **`components.xml`**, as with other built-in Seam components. For example, the following fragment (from **`components.xml`**) installs a component that performs basic CRUD operations for a **`Person`** entity:

```
<framework:entity-home name="personHome" entity-class="eg.Person"
            entity-manager="#{personDatabase}">
<framework:id>#{param.personId}</framework:id>
</framework:entity-home>
```

If this approach seems too XML-heavy, you can approach this through extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
@In EntityManager personDatabase;
  public EntityManager getEntityManager() {
    return personDatabase;
  }
}
```

The major advantage to the second approach is that the framework classes were designed for extension and customization, so it is easy to add extra functionality or override the built-in functionality.

Another advantage is that you have the option of using EJB stateful session beans (or plain JavaBean components) as your classes:

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person>
                        implements LocalPersonHome { }
```

You can also make your classes stateless session beans. In this case you *must* use injection to provide the persistence context, even if it is called **`entityManager`**:

```
@Stateless
@Name("personHome")
public class PersonHome extends EntityHome<Person>
                        implements LocalPersonHome {
  @In EntityManager entityManager;
  public EntityManager getPersistenceContext() {
```

```
    entityManager;
  }
}
```

At present, the Seam Application Framework provides four main built-in components: **`EntityHome`** and **`HibernateEntityHome`** for CRUD, and **`EntityQuery`** and **`HibernateEntityQuery`** for queries.

The Home and Query components are written so that they can be session-, event- or conversation-scoped. The scope depends upon the state model you wish to use in your application.

The Seam Application Framework works only with Seam-managed persistence contexts. By default, components will expect a persistence context named **`entityManager`**.

13.2. HOME OBJECTS

A Home object provides persistence operations for a particular entity class. Suppose we have our **`Person`** class:

```
@Entity
public class Person {
  @Id private Long id;
  private String firstName;
  private String lastName;
  private Country nationality;
  //getters and setters...
}
```

We can define a **`personHome`** component either through configuration:

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

Or through extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {}
```

A Home object provides operations like **`persist()`**, **`remove()`**, **`update()`** and **`getInstance()`**. Before you can call **`remove()`** or **`update()`**, you must set the identifier of the object you are interested in, using the **`setId()`** method.

For example, we can use a Home directly from a JSF page:

```
<h1>Create Person</h1>
<h:form>
  <div>
    First name: <h:inputText value="#{personHome.instance.firstName}"/>
  </div>
  <div>
    Last name:  <h:inputText value="#{personHome.instance.lastName}"/>
  </div>
  <div>
    <h:commandButton value="Create Person"
```

```
        action="#{personHome.persist}"/>
  </div>
</h:form>
```

It is useful to be able to refer to **Person** as **person**, so we will add that line to **components.xml** (if we are using configuration):

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

Or, if we are using extension, we can add a **@Factory** method to **PersonHome**:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
}
```

This change simplifies our JSF page to the following:

```
<h1>Create Person</h1>
<h:form>
  <div>
    First name: <h:inputText value="#{person.firstName}"/>
  </div>
  <div>
    Last name: <h:inputText value="#{person.lastName}"/>
  </div>
  <div>
    <h:commandButton value="Create Person"
        action="#{personHome.persist}"/>
  </div>
</h:form>
```

This is all the code required to create new **Person** entries. If we want to be able to display, update, and delete pre-existing **Person** entries in the database, we need to be able to pass the entry identifier to the **PersonHome**. An excellent method is through page parameters:

```
<pages>
  <page view-id="/editPerson.jsp">
    <param name="personId" value="#{personHome.id}"/>
  </page>
</pages>
```

Now we can add the extra operations to our JSF page:

```
<h1>
  <h:outputText rendered="#{!personHome.managed}" value="Create Person"/>
  <h:outputText rendered="#{personHome.managed}" value="Edit Person"/>
</h1>
<h:form>
  <div>
```

```
    First name: <h:inputText value="#{person.firstName}"/>
  </div>
  <div>
    Last name: <h:inputText value="#{person.lastName}"/>
  </div>
  <div>
    <h:commandButton value="Create Person" action="#{personHome.persist}"
       rendered="#{!personHome.managed}"/>
    <h:commandButton value="Update Person" action="#{personHome.update}"
       rendered="#{personHome.managed}"/>
    <h:commandButton value="Delete Person" action="#{personHome.remove}"
       rendered="#{personHome.managed}"/>
  </div>
</h:form>
```

When we link to the page with no request parameters, the page will be displayed as a *Create Person* page. When we provide a value for the **`personId`** request parameter, it will be an *Edit Person* page.

If we need to create **`Person`** entries with their nationality initialized, we can do so easily. Via configuration:

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person"
           new-instance="#{newPerson}"/>
<component name="newPerson" class="eg.Person">
  <property name="nationality">#{country}</property>
</component>
```

Or via extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
  protected Person createInstance() {
    return new Person(country);
  }
}
```

The **`Country`** could be an object managed by another Home object, for example, **`CountryHome`**.

To add more sophisticated operations (association management, etc.), we simply add methods to **`PersonHome`**.

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
  protected Person createInstance() {
```

```
    return new Person(country);
  }
  public void migrate() {
    getInstance().setCountry(country);
    update();
  }
}
```

The Home object raises an **`org.jboss.seam.afterTransactionSuccess`** event when a transaction (a call to **`persist()`**, **`update()`** or **`remove()`**) succeeds. By observing this event, we can refresh our queries when the underlying entities change. If we only want to refresh certain queries when a particular entry is persisted, updated, or removed, we can observe the **`org.jboss.seam.afterTransactionSuccess.<name>`** (where **`<name>`** is the name of the entity).

The Home object automatically displays Faces messages when an operation succeeds. To customize these messages we can, again, use configuration:

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person"
           new-instance="#{newPerson}">
  <framework:created-message>
    New person #{person.firstName} #{person.lastName} created
  </framework:created-message>
  <framework:deleted-message>
    Person #{person.firstName} #{person.lastName} deleted
  </framework:deleted-message>
  <framework:updated-message>
    Person #{person.firstName} #{person.lastName} updated
  </framework:updated-message>
</framework:entity-home>
<component name="newPerson" class="eg.Person">
  <property name="nationality">#{country}</property>
</component>
```

Or extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
  protected Person createInstance() {
    return new Person(country);
  }
  protected String getCreatedMessage() {
    return createValueExpression("New person #{person.firstName}
                                          #{person.lastName}
created");
  }
  protected String getUpdatedMessage() {
    return createValueExpression("Person #{person.firstName}
                                        #{person.lastName} updated");
  }
```

```
  protected String getDeletedMessage() {
  return createValueExpression("Person #{person.firstName}
                                       #{person.lastName} deleted");
  }
}
```

The best way to specify messages is to put them in a resource bundle known to Seam — by default, the bundle named **messages**.

```
Person_created=New person #{person.firstName} #{person.lastName} created
Person_deleted=Person #{person.firstName} #{person.lastName} deleted
Person_updated=Person #{person.firstName} #{person.lastName} updated
```

This enables internationalization, and keeps your code and configuration clean of presentation concerns.

13.3. QUERY OBJECTS

If we need a list of all **Person** instances in the database, we can use a Query object, like the following.

```
<framework:entity-query name="people" ejbql="select p from Person p"/>
```

We can use it from a JSF page:

```
<h1>List of people</h1>
<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp"
            value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>
  </h:column>
</h:dataTable>
```

If you require pagination support:

```
<framework:entity-query name="people" ejbql="select p from Person p"
           order="lastName" max-results="20"/>
```

Use a page parameter to determine which page to display:

```
<pages>
  <page view-id="/searchPerson.jsp">
    <param name="firstResult" value="#{people.firstResult}"/>
  </page>
</pages>
```

The JSF code for pagination control is slightly verbose, but manageable:

```
<h1>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
  <h:column>

    <s:link view="/editPerson.jsp"
```

```
            value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>

  </h:column>

</h:dataTable>

<s:link view="/search.xhtml" rendered="#{people.previousExists}"
        value="First Page">
  <f:param name="firstResult" value="0"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.previousExists}"
        value="Previous Page">
  <f:param name="firstResult" value="#{people.previousFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}"
        value="Next Page">
  <f:param name="firstResult" value="#{people.nextFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}"
        value="Last Page">
  <f:param name="firstResult" value="#{people.lastFirstResult}"/>
</s:link>
```

Real search screens let the user enter optional search criteria to narrow the list of returned results. The Query object lets you specify optional restrictions to support this usecase:

```
<component name="examplePerson" class="Person"/>
<framework:entity-query name="people" ejbql="select p from Person p"
          order="lastName" max-results="20">
  <framework:restrictions>

    <value>
      lower(firstName) like lower(concat(#{examplePerson.firstName},'%&'))
    </value>

    <value>
      lower(lastName) like lower(concat(#{examplePerson.lastName},'%&'))
    </value>

  </framework:restrictions>
</framework:entity-query>
```

Notice the use of an "example" object.

```
<h1>Search for people</h1>
<h:form>

  <div>
    First name: <h:inputText value="#{examplePerson.firstName}"/>
  </div>
```

```
  <div>
    Last name: <h:inputText value="#{examplePerson.lastName}"/>
  </div>

  <div>
    <h:commandButton value="Search" action="/search.jsp"/>
  </div>

</h:form>

<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp"
       value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>
  </h:column>
</h:dataTable>
```

To refresh the query when the underlying entities change, we observe the **`org.jboss.seam.afterTransactionSuccess`** event:

```
<event type="org.jboss.seam.afterTransactionSuccess">
  <action execute="#{people.refresh}" />
</event>
```

Or, to refresh the query when the person entity is persisted, updated or removed through **`PersonHome`**:

```
<event type="org.jboss.seam.afterTransactionSuccess.Person">
  <action execute="#{people.refresh}" />
</event>
```

Unfortunately, Query objects do not work well with *join fetch* queries. We do not recommend using pagination with these queries. You will need to implement your own method of total result number calculation by overriding **`getCountEjbql()`**.

All of the examples in this section have shown re-use via configuration. It is equally possibly to re-use via extension:

13.4. CONTROLLER OBJECTS

The class **`Controller`** and its subclasses (**`EntityController`**, **`HibernateEntityController`** and **`BusinessProcessController`**) are an optional part of the Seam Application Framework. These classes provide a convenient method to access frequently-used built-in components and component methods. They save keystrokes, and provide an excellent foothold for new users to explore the rich functionality built into Seam.

For example, **`RegisterAction`** (from the Seam registration example) looks like this:

```
@Stateless
@Name("register")
public class RegisterAction extends EntityController implements Register {
  @In private User user;
  public String register() {
```

```
    List existing = createQuery("select u.username from
                                 User u where u.username=:username").
                                 setParameter("username",
                                 user.getUsername()).getResultList();
    if ( existing.size()==0 ) {
      persist(user);
      info("Registered new user #{user.username}");
      return "/registered.jspx";
    } else {
      addFacesMessage("User #{user.username} already exists");
      return null;
    }
  }
}
```

CHAPTER 14. SEAM AND JBOSS RULES

Seam makes it easy to call JBoss Rules (Drools) rulebases from Seam components or jBPM process definitions.

14.1. INSTALLING RULES

The first step is to make an instance of **`org.drools.RuleBase`** available in a Seam context variable. For testing purposes, Seam provides a built-in component that compiles a static set of rules from the classpath. You can install this component via **`components.xml`**:

```
<drools:rule-base name="policyPricingRules">
  <drools:rule-files>
    <value>policyPricingRules.drl</value>
  </drools:rule-files>
</drools:rule-base>
```

This component compiles rules from a set of DRL (**`.drl`**) or decision table (**`.xls`**) files and caches an instance of **`org.drools.RuleBase`** in the Seam **`APPLICATION`** context. Note that you will likely need to install multiple rule bases in a rule-driven application.

If you want to use a Drools DSL, you must also specify the DSL definition:

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
  <drools:rule-files>
    <value>policyPricingRules.drl</value>
  </drools:rule-files>
</drools:rule-base>
```

If you want to register a custom consequence exception handler through the RuleBaseConfiguration, you need to write the handler. This is demonstrated in the following example:

```
@Scope(ScopeType.APPLICATION)
@Startup
@Name("myConsequenceExceptionHandler")
public class MyConsequenceExceptionHandler
       implements ConsequenceExceptionHandler, Externalizable {
  public void readExternal(ObjectInput in) throws IOException,
                                          ClassNotFoundException { }

  public void writeExternal(ObjectOutput out) throws IOException { }

  public void handleException(Activation activation,
                              WorkingMemory workingMemory,
                              Exception exception) {
    throw new ConsequenceException( exception, activation.getRule() );
  }
}
```

and register it:

```
<drools:rule-base name="policyPricingRules"
                  dsl-file="policyPricing.dsl"
                  consequence-exception-handler=
```

```
                    "#{myConsequenceExceptionHandler}">
  <drools:rule-files>
    <value>policyPricingRules.drl</value>
  </drools:rule-files>
</drools:rule-base>
```

In most rules-driven applications, rules must be dynamically deployable. A Drools RuleAgent is useful to manage the RuleBase. The RuleAgent can connect to a Drools rule server (BRMS), or hot-deploy rules packages from a local file repository. The RulesAgent-managed RuleBase is also configurable via **`components.xml`**:

```
<drools:rule-agent name="insuranceRules"
        configurationFile="/WEB-INF/deployedrules.properties" />
```

The properties file contains properties specific to the RulesAgent. The following is an example configuration file from the Drools example distribution:

```
newInstance=true
url=http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/
           org.acme.insurance/fmeyer
localCacheDir=/Users/fernandomeyer/projects/jbossrules/drools-examples/
               drools-examples-brms/cache
poll=30
name=insuranceconfig
```

It is also possible to configure the options on the component directly, bypassing the configuration file.

```
<drools:rule-agent name="insuranceRules"
        url="http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/
                    package/org.acme.insurance/fmeyer"
        local-cache-dir="/Users/fernandomeyer/projects/jbossrules/
                         drools-examples/drools-examples-brms/cache"
        poll="30"
 configuration-name="insuranceconfig" />
```

Next, make an instance of **`org.drools.WorkingMemory`** available to each conversation. (Each **`WorkingMemory`** accumulates facts relating to the current conversation.)

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
        auto-create="true" rule-base="#{policyPricingRules}"/>
```

Notice that we referred the **`policyPricingWorkingMemory`** back to our rule base via the **`ruleBase`** configuration property.

We can also add means to be notified of rule engine events, including, for example, rules firing or objects being asserted by adding event listeners to WorkingMemory.

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
                               auto-create="true"
                               rule-base="#{policyPricingRules}">
  <drools:event-listeners>
    <value>org.drools.event.DebugWorkingMemoryEventListener</value>
```

```
    <value>org.drools.event.DebugAgendaEventListener</value>
  </drools:event-listeners>
</drools:managed-working-memory>
```

14.2. USING RULES FROM A SEAM COMPONENT

We can now inject our **`WorkingMemory`** into any Seam component, assert facts, and fire rules:

```
@In WorkingMemory policyPricingWorkingMemory;
@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException {
  policyPricingWorkingMemory.insert(policy);
  policyPricingWorkingMemory.insert(customer);
  policyPricingWorkingMemory.fireAllRules();
}
```

14.3. USING RULES FROM A JBPM PROCESS DEFINITION

A rule base can act as a jBPM action, decision, or assignment handler in either a pageflow or a business process definition.

```
<decision name="approval">
  <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
    <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
    <assertObjects>
      <element>#{customer}</element>
      <element>#{order}</element>
      <element>#{order.lineItems}</element>
    </assertObjects>
  </handler>

  <transition name="approved" to="ship">
    <action class="org.jboss.seam.drools.DroolsActionHandler">
      <workingMemoryName>shippingRulesWorkingMemory</workingMemoryName>
      <assertObjects>
        <element>#{customer}</element>
        <element>#{order}</element>
        <element>#{order.lineItems}</element>
      </assertObjects>
    </action>
  </transition>
  <transition name="rejected" to="cancelled"/>
</decision>
```

The **`<assertObjects>`** element specifies EL expressions that return an object or collection of objects to be asserted as facts into the **`WorkingMemory`**.

Using Drools for jBPM task assignments is also supported:

```
<task-node name="review">
  <task name="review" description="Review Order">
```

```
    <assignment handler="org.jboss.seam.drools.DroolsAssignmentHandler">
      <workingMemoryName>
        orderApprovalRulesWorkingMemory
      </workingMemoryName>
      <assertObjects>
        <element>#{actor}</element>
        <element>#{customer}</element>
        <element>#{order}</element>
        <element>#{order.lineItems}</element>
      </assertObjects>
    </assignment>
  </task>
  <transition name="rejected" to="cancelled"/>
  <transition name="approved" to="approved"/>
</task-node>
```

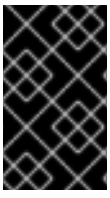
Certain objects are available as Drools globals — the jBPM **`Assignable`** is available as **`assignable`**, and the Seam **`Decision`** object is available as **`decision`**. Rules that handle decisions should call **`decision.setOutcome("result")`** to determine the decision result. Rules that perform assignments should set the actor ID with **`Assignable`**.

```
package org.jboss.seam.examples.shop
import org.jboss.seam.drools.Decision
global Decision decision
rule "Approve Order For Loyal Customer"
  when
    Customer( loyaltyStatus == "GOLD" )
    Order( totalAmount <= 10000 )
  then
    decision.setOutcome("approved");
end
```

```
package org.jboss.seam.examples.shop
import org.jbpm.taskmgmt.exe.Assignable
global Assignable assignable
rule "Assign Review For Small Order"
  when
    Order( totalAmount <= 100 )
  then
    assignable.setPooledActors( new String[] {"reviewers"} );
end
```

NOTE

More information about Drools is available at http://www.drools.org.



IMPORTANT

Seam comes packaged with enough Drools dependencies to implement some simple rules. To add extra capabilities, download the full Drools distribution and add extra dependencies as required.

CHAPTER 15. SECURITY

15.1. OVERVIEW

The Seam Security API provides a multitude of security-related features for your Seam-based application, including:

- Authentication — an extensible, Java Authentication and Authorization Service (JAAS) based authentication layer that allows users to authenticate against any security provider.
- Identity Management — an API for managing the users and roles of a Seam application at runtime.
- Authorization — an extremely comprehensive authorization framework, supporting user roles, persistent and rule-based permissions, and a pluggable permission-resolver that makes it easy to implement customized security logic.
- Permission Management — a set of built-in Seam components that make it easy to manage an application's security policy.
- CAPTCHA support — to assist in the prevention of automated software/scripts abusing your Seam-based site.

This chapter covers each of these features in detail.

15.2. DISABLING SECURITY

In some situations, you may need to disable Seam Security (during unit tests, for instance, or to use a different security approach, like native JAAS). To disable the security infrastructure, call the static method **`Identity.setSecurityEnabled(false)`**. However, when you want to configure the application, a more convenient alternative is to control the following settings in **`components.xml`**:

- Entity Security
- Hibernate Security Interceptor
- Seam Security Interceptor
- Page restrictions
- Servlet API security integration

This chapter documents the vast number of options available when establishing the user's identity (authentication) and establishing access constraints (authorization). We will begin with the foundation of the security model: authentication.

15.3. AUTHENTICATION

Seam Security provides Java Authentication and Authorization Service (JAAS) based authorization features, providing a robust and highly configurable API for handling user authentication. If your authentication needs are not this complex, Seam also offers a simplified authentication method.

15.3.1. Configuring an Authenticator component

NOTE

If you use Seam's Identity Management features, you can skip this section — it is not necessary to create an authenticator component.

Seam's simplified authentication method uses a built-in JAAS login module (**`SeamLoginModule`**) to delegate authentication to one of your own Seam components. (This module requires no additional configuration files, and comes pre-configured within Seam.) With this, you can write an authentication method with the entity classes provided by your own application, or authenticate through another third-party provider. Configuring this simplified authentication requires the **`identity`** component to be configured in **`components.xml`**

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/security
               http://jboss.com/products/seam/security-2.2.xsd">

<security:identity authenticate-method="#{authenticator.authenticate}"/>

</components>
```

`#{authenticator.authenticate}` is a method binding that indicates the **`authenticate`** method of the **`authenticator`** component will be used to authenticate the user.

15.3.2. Writing an authentication method

The **`authenticate-method`** property specified for **`identity`** in **`components.xml`** specifies the method used by **`SeamLoginModule`** to authenticate users. This method takes no parameters, and is expected to return a Boolean indicating authentication success or failure. Username and password are obtained from **`Credentials.getUsername()`** and **`Credentials.getPassword()`** respectively. (A reference to the **`credentials`** component can be obtained via **`Identity.instance().getCredentials()`**.) Any role that the user is a member of should be assigned with **`Identity.addRole()`**. The following is a complete example of an authentication method inside a POJO component:

```
@Name("authenticator")
public class Authenticator {
  @In EntityManager entityManager;
  @In Credentials credentials;
  @In Identity identity;

  public boolean authenticate() {
    try {
      User user = (User) entityManager.createQuery(
          "from User where username = :username and password = :password")
          .setParameter("username", credentials.getUsername())
          .setParameter("password", credentials.getPassword())
          .getSingleResult();
```

```
        if (user.getRoles() != null) {
          for (UserRole mr : user.getRoles())
            identity.addRole(mr.getName());
          }

          return true;
        } catch (NoResultException ex) {
          return false;
        }

    }

}
```

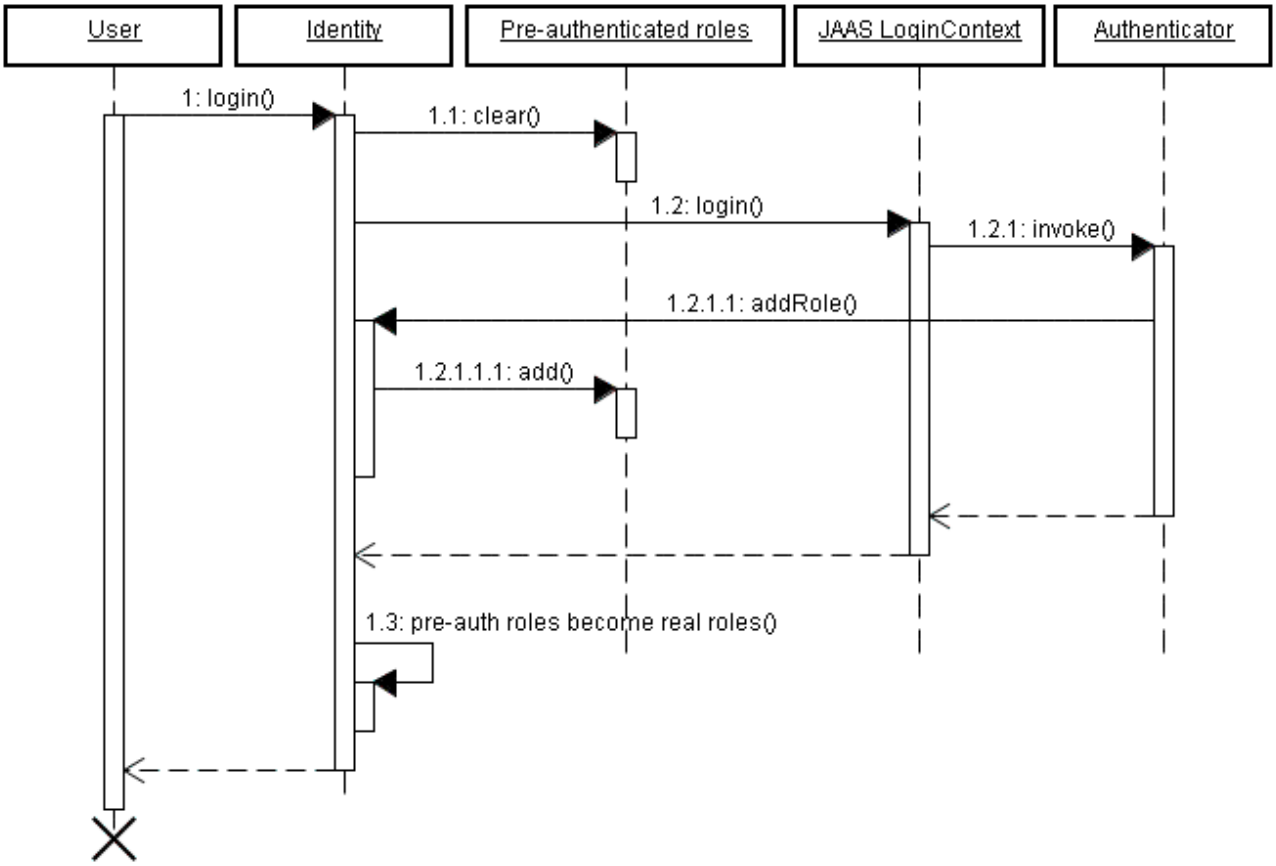
In the example, both **`User`** and **`UserRole`** are application-specific entity beans. The **`roles`** parameter is populated with roles that the user is a member of. This is added to the **`Set`** as literal string values — for example, "admin", "user", etc. If the user record is not found, and a **`NoResultException`** is thrown, the authentication method returns **`false`** to indicate authentication failure.

NOTE

It is important to keep authenticator methods minimal and free from any side-effects — they can be invoked multiple times during a single request, so any special code that should execute when authentication succeeds or fails should implement an event observer. See Section 15.10, “Security Events” later in this chapter for more information about events raised by Seam Security.

15.3.2.1. Identity.addRole()

The **`Identity.addRole()`** method's behavior depends upon current session authentication. If the session is not authenticated, **`addRole()`** should only be called during the authentication process. When called here, the role name is placed in a temporary list of pre-authenticated roles. Once authentication succeeds, the pre-authenticated roles then become "real" roles, and calling **`Identity.hasRole()`** for those roles returns **`true`**. The following sequence diagram represents the list of pre-authenticated roles as a first class object to clarify its position in the authentication process.



If the current session is already authenticated, then calling **`Identity.addRole()`** grants the specified role to the current user immediately.

15.3.2.2. Writing an event observer for security-related events

If, upon successful log in, some user statistics require updates, you can write an event observer for the **`org.jboss.seam.security.loginSuccessful`** event, like this:

```
@In UserStats userStats;

@Observer("org.jboss.seam.security.loginSuccessful")
public void updateUserStats() {
  userStats.setLastLoginDate(new Date());
  userStats.incrementLoginCount();
}
```

This observer method can be placed anywhere, even in the Authenticator component itself. More information about other security-related events appears later in the chapter.

15.3.3. Writing a login form

The **`credentials`** component provides both **`username`** and **`password`** properties, catering for the most common authentication scenario. These properties can be bound directly to the username and password fields on a login form. Once these properties are set, calling **`identity.login()`** authenticates the user with the credentials provided. An example of a simple login form is as follows:

```
<div>
  <h:outputLabel for="name" value="Username"/>
```

```
  <h:inputText id="name" value="#{credentials.username}"/>
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{credentials.password}"/>
</div>

<div>
  <h:commandButton value="Login" action="#{identity.login}"/>
</div>
```

Similarly, the user is logged out by calling **`#{identity.logout}`**. This action clears the security state of the currently authenticated user and invalidate the user's session.

15.3.4. Configuration Summary

There are three easy steps to configure authentication:

- Configure an authentication method in **`components.xml`**.
- Write an authentication method.
- Write a login form so that the user can authenticate.

15.3.5. Remember Me

Seam Security supports two different modes of the *Remember Me* functionality common to many web-based applications. The first mode allows the username to be stored in the user's browser as a cookie, and leaves the browser to remember the password. The second mode stores a unique token in a cookie, and lets a user authenticate automatically when they return to the site, without having to provide a password.

WARNING

Although it is convenient for users, automatic client authentication through a persistent cookie on the client machine is dangerous because the effects of any cross-site scripting (XSS) security hole are magnified. Without the authentication cookie, the only cookie an attacker can steal with XSS is the user's *current session cookie* — so an attack can only occur while a user has a session open. If a persistent *Remember Me* cookie is stolen, an attacker can log in without authentication at any time. If you wish to use automatic client authentication, it is vital to protect your website against XSS attacks.

Browser vendors introduced the *Remember Passwords* feature to combat this issue. Here, the browser remembers the username and password used to log in to a particular website and domain, and automatically fills in the login form when there is no session active. A log in keyboard shortcut on your website can make the log in process almost as convenient as the "Remember Me" cookie, and much safer. Some browsers (for example, Safari on OS X) store the login form data in the encrypted global operation system keychain. In a networked environment, the keychain can be transported with the user between laptop and desktop — cookies are not usually synchronized.

Although persistent *Remember Me* cookies with automatic authentication are widely used, they are bad security practice. Cookies that recall only the user's login name, and fill out the login form with that username as a convenience, are much more secure.

No special configuration is required to enable the *Remember Me* feature for the default (safe, username-only) mode. In your login form, simply bind the *Remember Me* checkbox to **`rememberMe.enabled`**, as seen in the following example:

```
<div>
  <h:outputLabel for="name" value="User name"/>
  <h:inputText id="name" value="#{credentials.username}"/>
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{credentials.password}"
redisplay="true"/>
</div>

<div class="loginRow">
  <h:outputLabel for="rememberMe" value="Remember me"/>
  <h:selectBooleanCheckbox id="rememberMe" value="#{rememberMe.enabled}"/>
</div>
```

15.3.5.1. Token-based *Remember Me* Authentication

To use the automatic, token-based mode of the *Remember Me* feature, you must first configure a token store. These authentication tokens are commonly stored within a database. Seam supports this method,

but you can also implement your own token store by using the **`org.jboss.seam.security.TokenStore`** interface. This section assumes that you will be using the provided **`JpaTokenStore`** implementation to store authentication tokens inside a database table.

First, create a new Entity to hold the tokens. The following is one possible structure:

```
@Entity
public class AuthenticationToken implements Serializable {
  private Integer tokenId;
  private String username;
  private String value;

  @Id @GeneratedValue
  public Integer getTokenId() {
    return tokenId;
  }

  public void setTokenId(Integer tokenId) {
     this.tokenId = tokenId;
  }

  @TokenUsername
  public String getUsername() {
     return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  @TokenValue
  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}
```

Several special annotations, **`@TokenUsername`** and **`@TokenValue`**, are used to configure the username and token properties of the entity. These annotations are required for the entity that holds the authentication tokens.

The next step is to configure **`JpaTokenStore`** to store and retrieve authentication tokens with this entity bean. Do this by specifying the **`token-class`** attribute in **`components.xml`**:

```
<security:jpa-token-store
     token-class="org.jboss.seam.example.seamspace.AuthenticationToken"/>
```

The final step is to configure the **`RememberMe`** component in **`components.xml`**. Its **`mode`** should be set to **`autoLogin`**:

```
<security:remember-me mode="autoLogin"/>
```

Users who check the *Remember Me* checkbox will now be authenticated automatically.

To ensure that users are automatically authenticated when returning to the site, the following section should be placed in **components.xml**:

```
<event type="org.jboss.seam.security.notLoggedIn">
  <action execute="#{redirect.captureCurrentView}"/>
  <action execute="#{identity.tryLogin()}"/>
</event>
<event type="org.jboss.seam.security.loginSuccessful">
  <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

15.3.6. Handling Security Exceptions

So that users do not receive a basic default error page when a security error occurs, you should edit **pages.xml** to redirect users to a more attractive page. The two main exceptions thrown by the security API are:

- **NotLoggedInException** — This exception is thrown when the user attempts to access a restricted action or page when they are not logged in.
- **AuthorizationException** — This exception is only thrown if the user is already logged in, and they have attempted to access a restricted action or page for which they do not have the necessary privileges.

In the case of a **NotLoggedInException**, we recommend the user be redirected to a login or registration page so that they can log in. For an **AuthorizationException**, it may be useful to redirect the user to an error page. Here's an example of a **pages.xml** file that redirects both of these security exceptions:

```
<pages>

  ...

  <exception class="org.jboss.seam.security.NotLoggedInException">
    <redirect view-id="/login.xhtml">
      <message>You must be logged in to perform this action</message>
    </redirect>
  </exception>

  <exception class="org.jboss.seam.security.AuthorizationException">
    <end-conversation/>
      <redirect view-id="/security_error.xhtml">
        <message>
          You do not have the necessary security privileges to perform
this action.
        </message>
      </redirect>
```

```
  </exception>

</pages>
```

Most web applications require more sophisticated handling of login redirection. Seam includes some special functionality, outlined in the following section.

15.3.7. Login Redirection

When an unauthenticated user tries to access a particular view or wildcarded view ID, you can have Seam redirect the user to a login screen as follows:

```
<pages login-view-id="/login.xhtml">

  <page view-id="/members/*" login-required="true"/>
...
</pages>
```

NOTE

This is more refined than the exception handler shown above, but should probably be used in conjunction with it.

After the user logs in, we want to automatically redirect them to the action that required log in. If you add the following event listeners to **components.xml**, attempts to access a restricted view while not logged in are remembered. Upon a successful log in, the user is redirected to the originally requested view, with any page parameters that existed in the original request.

```
<event type="org.jboss.seam.security.notLoggedIn">
  <action execute="#{redirect.captureCurrentView}"/>
</event>

<event type="org.jboss.seam.security.postAuthenticate">
  <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

NOTE

Login redirection is implemented as a conversation-scoped mechanism, so do not end the conversation in your **authenticate()** method.

15.3.8. HTTP Authentication

Although we do not recommend it unless absolutely necessary, Seam provides the means to authenticate with either HTTP Basic or HTTP Digest (RFC 2617) methods. For either form, you must first enable the **authentication-filter** component in **components.xml**:

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

To enable basic authentication, set **auth-type** to **basic**. For digest authentication, set it to **digest**. If you want to use digest authentication, you must also set the **key** and **realm**:

```
<web:authentication-filter url-pattern="*.seam" auth-type="digest"
     key="AA3JK34aSDlkj" realm="My App"/>
```

The **key** can be any String value. The **realm** is the name of the authentication realm that is presented to the user when they authenticate.

15.3.8.1. Writing a Digest Authenticator

If using digest authentication, your authenticator class should extend the abstract class **org.jboss.seam.security.digest.DigestAuthenticator**, and use the **validatePassword()** method to validate the user's plain text password against the digest request. Here is an example:

```
public boolean authenticate() {
  try {
    User user = (User) entityManager.createQuery(
         "from User where username = "username")
         .setParameter("username", identity.getUsername())
         .getSingleResult();

    return validatePassword(user.getPassword());
  } catch (NoResultException ex) {
    return false;
  }
}
```

15.3.9. Advanced Authentication Features

This section explores some of the advanced features provided by the security API for addressing more complex security requirements.

15.3.9.1. Using your container's JAAS configuration

If you prefer not to use the simplified JAAS configuration provided by the Seam Security API, you can use the default system JAAS configuration by adding a **jaas-config-name** property to **components.xml**. For example, if you use JBoss Enterprise Application Platform and want to use the **other** policy (which uses the **UsersRolesLoginModule** login module provided by JBoss Enterprise Application Platform), then the entry in **components.xml** would look like this:

```
<security:identity jaas-config-name="other"/>
```

Keep in mind that doing this does not mean that your user will be authenticated in your Seam application container — it instructs Seam Security to authenticate itself with the configured JAAS security policy.

15.4. IDENTITY MANAGEMENT

Identity Management provides a standard API for managing a Seam application's users and roles, regardless of the identity store (database, LDAP, etc.) used in back-end operations. The **identityManager** component is at the core of the Identity Management API, and provides all methods for creating, modifying, and deleting users, granting and revoking roles, changing passwords, enabling and disabling user accounts, authenticating users, and listing users and roles.

Before use, the **identityManager** must be configured with at least one **IdentityStore**. These components interact with the back-end security provider.

15.4.1. Configuring IdentityManager

The **identityManager** component allows you to configure separate identity stores for authentication and authorization. This means that users can be authenticated against one identity store (for example, an LDAP directory), but have their roles loaded from another identity store (such as a relational database).

Seam provides two **IdentityStore** implementations out of the box. The default, **JpaIdentityStore**, uses a relational database to store user and role information. The other implementation is **LdapIdentityStore**, which uses an LDAP directory to store users and roles.

The **identityManager** component has two configurable properties: **identityStore** and **roleIndentityStore**. The value for these properties must be an EL expression that refers to a Seam component with the **IdentityStore** interface. If left unconfigured, the default (**JpaIdentityStore**) will be used. If only the **identityStore** property is configured, the same value will be used for **roleIdentityStore**. For example, the following entry in **components.xml** will configure **identityManager** to use an **LdapIdentityStore** for both user-related and role-related operations:

```
<security:identity-manager identity-store="#{ldapIdentityStore}"/>
```

The following example configures **identityManager** to use an **LdapIdentityStore** for user-related operations, and **JpaIdentityStore** for role-related operations:

```
<security:identity-manager identity-store="#{ldapIdentityStore}"
          role-identity-store="#{jpaIdentityStore}"/>
```

The following sections explain each identity storage method in greater detail.

15.4.2. JpaIdentityStore

This method stores users and roles in a relational database. It is designed to allow flexible database design and table structure. A set of special annotations lets entity beans store user and role records.

15.4.2.1. Configuring JpaIdentityStore

Both **user-class** and **role-class** must be configured before **JpaIdentityStore** can be used. These properties refer to the entity classes used to store user and role records, respectively. The following example shows the **components.xml** file from the SeamSpace example:

```
<security:jpa-identity-store
          user-class="org.jboss.seam.example.seamspace.MemberAccount"
          role-class="org.jboss.seam.example.seamspace.MemberRole"/>
```

15.4.2.2. Configuring the Entities

The following table describes the special annotations used to configure entity beans for user and role storage.

Table 15.1. User Entity Annotations

Annotation	Status	Description
`@UserPrincipal`	Required	This annotation marks the field or method containing the user's username.

`@UserPassword`	Required	This annotation marks the field or method containing the user's password. It allows a **hash** algorithm to be specified for password hashing. Possible values for **hash** are **md5**, **sha** and **none**. For example:
		`@UserPassword(hash = "md5")`
		`public String getPasswordHash() {`
		`  return passwordHash;`
		`}`
		It is possible to extend the **PasswordHash** component to implement other hashing algorithms, if required.

`@UserFirstName`	Optional	This annotation marks the field or method containing the user's first name.
`@UserLastName`	Optional	This annotation marks the field or method containing the user's last name.
`@UserEnabled`	Optional	This annotation marks the field or method containing the enabled user status. This should be a Boolean property. If not present, all user accounts are assumed to be enabled.
`@UserRoles`	Required	This annotation marks the field or method containing the roles of the user. This property will be described in more detail in a later section.

Table 15.2. Role Entity Annotations

Annotation	Status	Description

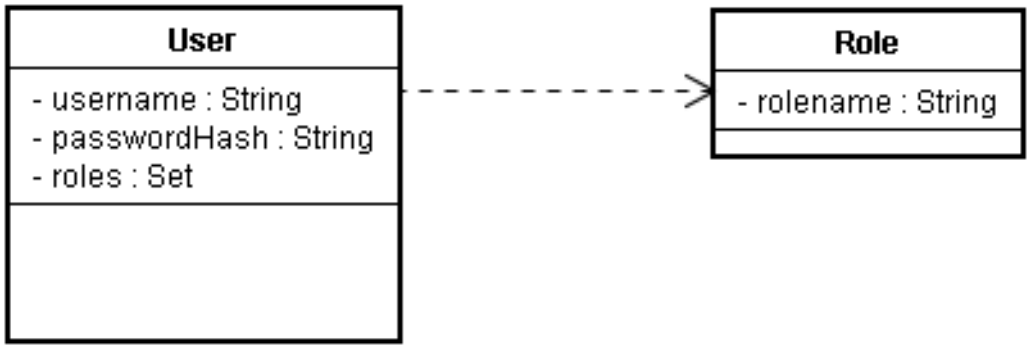
Annotation	Status	Description
`@RoleName`	Required	This annotation marks the field or method containing the name of the role.
`@RoleGroups`	Optional	This annotation marks the field or method containing the group memberships of the role.
`@RoleConditional`	Optional	This annotation marks the field or method that indicates whether a role is conditional. Conditional roles are explained later in this chapter.

15.4.2.3. Entity Bean Examples

As mentioned previously, **`JpaIdentityStore`** is designed to be as flexible as possible when it comes to the database schema design of your user and role tables. This section looks at a number of possible database schemas that can be used to store user and role records.

15.4.2.3.1. Minimal schema example

Here, a simple user and role table are linked via a many-to-many relationship using a cross-reference table named **`UserRoles`**.



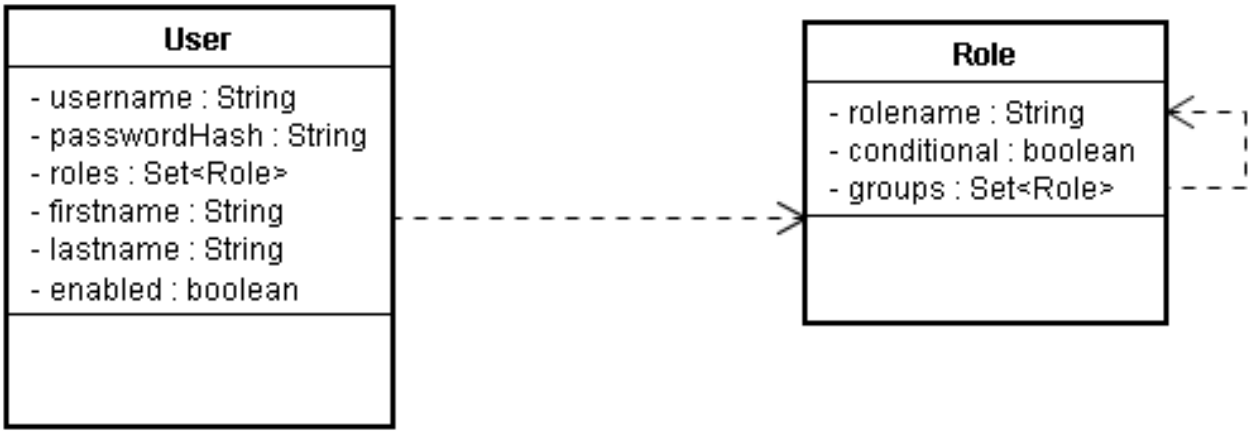

```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) {
    this.passwordHash = passwordHash;
```

```
  }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
       joinColumns = @JoinColumn(name = "UserId"),
       inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }
}
```

15.4.2.3.2. Complex Schema Example

This example builds on the previous minimal example by including all optional fields, and allowing group memberships for roles.




```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;
  private String firstname;
  private String lastname;
  private boolean enabled;

  @Id @GeneratedValue
```

```
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) {
    this.passwordHash = passwordHash;
  }

  @UserFirstName
  public String getFirstname() { return firstname; }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }

  @UserLastName
  public String getLastname() { return lastname; }
  public void setLastname(String lastname) { this.lastname = lastname; }

  @UserEnabled
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) { this.enabled = enabled; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;
  private boolean conditional;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }

  @RoleConditional
  public boolean isConditional() { return conditional; }
  public void setConditional(boolean conditional) {
    this.conditional = conditional;
  }
```

```
    @RoleGroups
    @ManyToMany(targetEntity = Role.class)
    @JoinTable(name = "RoleGroups",
               joinColumns = @JoinColumn(name = "RoleId"),
               inverseJoinColumns = @JoinColumn(name = "GroupId"))
    public Set<Role> getGroups() { return groups; }
    public void setGroups(Set<Role> groups) { this.groups = groups; }

}
```

15.4.2.4. JpaIdentityStore Events

When using **`JpaIdentityStore`** with **`IdentityManager`**, several events are raised when certain **`IdentityManager`** methods are invoked.

15.4.2.4.1. JpaIdentityStore.EVENT_PRE_PERSIST_USER

This event is raised in response to calling **`IdentityManager.createUser()`**. Just before the user entity is persisted to the database, this event is raised to pass the entity instance as an event parameter. The entity will be an instance of the **`user-class`** configured for **`JpaIdentityStore`**.

An observer can be useful, here, for setting entity field values that are not part of standard **`createUser()`** functionality.

15.4.2.4.2. JpaIdentityStore.EVENT_USER_CREATED

This event is also raised in response to calling **`IdentityManager.createUser()`**. However, it is raised after the user entity has already been persisted to the database. Like the **`EVENT_PRE_PERSIST_USER`** event, it also passes the entity instance as an event parameter. It may be useful to observe this event if you need to persist other entities that reference the user entity, such as contact detail records or other user-specific data.

15.4.2.4.3. JpaIdentityStore.EVENT_USER_AUTHENTICATED

This event is raised when calling **`IdentityManager.authenticate()`**. It passes the user entity instance as the event parameter, and is useful for reading additional properties from the user entity being authenticated.

15.4.3. LdapIdentityStore

This identity storage method is designed to work with user records stored in an LDAP directory. It is highly configurable, and allows very flexible directory storage of both users and roles. The following sections describe the configuration options for this identity store, and provide some configuration examples.

15.4.3.1. Configuring LdapIdentityStore

The following table describes the properties that can be configured in **`components.xml`** for **`LdapIdentityStore`**.

Table 15.3. LdapIdentityStore Configuration Properties

Property	Default Value	Description
server-address	**localhost**	The address of the LDAP server.
server-port	**389**	The port number that the LDAP server listens on.
user-context-DN	**ou=Person,dc=acme,dc=com**	The Distinguished Name (DN) of the context containing user records.
user-DN-prefix	**uid=**	This value is prefixed to the front of the username to locate the user's record.
user-DN-suffix	**,ou=Person,dc=acme,dc=com**	This value is appended to the end of the username to locate the user's record.
role-context-DN	**ou=Role,dc=acme,dc=com**	The DN of the context containing role records.
role-DN-prefix	**cn=**	This value is prefixed to the front of the role name to form the DN that locates the role record.
role-DN-suffix	**,ou=Roles,dc=acme,dc=com**	This value is appended to the role name to form the DN that locates the role record.
bind-DN	**cn=Manager,dc=acme,dc=com**	This is the context used to bind to the LDAP server.
bind-credentials	**secret**	These are the credentials (the password) used to bind to the LDAP server.

Property	Default Value	Description
`user-role-attribute`	`roles`	The attribute name of the user record containing the list of roles that the user is a member of.
`role-attribute-is-DN`	`true`	This Boolean property indicates whether the role attribute of the user record is itself a distinguished name.
`user-name-attribute`	`uid`	Indicates the user record attribute containing the username.
`user-password-attribute`	`userPassword`	Indicates the user record attribute containing the user's password.
`first-name-attribute`	`null`	Indicates the user record attribute containing the user's first name.
`last-name-attribute`	`sn`	Indicates the user record attribute containing the user's last name.
`full-name-attribute`	`cn`	Indicates the user record attribute containing the user's full (common) name.
`enabled-attribute`	`null`	Indicates the user record attribute that determines whether the user is enabled.
`role-name-attribute`	`cn`	Indicates the role record attribute containing the name of the role.

Property	Default Value	Description
object-class-attribute	**objectClass**	Indicates the attribute that determines the class of an object in the directory.
role-object-classes	**organizationalRole**	An array of the object classes that new role records should be created as.
user-object-classes	**person,uidObject**	An array of the object classes that new user records should be created as.

15.4.3.2. LdapIdentityStore Configuration Example

The following configuration example shows how **LdapIdentityStore** can be configured for an LDAP directory running on fictional host **directory.mycompany.com**. The users are stored within this directory under the **ou=Person,dc=mycompany,dc=com** context, and are identified by the **uid** attribute (which corresponds to their username). Roles are stored in their own context, **ou=Roles,dc=mycompany,dc=com**, and are referenced from the user's entry via the **roles** attribute. Role entries are identified by their common name (the **cn** attribute), which corresponds to the role name. In this example, users can be disabled by setting the value of their **enabled** attribute to **false**.

```
<security:ldap-identity-store
  server-address="directory.mycompany.com"
  bind-DN="cn=Manager,dc=mycompany,dc=com"
  bind-credentials="secret"
  user-DN-prefix="uid="
  user-DN-suffix=",ou=Person,dc=mycompany,dc=com"
  role-DN-prefix="cn="
  role-DN-suffix=",ou=Roles,dc=mycompany,dc=com"
  user-context-DN="ou=Person,dc=mycompany,dc=com"
  role-context-DN="ou=Roles,dc=mycompany,dc=com"
  user-role-attribute="roles"
  role-name-attribute="cn"
  user-object-classes="person,uidObject"
  enabled-attribute="enabled"
/>
```

15.4.4. Writing your own IdentityStore

Writing your own identity store implementation allows you to authenticate and perform identity management operations against security providers that are not supported out of the box by Seam. You only need a single class that implements the **org.jboss.seam.security.management.IdentityStore** interface to achieve this.

Refer to the JavaDoc about **`IdentityStore`** for a description of the methods that must be implemented.

15.4.5. Authentication with Identity Management

If you use Identity Management features in your Seam application, then you do not need to provide an authenticator component (see previous Authentication section) to enable authentication. Simply omit the **`authenticator-method`** from the **`identity`** configuration in **`components.xml`**, and the **`SeamLoginModule`** will use **`IdentityManager`** to authenticate your application's users without any special configuration.

15.4.6. Using IdentityManager

Access the **`IdentityManager`** either by injecting it into your Seam component, like so:

```
@In IdentityManager identityManager;
```

or, through its static **`instance()`** method:

```
IdentityManager identityManager = IdentityManager.instance();
```

The following table describes **`IdentityManager`**'s API methods:

Table 15.4. Identity Management API

Method	Returns	Description
`createUser(String name, String password)`	**`boolean`**	Creates a new user account, with the specified name and password. Returns **`true`** if successful; otherwise, returns **`false`**.
`deleteUser(String name)`	**`boolean`**	Deletes the user account with the specified name. Returns **`true`** if successful; otherwise, returns **`false`**.
`createRole(String role)`	**`boolean`**	Creates a new role, with the specified name. Returns **`true`** if successful; otherwise, returns **`false`**.
`deleteRole(String name)`	**`boolean`**	Deletes the role with the specified name. Returns **`true`** if successful; otherwise, returns **`false`**.
`enableUser(String name)`	**`boolean`**	Enables the user account with the specified name. Accounts that are not enabled cannot authenticate. Returns **`true`** if successful; otherwise, returns **`false`**.
`disableUser(String name)`	**`boolean`**	Disables the user account with the specified name. Returns **`true`** if successful; otherwise, returns **`false`**.

Method	Returns	Description
`changePassword(String name, String password)`	`boolean`	Changes the password for the user account with the specified name. Returns `true` if successful; otherwise, returns `false`.
`isUserEnabled(String name)`	`boolean`	Returns `true` if the specified user account is enabled; otherwise, returns `false`.
`grantRole(String name, String role)`	`boolean`	Grants the specified role to the specified user or role. The role must already exist for it to be granted. Returns `true` if the role is successfully granted, or `false` if the user has already been granted the role.
`revokeRole(String name, String role)`	`boolean`	Revokes the specified role from the specified user or role. Returns `true` if the specified user is a member of the role and it is successfully revoked, or `false` if the user is not a member of the role.
`userExists(String name)`	`boolean`	Returns `true` if the specified user exists, or `false` if it does not.
`listUsers()`	`List`	Returns a list of all user names, sorted in alpha-numeric order.
`listUsers(String filter)`	`List`	Returns a list of all user names filtered by the specified filter parameter, sorted in alpha-numeric order.
`listRoles()`	`List`	Returns a list of all role names.
`getGrantedRoles(String name)`	`List`	Returns a list of all roles explicitly granted to the specified user name.
`getImpliedRoles(String name)`	`List`	Returns a list of all roles implicitly granted to the specified user name. Implicitly granted roles include those that are granted to the roles that the user is a member of, rather than granted directly to the user. For example, if the `admin` role is a member of the `user` role, and a user is a member of the `admin` role, then the implied roles for the user are both the `admin`, and `user` roles.

Method	Returns	Description
`authenticate(String name, String password)`	`boolean`	Authenticates the specified username and password using the configured Identity Store. Returns `true` if successful or `false` if authentication failed. Successful authentication implies nothing beyond the return value of the method. It does not change the state of the `Identity` component - to perform a proper Seam log in the `Identity.login()` must be used instead.
`addRoleToGroup(String role, String group)`	`boolean`	Adds the specified role as a member of the specified group. Returns true if the operation is successful.
`removeRoleFromGroup(String role, String group)`	`boolean`	Removes the specified role from the specified group. Returns true if the operation is successful.
`listRoles()`	`List`	Lists the names of all roles.

A calling user must have appropriate authorization to invoke methods on the Identity Management API. The following table describes the permission requirements for each of the methods in `IdentityManager`. The permission targets listed below are literal String values.

Table 15.5. Identity Management Security Permissions

Method	Permission Target	Permission Action
`createUser()`	`seam.user`	`create`
`deleteUser()`	`seam.user`	`delete`
`createRole()`	`seam.role`	`create`
`deleteRole()`	`seam.role`	`delete`
`enableUser()`	`seam.user`	`update`
`disableUser()`	`seam.user`	`update`
`changePassword()`	`seam.user`	`update`
`isUserEnabled()`	`seam.user`	`read`
`grantRole()`	`seam.user`	`update`

Method	Permission Target	Permission Action
`revokeRole()`	`seam.user`	`update`
`userExists()`	`seam.user`	`read`
`listUsers()`	`seam.user`	`read`
`listRoles()`	`seam.role`	`read`
`addRoleToGroup()`	`seam.role`	`update`
`removeRoleFromGroup()`	`seam.role`	`update`

The following code listing provides an example set of security rules that grants all **`admin`** role members access to all Identity Management-related methods:

```
rule ManageUsers
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.user", granted == false)
  Role(name == "admin")
then
  check.grant();
end

rule ManageRoles
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.role", granted == false)
  Role(name == "admin")
then
  check.grant();
end
```

15.5. ERROR MESSAGES

The security API produces a number of default Faces messages for various security-related events. The following table lists the message keys to specify in a **`message.properties`** resource file if you want to override the messages. To suppress a message, add the key (with an empty value) to the resource file.

Table 15.6. Security Message Keys

Message Key	Description

Message Key	Description
`org.jboss.seam.loginSuccessful`	This message is produced when a user successfully logs in via the security API.
`org.jboss.seam.loginFailed`	This message is produced when the log in process fails, either because the user provided an incorrect username or password, or because authentication failed in some other way.
`org.jboss.seam.NotLoggedIn`	This message is produced when a user attempts to perform an action or access a page that requires a security check, and the user is not currently authenticated.
`org.jboss.seam.AlreadyLoggedIn`	This message is produced when a user that is already authenticated attempts to log in again.

15.6. AUTHORIZATION

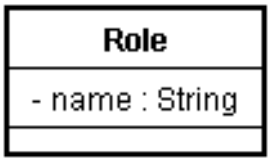
This section describes the range of authorization mechanisms provided by the Seam Security API for securing access to components, component methods, and pages. If you wish to use any of the advanced features (for example, rule-based permissions), you may need to configure your `components.xml` file — see the Configuration section previous.

15.6.1. Core concepts

Seam Security operates on the principle that users are granted roles or permissions, or both, which allow them to perform operations that are not permissible for users without the required security privileges. Each authorization mechanism provided by the Seam Security API is built upon this core concept of roles and permissions, with an extensible framework to provide multiple ways to secure application resources.

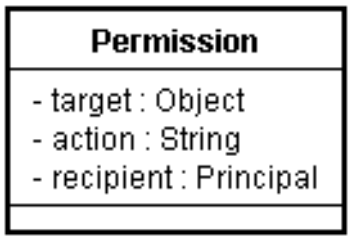
15.6.1.1. What is a role?

A role is a type of user that may have been granted certain privileges for performing one or more specific actions within an application. They are simple constructs, consisting of a name (such as "admin", "user", "customer", etc.) applied to users, or other roles. They are used to create logical user groups so that specific application privileges can be easily assigned.



15.6.1.2. What is a permission?

A permission is a privilege (sometimes once-off) for performing a single, specific action. You can build an application that operates solely on permissions, but roles are more convenient when granting privileges to groups. Permissions are slightly more complex in structure than roles, consisting of three "aspects"; a target, an action, and a recipient. The target of a permission is the object (or an arbitrary name or class) for which a particular action is allowed to be performed by a specific recipient (or user). For example, the user "Bob" may have permission to delete customer objects. In this case, the permission target may be "customer", the permission action would be "delete" and the recipient would be "Bob".



In this documentation, permissions are usually represented in the form **`target:action`**, omitting the recipient. In reality, a recipient is always required.

15.6.2. Securing components

We will start with the simplest form of authorization: component security. First, we will look at the **`@Restrict`** annotation.

NOTE

While the **`@Restrict`** annotation is a powerful and flexible method for security components, it cannot support EL expressions. Therefore, we recommend using the typesafe equivalent (described later in this chapter) for its compile-time safety.

15.6.2.1. The @Restrict annotation

Seam components can be secured at either the method or the class level with the **`@Restrict`** annotation. If both a method and its declaring class are annotated with **`@Restrict`**, the method restriction will take precedence and the class restriction will not apply. If a method invocation fails a security check, an exception will be thrown as per the contract for **`Identity.checkRestriction()`**. (See the section following for more information on Inline Restrictions.) Placing **`@Restrict`** on the component class itself is the equivalent of adding **`@Restrict`** to each of its methods.

An empty **`@Restrict`** implies a permission check of **`componentName:methodName`**. Take for example the following component method:

```
@Name("account")
public class AccountAction {
  @Restrict
  public void delete() {
    ...
  }
}
```

In this example, **`account:delete`** is the implied permission required to call the **`delete()`** method. This is equivalent to writing **`@Restrict("#{s:hasPermission('account','delete')}")`**. The following is another example:

```
@Restrict @Name("account")
public class AccountAction {
  public void insert() {
    ...
  }
  @Restrict("#{s:hasRole('admin')}")
  public void delete() {
    ...
  }
}
```

Here, the component class itself is annotated with **`@Restrict`**. This means that any methods without an overriding **`@Restrict`** annotation require an implicit permission check. In this case, the **`insert()`** method requires a permission of **`account:insert`**, while the **`delete()`** method requires that the user is a member of the **`admin`** role.

Before we go further, we will address the **`#{s:hasRole()}`** expression seen in the previous example. **`s:hasRole`** and **`s:hasPermission`** are EL functions that delegate to the correspondingly-named methods of the **`Identity`** class. These functions can be used within any EL expression throughout the entirety of the security API.

Being an EL expression, the value of the **`@Restrict`** annotation may refer to any object within a Seam context. This is extremely useful when checking permissions for a specific object instance. Take the following example:

```
@Name("account")
public class AccountAction {
  @In Account selectedAccount;
  @Restrict("#{s:hasPermission(selectedAccount,'modify')}")
  public void modify() {
    selectedAccount.modify();
  }
}
```

In this example, the **`hasPermission()`** function call refers to **`selectedAccount`**. The value of this variable will be looked up from within the Seam context, and passed to the **`hasPermission()`** method in **`Identity`**. This will determine whether the user has the required permissions to modify the specified **`Account`** object.

15.6.2.2. Inline restrictions

It is sometimes necessary to perform a security check in code, without using the **`@Restrict`** annotation. To do so, use **`Identity.checkRestriction()`** to evaluate a security expression, like this:

```
public void deleteCustomer() {
  Identity.instance().checkRestriction("#
{s:hasPermission(selectedCustomer,
                                                    'delete')}");
}
```

If the specified expression does not evaluate to **`true`**, one of two exceptions occurs. If the user is not logged in, a **`NotLoggedInException`** is thrown. If the user is logged in, an **`AuthorizationException`** is thrown.

You can also call the **`hasRole()`** and **`hasPermission()`** methods directly from Java code:

```
if (!Identity.instance().hasRole("admin"))
  throw new AuthorizationException("Must be admin to perform this
action");

if (!Identity.instance().hasPermission("customer", "create"))
     throw new AuthorizationException("You may not create new customers");
```

15.6.3. Security in the user interface

A well-designed interface does not present a user with options they are not permitted to use. Seam Security allows conditional rendering of page sections or individual controls based on user privileges, using the same EL expressions that are used for component security.

In this section, we will go through some examples of interface security. Say we have a login form that we want rendered only if the user is not already logged in. We can write the following with the **`identity.isLoggedIn()`** property:

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}">
```

If the user is not logged in, the login form will be rendered — very straightforward. Say we also have a menu on this page, and we want some actions to be accessed only by users in the **`manager`** role. One way you could write this is the following:

```
<h:outputLink action="#{reports.listManagerReports}"
   rendered="#{s:hasRole('manager')}"> Manager Reports
</h:outputLink>
```

This, too, is straightforward — if the user is not a member of the **`manager`** role, the outputLink will not be rendered. The **`rendered`** attribute can generally be used on the control itself, or on a surrounding **`<s:div>`** or **`<s:span>`** control.

A more complex example of conditional rendering might be the following situation: say you have a **`h:dataTable`** control on a page, and you want to render action links on its records only for users with certain privileges. The **`s:hasPermission`** EL function lets us use an object parameter to determine whether the user has the necessary permission for that object. A dataTable with secured links might look like this:

```
<h:dataTable value="#{clients}" var="cl">
  <h:column>
    <f:facet name="header">Name</f:facet>
    #{cl.name}
  </h:column>
  <h:column>
    <f:facet name="header">City</f:facet>
    #{cl.city}
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
```

```
    <s:link value="Modify Client" action="#{clientAction.modify}"
            rendered="#{s:hasPermission(cl,'modify')}"/>
    <s:link value="Delete Client" action="#{clientAction.delete}"
            rendered="#{s:hasPermission(cl,'delete')}"/>
  </h:column>
</h:dataTable>
```

15.6.4. Securing pages

To use page security, you will need a **pages.xml** file. Page security is easy to configure: simply include a **<restrict/>** element in the **page** elements that you want to secure. If no explicit restriction is specified in the **restrict** element, access via a non-Faces (GET) request requires an implied **/viewId.xhtml:render** permission, and **/viewId.xhtml:restore** permission is required when any JSF postback (form submission) originates from the page. Otherwise, the specified restriction will be evaluated as a standard security expression. Some examples are:

```
<page view-id="/settings.xhtml">
  <restrict/>
</page>
```

This page requires an implied permission of **/settings.xhtml:render** for non-Faces requests, and an implied permission of **/settings.xhtml:restore** for Faces requests.

```
<page view-id="/reports.xhtml">
  <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

Both Faces and non-Faces requests to this page require that the user is a member of the **admin** role.

15.6.5. Securing Entities

Seam Security also lets you apply security restrictions to certain actions (read, insert, update, and delete) for entities.

To secure all actions for an entity class, add a **@Restrict** annotation on the class itself:

```
@Entity
@Name("customer")
@Restrict
public class Customer {
  ...
}
```

If no expression is specified in the **@Restrict** annotation, the default action is a permission check of **entity:action**, where the permission target is the entity instance, and the **action** is either **read**, **insert**, **update** or **delete**.

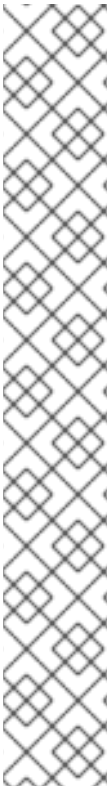
You can also restrict certain actions by placing a **@Restrict** annotation on the relevant entity life cycle method (annotated as follows):

- **@PostLoad** — Called after an entity instance is loaded from the database. Use this method to configure a **read** permission.

- **`@PrePersist`** — Called before a new instance of the entity is inserted. Use this method to configure an **`insert`** permission.
- **`@PreUpdate`** — Called before an entity is updated. Use this method to configure an **`update`** permission.
- **`@PreRemove`** — Called before an entity is deleted. Use this method to configure a **`delete`** permission.

The following example shows how an entity can be configured to perform a security check for any **`insert`** operations. Note that the method need not perform any action; it is only important that it be annotated correctly:

```
@PrePersist
@Restrict
public void prePersist() {}
```



NOTE

You can also specify the callback method in **`/META-INF/orm.xml`**:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
          http://java.sun.com/xml/ns/persistence/orm
          http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
          version="1.0">

  <entity class="Customer">
    <pre-persist method-name="prePersist" />
  </entity>

</entity-mappings>
```

You will still need to annotate the **`prePersist()`** method on **`Customer`** with **`@Restrict`**.

The following configuration is based on the Seamspace example, and checks if the authenticated user has permission to insert a new **`MemberBlog`** record. The entity being checked is automatically inserted into the working memory (in this case, **`MemberBlog`**):

```
rule InsertMemberBlog
  no-loop
  activation-group "permissions"
when
  principal: Principal()
  memberBlog: MemberBlog(member : member ->

(member.getUsername().equals(principal.getName())))
  check: PermissionCheck(target == memberBlog,
                         action == "insert", granted == false)
```

```
then
  check.grant();
end;
```

This rule grants the permission **`memberBlog:insert`** if the name of the currently authenticated user (indicated by the **`Principal`** fact) matches that of the member for whom the blog entry is being created. The **`principal: Principal()`** structure is a variable binding. It binds the instance of the **`Principal`** object placed in the working memory during authentication, and assigns it to a variable called **`principal`**. Variable bindings let the variable be referenced in other places, such as the following line, which compares the member name to the **`Principal`** name. For further details, refer to the JBoss Rules documentation.

Finally, install a listener class to integrate Seam Security with your JPA provider.

15.6.5.1. Entity security with JPA

Security checks for EJB3 entity beans are performed with an **`EntityListener`**. Install this listener with the following **`META-INF/orm.xml`** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="
                   http://java.sun.com/xml/ns/persistence/orm
                   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                 version="1.0">

  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener
                class="org.jboss.seam.security.EntitySecurityListener"/>
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>

</entity-mappings>
```

15.6.5.2. Entity security with a Managed Hibernate Session

If you use a Hibernate **`SessionFactory`** configured with Seam, and use annotations or **`orm.xml`**, you do not need to make any changes to use entity security.

15.6.6. Typesafe Permission Annotations

Seam provides a number of alternative annotations to **`@Restrict`**. These support arbitrary EL expressions differently, which gives them additional compile-time safety.

Seam comes with a set of annotations for standard CRUD-based permissions. The following annotations are provided in the **`org.jboss.seam.annotations.security`** package:

- @Insert
- @Read

- @Update
- @Delete

To use these annotations, place them on the method or parameter for which you wish to perform a security check. When placed on a method, they specify a target class for which the permission will be checked. Take the following example:

```
@Insert(Customer.class)
public void createCustomer() { ... }
```

Here, a permission check will be performed for the user to ensure that they have permission to create new **`Customer`** objects. The target of the permission check is **`Customer.class`** (the actual **`java.lang.Class`** instance itself), and the action is the lower case representation of the annotation name, which in this example is **`insert`**.

You can annotate a component method's parameters in the same way, as follows. If you do this, you need not specify a permission target, since the parameter value itself will be the target of the permission check.

```
public void updateCustomer(@Update Customer customer) {
  ...
}
```

To create your own security annotation, just annotate it with **`@PermissionCheck`**. For example:

```
@Target({METHOD, PARAMETER})
@Documented
@Retention(RUNTIME)
@Inherited
@PermissionCheck
public @interface Promote {
  Class value() default void.class;
}
```

If you wish to override the default permission action name (the lower case version of the annotation name) with another value, you can specify this within the **`@PermissionCheck`** annotation:

```
@PermissionCheck("upgrade")
```

15.6.7. Typesafe Role Annotations

In addition to typsesafe permission annotation support, Seam Security provides typesafe role annotations that let you restrict access to component methods based on the role memberships of the currently authenticated user. Seam provides one such annotation (**`org.jboss.seam.annotations.security.Admin`**) out of the box. This restricts access of a particular method to users that belong to the **`admin`** role, as long as such a role is supported by your application. To create your own role annotations, meta-annotate them with **`org.jboss.seam.annotations.security.RoleCheck`** as in the following example:

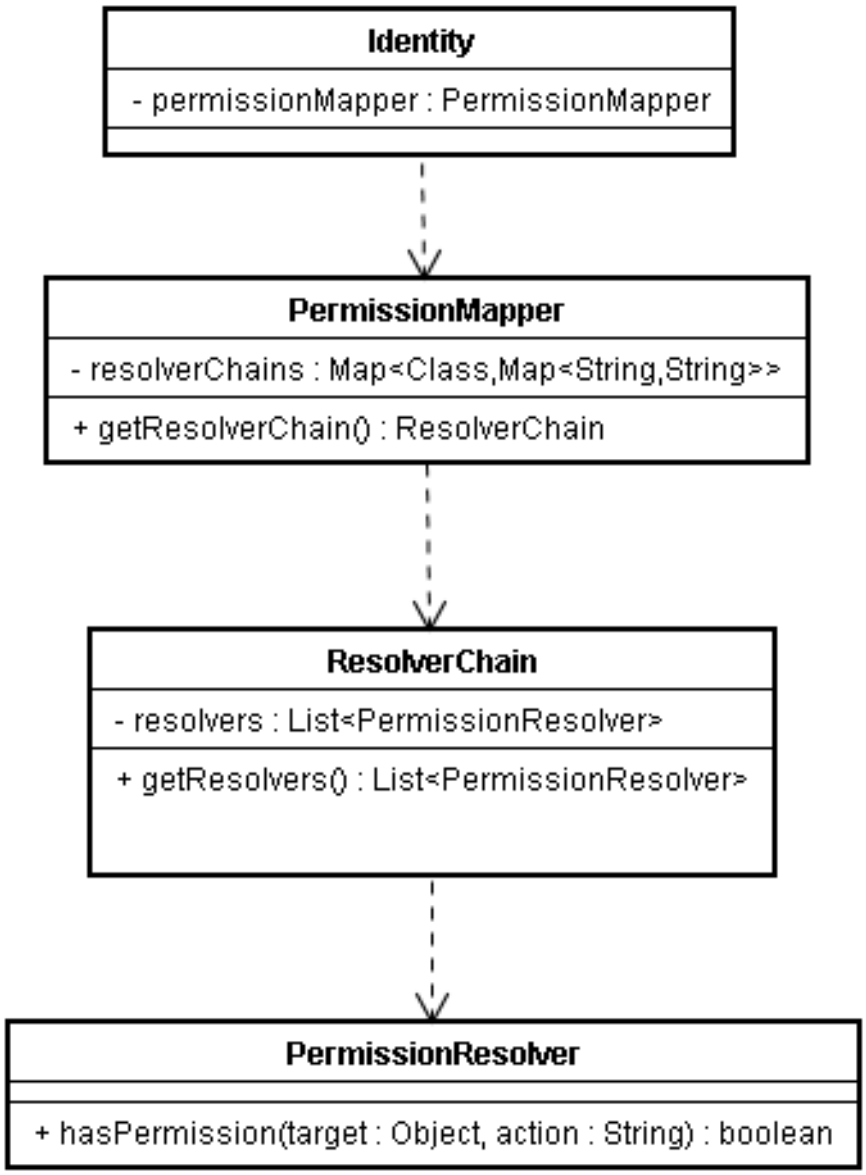
```
@Target({METHOD})
@Documented
@Retention(RUNTIME)
```

```
@Inherited
@RoleCheck
public @interface User { }
```

Any methods subsequently annotated with the **`@User`** annotation will be automatically intercepted. The user will be checked for membership of the corresponding role name (the lower case version of the annotation name, in this case **`user`**).

15.6.8. The Permission Authorization Model

Seam Security provides an extensible framework for resolving application permissions. The following class diagram shows an overview of the main components of the permission framework:



The relevant classes are explained in more detail in the following sections.

15.6.8.1. PermissionResolver

An interface that provides methods for resolving individual object permissions. Seam provides the following built-in **`PermissionResolver`** implementations, which are described in greater detail later in the chapter:

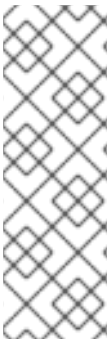
- **`RuleBasedPermissionResolver`** — Resolves rule-based permission checks with Drools.
- **`PersistentPermissionResolver`** — Stores object permissions in a permanent store, such as a relational database.

15.6.8.1.1. Writing your own PermissionResolver

Implementing your own permission resolver is simple. The **`PermissionResolver`** interface defines two methods that must be implemented, as seen in the following table. If your **`PermissionResolver`** is deployed in your Seam project, it will be scanned automatically during deployment and registered with the default **`ResolverChain`**.

Table 15.7. PermissionResolver interface

Return type	Method	Description
`boolean`	**`hasPermission(Object target, String action)`**	This method resolves whether the currently authenticated user (obtained via a call to **`Identity.getPrincipal()`**) has the permission specified by the **`target`** and **`action`** parameters. It returns **`true`** if the user has the specified permission, or **`false`** if they do not.
`void`	**`filterSetByAction(Set<Object> targets, String action)`**	This method removes any objects from the specified set that would return **`true`** if passed to the **`hasPermission()`** method with the same **`action`** parameter value.



NOTE

Because they are cached in the user's session, any custom **`PermissionResolver`** implementations must adhere to several restrictions. Firstly, they cannot contain any state that is more fine-grained than the session scope, and the component itself should be either application- or session-scoped. Secondly, they must not use dependency injection, as they may be accessed from multiple threads simultaneously. For optimal performance, we recommend annotating with **`@BypassInterceptors`** to bypass Seam's interceptor stack altogether.

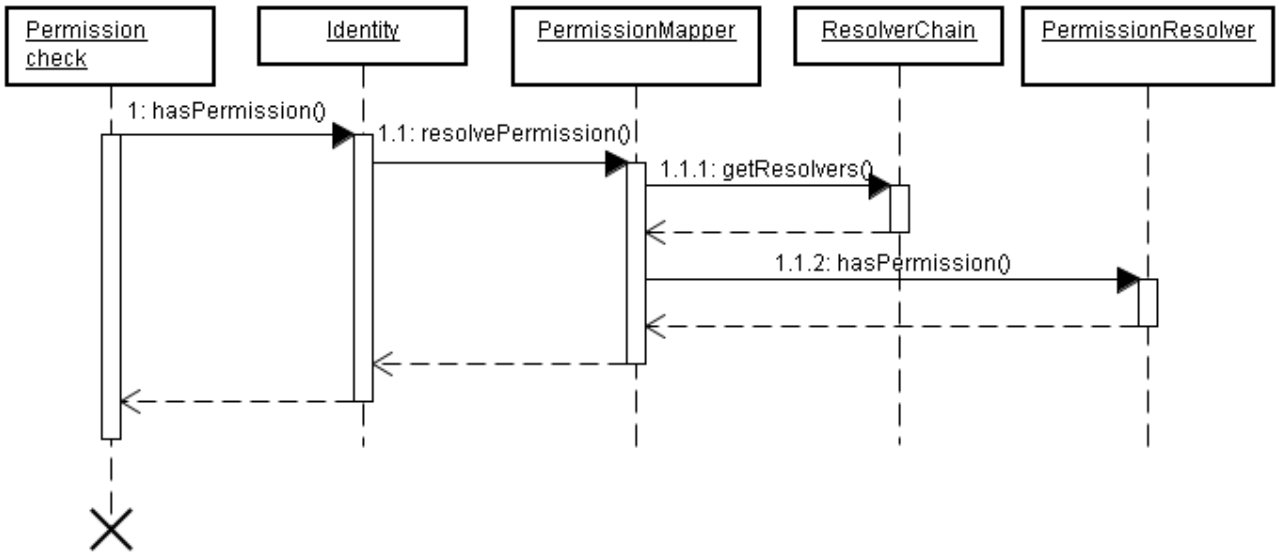
15.6.8.2. ResolverChain

A **`ResolverChain`** contains an ordered list of **`PermissionResolver`**s, to resolve object permissions for a particular object class or permission target.

The default **`ResolverChain`** consists of all permission resolvers discovered during application deployment. The **`org.jboss.seam.security.defaultResolverChainCreated`** event is raised (and the **`ResolverChain`** instance passed as an event parameter) when the default **`ResolverChain`** is created. This allows additional resolvers that were not discovered during deployment to be added, or for resolvers that are in the chain to be re-ordered or removed.

The following sequence diagram shows the interaction between the components of the permission

framework during a permission check. A permission check can originate from a number of possible sources: the security interceptor, the **`s:hasPermission`** EL function, or via an API call to **`Identity.checkPermission`**:



- 1. A permission check is initiated (either in code or via an EL expression), resulting in a call to **`Identity.hasPermission()`**.
- 1.1. **`Identity`** invokes **`PermissionMapper.resolvePermission()`**, passing in the permission to be resolved.
- 1.1.1. **`PermissionMapper`** maintains a **`Map`** of **`ResolverChain`** instances, keyed by class. It uses this map to locate the correct **`ResolverChain`** for the permission's target object. Once it has the correct **`ResolverChain`**, it retrieves the list of **`PermissionResolver`**s it contains by calling **`ResolverChain.getResolvers()`**.
- 1.1.2. For each **`PermissionResolver`** in the **`ResolverChain`**, the **`PermissionMapper`** invokes its **`hasPermission()`** method, passing in the permission instance to be checked. If the **`PermissionResolver`**s return **`true`**, the permission check has succeeded and the **`PermissionMapper`** also returns **`true`** to **`Identity`**. If none of the **`PermissionResolver`**s return **`true`**, then the permission check has failed.

15.6.9. RuleBasedPermissionResolver

One of the built-in permission resolvers provided by Seam. This evaluates permissions based on a set of Drools (JBoss Rules) security rules. Some advantages to the rule engine are a centralized location for the business logic used to evaluate user permissions, and speed — Drools algorithms are very efficient for evaluating large numbers of complex rules involving multiple conditions.

15.6.9.1. Requirements

If you want to use the rule-based permission features provided by Seam Security, Drools requires the following **JAR** files to be distributed with your project:

- drools-api.jar
- drools-compiler.jar
- drools-core.jar

- janino.jar
- antlr-runtime.jar
- mvel2.jar

15.6.9.2. Configuration

The configuration for **`RuleBasedPermissionResolver`** requires that a Drools rule base is first configured in **`components.xml`**. By default, it expects the rule base to be named **`securityRules`**, as per the following example:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:drools="http://jboss.com/products/seam/drools"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core
               http://jboss.com/products/seam/core-2.2.xsd
               http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/drools
               http://jboss.com/products/seam/drools-2.2.xsd
               http://jboss.com/products/seam/security
               http://jboss.com/products/seam/security-2.2.xsd">

  <drools:rule-base name="securityRules">
    <drools:rule-files>
      <value>/META-INF/security.drl</value>
    </drools:rule-files>
  </drools:rule-base>

</components>
```

The default rule base name can be overridden by specifying the **`security-rules`** property for **`RuleBasedPermissionResolver`**:

```
<security:rule-based-permission-resolver
          security-rules="#{prodSecurityRules}"/>
```

Once the **`RuleBase`** component is configured, you must write the security rules.

15.6.9.3. Writing Security Rules

The first step to writing security rules is to create a new rule file in the **`/META-INF`** directory of your application's jar file. This file should be named **`security.drl`** or similar, but can be named anything as long as it is configured correspondingly in **`components.xml`**.

We recommend the Drools documentation when you write your rules file. A simple example of rules file contents is:

```
package MyApplicationPermissions;
```

```
import org.jboss.seam.security.permission.PermissionCheck;
import org.jboss.seam.security.Role;

rule CanUserDeleteCustomers
when
  c: PermissionCheck(target == "customer", action == "delete")
  Role(name == "admin")
then
  c.grant();
end
```

Here, the first thing we see is the package declaration. A package in Drools is a collection of rules. The package name does not relate to anything outside the scope of the rule base, so it can be given any name.

Next, we have several import statements for the **`PermissionCheck`** and **`Role`** classes. These imports inform the rules engine that our rules will refer to these classes.

Finally, we have the rule code. Each rule within a package should have a unique name, usually to describe the rule's purpose. In this case our rule is called **`CanUserDeleteCustomers`** and will be used to check whether a user is allowed to delete a customer record.

There are two distinct sections in the body of the rule definition. Rules have a left hand side (LHS) and a right hand side (RHS). The LHS is the conditional portion of the rule, that is, a list of conditions that must be satisfied for the rule to fire. The LHS is represented by the **`when`** section. The RHS is the consequence or action section of the rule, which will only be fired if all conditions in the LHS are met. The RHS is represented by the **`then`** section. The end of the rule is denoted by the **`end`** line.

There are two conditions listed in the example LHS. The first condition is:

```
c: PermissionCheck(target == "customer", action == "delete")
```

More plainly, this condition states that, to be fulfilled, there must be a **`PermissionCheck`** object with a **`target`** property equal to **`customer`**, and an **`action`** property equal to **`delete`** within the working memory.

Working memory is also known as a *stateful session* in Drools terminology. It is a session-scoped object containing the contextual information that the rules engine requires to make a decision about a permission check. Each time the **`hasPermission()`** method is called, a temporary **`PermissionCheck`** object, or *Fact*, is inserted into the working memory. This **`PermissionCheck`** corresponds exactly to the permission being checked, so if you call **`hasPermission("account", "create")`**, a **`PermissionCheck`** object with a **`target`** equal to "account" and **`action`** equal to "create" will be inserted into the working memory for the duration of the permission check.

Other than the **`PermissionCheck`** facts, there is also an **`org.jboss.seam.security.Role`** fact for each role that the authenticated user is a member of. These **`Role`** facts are synchronized with the user's authenticated roles at the beginning of every permission check. As a consequence, any **`Role`** object inserted into the working memory during the course of a permission check will be removed before the next permission check occurs, unless the authenticated user is actually a member of that role. The working memory also contains the **`java.security.Principal`** object created as a result of the authentication process.

You can insert additional long-lived facts into the working memory by calling **`RuleBasedPermissionResolver.instance().getSecurityContext().insert ()`**, which

passes the object as a parameter. **Role** objects are the exception, here, since they are synchronized at the start of each permission check.

To return to our simple example, the first line of our LHS is prefixed with **c:**. This is a variable binding, and is used to refer back to the object matching the condition (in this case, the **PermissionCheck**). The second line of the LHS is:

```
Role(name == "admin")
```

This condition states that there must be a **Role** object with a **name** of "admin" within the working memory. So, if you are checking for the **customer:delete** permission and the user is a member of the **admin** role, this rule will fire.

The RHS shows us the consequence of the rule firing:

```
c.grant()
```

The RHS consists of Java code. In this case it invokes the **grant()** method of the **c** object, which is a variable binding for the **PermissionCheck** object. Other than the **name** and **action** properties of the **PermissionCheck** object, there is also a **granted** property. This is initially set to **false**. Calling **grant()** on a **PermissionCheck** sets the **granted** property to **true**. This means the permission check succeeded, and the user has permission to carry out the action that prompted the permission check.

15.6.9.4. Non-String permission targets

So far, we have only looked at permission checks for String-literal permission targets. However, you can also write security rules for more complex permission targets. For example, say you want to write a security rule to allow your users to create blog comments. The following rule shows one way this can be expressed, by requiring that the target of the permission check be an instance of **MemberBlog**, and that the currently authenticated user be a member of the **user** role:

```
rule CanCreateBlogComment
  no-loop
  activation-group "permissions"
when
  blog: MemberBlog()
  check: PermissionCheck(target == blog, action == "create",
                         granted == false)
  Role(name == "user")
then
  check.grant();
end
```

15.6.9.5. Wildcard permission checks

It is possible to implement a wildcard permission check (which allows all actions for a given permission target), by omitting the **action** constraint for the **PermissionCheck** in your rule, like so:

```
rule CanDoAnythingToCustomersIfYouAreAnAdmin
when
  c: PermissionCheck(target == "customer")
  Role(name == "admin")
```

```
then
  c.grant();
end;
```

This rule allows users with the **admin** role to perform *any* action for any **customer** permission check.

15.6.10. PersistentPermissionResolver

Another built-in permission resolver provided by Seam, **PersistentPermissionResolver**, allows permissions to be loaded from persistent storage, such as a relational database. This permission resolver provides Access Control List-style instance-based security, allowing specific object permissions to be assigned to individual users and roles. It also allows persistent, arbitrarily-named permission targets (which are not necessarily object/class based) to be assigned in the same way.

15.6.10.1. Configuration

To use **PersistentPermissionResolver**, you must configure a valid **PermissionStore** in **components.xml**. If this is not configured, the **PersistentPermissionResolver** will attempt to use the default permission store, Section 15.4.2.4, “JpaIdentityStore Events”. To use a permission store other than the default, configure the **permission-store** property as follows:

```
<security:persistent-permission-resolver
          permission-store="#{myCustomPermissionStore}"/>
```

15.6.10.2. Permission Stores

PersistentPermissionResolver requires a permission store to connect to the back-end storage where permissions are persisted. Seam provides one **PermissionStore** implementation out of the box, **JpaPermissionStore**, which stores permissions inside a relational database. You can write your own permission store by implementing the **PermissionStore** interface, which defines the following methods:

Table 15.8. PermissionStore interface

Return type	Method	Description
List<Permission>	**listPermissions(Object target)**	This method should return a **List** of **Permission** objects representing all the permissions granted for the specified target object.
List<Permission>	**listPermissions(Object target, String action)**	This method should return a **List** of **Permission** objects representing all the permissions with the specified action granted for the specified target object.

Return type	Method	Description
`List<Permission>`	`listPermissions(Set<Object> targets, String action)`	This method should return a `List` of `Permission` objects representing all the permissions with the specified action granted for the specified set of target objects.
`boolean`	`grantPermission(Permission)`	This method should persist the specified `Permission` object to the back-end storage, and return `true` if successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	This method should persist all of the `Permission` objects contained in the specified `List`, and return `true` if successful.
`boolean`	`revokePermission(Permission permission)`	This method should remove the specified `Permission` object from persistent storage.
`boolean`	`revokePermissions(List<Permission> permissions)`	This method should remove all of the `Permission` objects in the specified list from persistent storage.
`List<String>`	`listAvailableActions(Object target)`	This method should return a list of all available actions (as Strings) for the class of the specified target object. It is used in conjunction with permission management to build the user interface for granting specific class permissions.

15.6.10.3. JpaPermissionStore

The Seam-provided default `PermissionStore` implementation, which stores permissions in a relational database. It must be configured with either one or two entity classes for storing user and role permissions before it can be used. These entity classes must be annotated with a special set of security

annotations to configure the entity properties that correspond to various aspects of the stored permissions.

If you want to use the same entity (that is, a single database table) to store both user and role permissions, then you only need to configure the **`user-permission-class`** property. To user separate tables for user and role permission storage, you must also configure the **`role-permission-class`** property.

For example, to configure a single entity class to store both user and role permissions:

```
<security:jpa-permission-store
          user-permission-class="com.acme.model.AccountPermission"/>
```

To configure separate entity classes for storing user and role permissions:

```
<security:jpa-permission-store
          user-permission-class="com.acme.model.UserPermission"
          role-permission-class="com.acme.model.RolePermission"/>
```

15.6.10.3.1. Permission annotations

The entity classes that contain the user and role permissions must be configured with a special set of annotations in the **`org.jboss.seam.annotations.security.permission`** package. The following table describes these annotations:

Table 15.9. Entity Permission annotations

Annotation	Target	Description
`@PermissionTarget`	**`FIELD,METHOD`**	This annotation identifies the entity property containing the permission target. The property should be of type **`java.lang.String`**.
`@PermissionAction`	**`FIELD,METHOD`**	This annotation identifies the entity property containing the permission action. The property should be of type **`java.lang.String`**.
`@PermissionUser`	**`FIELD,METHOD`**	This annotation identifies the entity property containing the recipient user for the permission. It should be of type **`java.lang.String`** and contain the user's username.
`@PermissionRole`	**`FIELD,METHOD`**	This annotation identifies the entity property containing the recipient role for the permission. It should be of type **`java.lang.String`** and contain the role name.

Annotation	Target	Description
`@PermissionDiscriminator`	**`FIELD,METHOD`**	This annotation should be used when the same entity/table stores both user and role permissions. It identifies the property of the entity being used to discriminate between user and role permissions. By default, if the column value contains the string literal **`user`**, then the record will be treated as a user permission. If it contains the string literal **`role`**, it will be treated as a role permission. You can also override these defaults by specifying the **`userValue`** and **`roleValue`** properties within the annotation. For example, to use **`u`** and **`r`** instead of **`user`** and **`role`**, write the annotation like so: `@PermissionDiscriminator( userValue = "u", roleValue = "r")`

15.6.10.3.2. Example Entity

The following is an example of an entity class that stores both user and role permissions, taken from the Seamspace example.

```
@Entity
public class AccountPermission implements Serializable {
  private Integer permissionId;
  private String recipient;
  private String target;
  private String action;
  private String discriminator;

  @Id @GeneratedValue
  public Integer getPermissionId() {
    return permissionId;
  }

  public void setPermissionId(Integer permissionId) {
    this.permissionId = permissionId;
  }

  @PermissionUser @PermissionRole
  public String getRecipient() {
    return recipient;
  }

  public void setRecipient(String recipient) {
    this.recipient = recipient;
  }

  @PermissionTarget
```

```
  public String getTarget() {
    return target;
  }

  public void setTarget(String target) {
    this.target = target;
  }

  @PermissionAction
  public String getAction() {
    return action;
  }

  public void setAction(String action) {
    this.action = action;
  }

  @PermissionDiscriminator
  public String getDiscriminator() {
    return discriminator;
  }

  public void setDiscriminator(String discriminator) {
    this.discriminator = discriminator;
  }
}
```

Here, the **`getDiscriminator()`** method has been annotated with **`@PermissionDiscriminator`**, to allow **`JpaPermissionStore`** to determine which records represent user permissions and which represent role permissions. The **`getRecipient()`** method is annotated with both **`@PermissionUser`** and **`@PermissionRole`**. This means that the **`recipient`** property of the entity will either contain the name of the user or the name of the role, depending on the value of the **`discriminator`** property.

15.6.10.3.3. Class-specific Permission Configuration

The permissions included in the **`org.jboss.seam.annotation.security.permission`** package can be used to configure a specific set of allowable permissions for a target class.

Table 15.10. Class Permission Annotations

Annotation	Target	Description
`@Permissions`	**`TYPE`**	A container annotation, which can contain an array of **`@Permission`** annotations.
`@Permission`	**`TYPE`**	This annotation defines a single allowable permission action for the target class. Its **`action`** property must be specified, and an optional **`mask`** property may also be specified if permission actions are to be persisted as bitmasked values (see section following).

The following shows the above annotations in use. They can also be seen in the SeamSpace example.

```
@Permissions({
  @Permission(action = "view"),
  @Permission(action = "comment")
})

@Entity
public class MemberImage implements Serializable {...}
```

This example demonstrates how two allowable permission actions, **`view`** and **`comment`** can be declared for the entity class **`MemberImage`**.

15.6.10.3.4. Permission masks

By default, multiple permissions for the same target object and recipient will be persisted as a single database record, with the **`action`** property/column containing a list of granted actions, separated by commas. You can use a bitmasked integer value to store the list of permission actions — this reduces the amount of physical storage required to persist a large number of permissions.

For example, if recipient "Bob" is granted both the **`view`** and **`comment`** permissions for a particular **`MemberImage`** (an entity bean) instance, then by default the **`action`** property of the permission entity will contain "**`view,comment`**", representing the two granted permission actions. Or, if you are using bitmasked values defined as follows:

```
@Permissions({
  @Permission(action = "view", mask = 1),
  @Permission(action = "comment", mask = 2)
})

@Entity
public class MemberImage implements Serializable {...}
```

The **`action`** property will contain "3" (with both the 1 bit and 2 bit switched on). For a large number of allowable actions for any particular target class, the storage required for the permission records is greatly reduced by using bitmasked actions.

IMPORTANT

The **`mask`** values specified must be powers of 2.

15.6.10.3.5. Identifier Policy

When storing or looking up permissions, **`JpaPermissionStore`** must be able to uniquely identify specific object instances. To achieve this, an *identifier strategy* may be assigned to each target class so that unique identifier values can be generated. Each identifier strategy implementation knows how to generate unique identifiers for a particular type of class, and creating new identifier strategies is simple.

The **`IdentifierStrategy`** interface is very simple, declaring only two methods:

```
public interface IdentifierStrategy {
  boolean canIdentify(Class targetClass);
  String getIdentifier(Object target);
}
```

The first method, **canIdentify()**, returns **true** if the identifier strategy is capable of generating a unique identifier for the specified target class. The second method, **getIdentifier()**, returns the unique identifier value for the specified target object.

Seam also provides two **IdentifierStrategy** implementations, **ClassIdentifierStrategy** and **EntityIdentifierStrategy**, which are described in the sections following.

To explicitly configure a specific identifier strategy for a particular class, annotate the strategy with **org.jboss.seam.annotations.security.permission.Identifier** and set the value to a concrete implementation of the **IdentifierStrategy** interface. You may also specify a **name** property. (The effect of this depends upon the **IdentifierStrategy** implementation used.)

15.6.10.3.6. ClassIdentifierStrategy

This identifier strategy generates unique identifiers for classes, and uses the value of the **name** (if specified) in the **@Identifier** annotation. If no **name** property is provided, the identifier strategy attempts to use the component name of the class (if the class is a Seam component). It will create an identifier based on the class name (excluding the package name) as a last resort. For example, the identifier for the following class will be **customer**:

```
@Identifier(name = "customer")
public class Customer {...}
```

The identifier for the following class will be **customerAction**:

```
@Name("customerAction")
public class CustomerAction {...}
```

Finally, the identifier for the following class will be **Customer**:

```
public class Customer {...}
```

15.6.10.3.7. EntityIdentifierStrategy

This identifier strategy generates unique identifiers for entity beans. It concatenates the entity name (or otherwise configured name) with a string representation of the primary key value of the entity. The rules for generating the name section of the identifier are similar to those in **ClassIdentifierStrategy**. The primary key value (that is, the entity ID) is obtained with the **PersistenceProvider** component, which can determine the value regardless of the persistence implementation being used in the Seam application. For entities not annotated with **@Entity**, you must explicitly configure the identifier strategy on the entity class itself, like this:

```
@Identifier(value = EntityIdentifierStrategy.class)
public class Customer {...}
```

Assume we have the following entity class:

```
@Entity
public class Customer {
  private Integer id;
  private String firstName;
  private String lastName;
```

```
  @Id
  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  public String getFirstName() { return firstName; }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() { return lastName; }
  public void setLastName(String lastName) { this.lastName = lastName; }
}
```

For a **`Customer`** instance with an **`id`** value of **`1`**, the value of the identifier would be **`Customer:1`**. If the entity class is annotated with an explicit identifier name, like so:

```
@Entity @Identifier(name = "cust")
public class Customer {...}
```

Then a **`Customer`** with an **`id`** value of **`123`** would have an identifier value of "**`cust:123`**".

15.7. PERMISSION MANAGEMENT

Just as Seam Security provides an Identity Management API to let you manage users and roles, it also provides a Permissions Management API to let you manage persistent user permissions — the **`PermissionManager`** component.

15.7.1. PermissionManager

The **`PermissionManager`** component is an application-scoped Seam component that provides a number of permission-management methods. It must be configured with a permission store before use. By default, it will attempt to use **`JpaPermissionStore`**. To configure a custom permission store, specify the **`permission-store`** property in **`components.xml`**:

```
<security:permission-manager permission-store="#{ldapPermissionStore}"/>
```

The following table describes each of the methods provided by **`PermissionManager`**:

Table 15.11. PermissionManager API methods

Return type	Method	Description
`List<Permission>`	**`listPermissions(Object target, String action)`**	Returns a list of **`Permission`** objects representing all of the permissions that have been granted for the specified target and action.

Return type	Method	Description
`List<Permission>`	`listPermissions(Object target)`	Returns a list of **`Permission`** objects representing all of the permissions that have been granted for the specified target and action.
`boolean`	`grantPermission(Permission permission)`	Persists (grants) the specified **`Permission`** to the back-end permission store. Returns **`true`** if the operation succeeds.
`boolean`	`grantPermissions(List<Permission> permissions)`	Persists (grants) the specified list of **`Permission`**s to the back-end permission store. Returns **`true`** if the operation succeeds.
`boolean`	`revokePermission(Permission permission)`	Removes (revokes) the specified **`Permission`** from the back-end permission store. Returns **`true`** if the operation succeeds.
`boolean`	`revokePermissions(List<Permission> permissions)`	Removes (revokes) the specified list of **`Permission`**s from the back-end permission store. Returns **`true`** if the operation succeeds.
`List<String>`	`listAvailableActions(Object target)`	Returns a list of the available actions for the specified target object. The actions that this method returns are dependent on the **`@Permission`** annotations configured on the target object's class.

15.7.2. Permission checks for PermissionManager operations

To invoke **`PermissionManager`** methods, the currently authenticated user must be authorized to perform that management operation. The following table lists the permissions required to invoke a particular method.

Table 15.12. Permission Management Security Permissions

Method	Permission Target	Permission Action
`listPermissions()`	The specified `target`.	`seam.read-permissions`
`grantPermission()`	The target of the specified `Permission`, or each of the targets for the specified list of `Permission`s (depending on the method called).	`seam.grant-permission`
`grantPermission()`	The target of the specified `Permission`.	`seam.grant-permission`
`grantPermissions()`	Each of the targets of the specified list of `Permission`s.	`seam.grant-permission`
`revokePermission()`	The target of the specified `Permission`.	`seam.revoke-permission`
`revokePermissions()`	Each of the targets of the specified list of `Permission`s.	`seam.revoke-permission`

15.8. SSL SECURITY

Seam includes basic support for serving sensitive pages via the HTTPS protocol. To configure this, specify a `scheme` for the page in `pages.xml`. The following example shows how the view `/login.xhtml` can be configured to use HTTPS:

```
<page view-id="/login.xhtml" scheme="https"/>
```

This configuration automatically extends to both `s:link` and `s:button` JSF controls, which (when specifying the `view`) will render the link under the correct protocol. Based on the previous example, the following link will use the HTTPS protocol because `/login.xhtml` is configured to use it:

```
<s:link view="/login.xhtml" value="Login"/>
```

If a user browses directly to a view with the incorrect protocol, a redirect is triggered, and the same view will be reloaded with the correct protocol. For example, browsing to a `scheme="https"` page with HTTP triggers a redirect to the same page using HTTPS.

You can also configure a *default scheme* for all pages. This is useful if you only want to use HTTPS for a few pages. If no default scheme is specified, the current scheme will be used. So, once the user accesses a page requiring HTTPS, then HTTPS continues to be used after the user has navigated to other non-HTTPS pages. This is good for security, but not for performance. To define HTTP as the default `scheme`, add this line to `pages.xml`:

```
<page view-id="*" scheme="http" />
```

If *none* of the pages in your application use HTTPS, you need not define a default scheme.

You can configure Seam to automatically invalidate the current HTTP session each time the scheme changes. To do so, add this line to **components.xml**:

```
<web:session invalidate-on-scheme-change="true"/>
```

This option offers more protection from session ID sniffing and sensitive data leakage from pages using HTTPS to pages using HTTP.

15.8.1. Overriding the default ports

If you wish to configure the HTTP and HTTPS ports manually, you can do so in **pages.xml** by specifying the **http-port** and **https-port** attributes on the **pages** element:

```
<pages xmlns="http://jboss.com/products/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.2.xsd"
       no-conversation-view-id="/home.xhtml"
       login-view-id="/login.xhtml" http-port="8080" https-port="8443">
```

15.9. CAPTCHA

Though not strictly part of the security API, Seam provides a built-in CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) algorithm to prevent automated processes from interacting with your application.

15.9.1. Configuring the CAPTCHA Servlet

To use CAPTCHA, you need to configure the Seam Resource Servlet, which provides your pages with CAPTCHA challenge images. Add the following to **web.xml**:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-
class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

15.9.2. Adding a CAPTCHA to a form

It is easy to add a CAPTCHA challenge to a form:

```
<h:graphicImage value="/seam/resource/captcha"/>
  <h:inputText id="verifyCaptcha" value="#{captcha.response}"
     required="true">
```

```
    <s:validate />
  </h:inputText>
<h:message for="verifyCaptcha"/>
```

That is all you need to do. The **graphicImage** control displays the CAPTCHA challenge, and the **inputText** receives the user's response. The response is automatically validated against the CAPTCHA when the form is submitted.

15.9.3. Customizing the CAPTCHA algorithm

You can customize the CAPTCHA algorithm by overriding the built-in component:

```
@Name("org.jboss.seam.captcha.captcha")
@Scope(SESSION)
public class HitchhikersCaptcha extends Captcha
{
  @Override @Create
  public void init() {
    setChallenge("What is the answer to life, the universe and
everything?");
    setCorrectResponse("42");
  }

  @Override
  public BufferedImage renderChallenge() {
    BufferedImage img = super.renderChallenge();
    img.getGraphics().drawOval(5, 3, 60, 14); //add an obscuring
decoration
    return img;
  }
}
```

15.10. SECURITY EVENTS

The following table describes a number of events (see Chapter 7, *Events, interceptors and exception handling*) raised by Seam Security in response to certain security-related events.

Table 15.13. Security Events

Event Key	Description
org.jboss.seam.security.loginSuccessful	Raised when a log in attempt is successful.
org.jboss.seam.security.loginFailed	Raised when a log in attempt fails.
org.jboss.seam.security.alreadyLoggedIn	Raised when a user that is already authenticated attempts to log in again.
org.jboss.seam.security.notLoggedIn	Raised when a security check fails when the user is not logged in.

Event Key	Description
org.jboss.seam.security.notAuthorized	Raised when a security check fails because the user is logged in, but does not have sufficient privileges.
org.jboss.seam.security.preAuthenticate	Raised just prior to user authentication.
org.jboss.seam.security.postAuthenticate	Raised just after user authentication.
org.jboss.seam.security.loggedOut	Raised after the user has logged out.
org.jboss.seam.security.credentialsUpdated	Raised when the user's credentials have been changed.
org.jboss.seam.security.rememberMe	Raised when the Identity's rememberMe property is changed.

15.11. RUN AS

Users sometimes need to perform certain operations with elevated privileges — for example, an unauthenticated user may need to create a new user account. Seam Security provides support in this situation with the **RunAsOperation** class. This class allows either the **Principal** or **Subject**, or the user's roles, to be overridden for a single set of operations.

The following code demonstrates **RunAsOperation** usage. The **addRole()** method is called to provide a set of roles to 'borrow' for the duration of the operation. The **execute()** method contains the code that will be executed with the elevated privileges.

```
new RunAsOperation() {
  public void execute() {
    executePrivilegedOperation();
  }
}.addRole("admin")
 .run();
```

Similarly, the **getPrincipal()** or **getSubject()** methods can also be overidden to specify the **Principal** and **Subject** instances to use for the duration of the operation. Finally, the **run()** method is used to carry out the **RunAsOperation**.

15.12. EXTENDING THE IDENTITY COMPONENT

If your application has special security requirements, you may need to extend your Identity component. The following example shows an Identity component extended with an additional **companyCode** field to handle credentials. (Usually this would be handled by a **Credentials** component.) The install precedence of **APPLICATION** ensures that this extended Identity is installed instead of the built-in Identity.

```
@Name("org.jboss.seam.security.identity")
```

```
@Scope(SESSION)
@Install(precedence = APPLICATION)
@BypassInterceptors
@Startup
public class CustomIdentity extends Identity {
  private static final LogProvider log =
                       Logging.getLogProvider(CustomIdentity.class);

  private String companyCode;

  public String getCompanyCode() {
    return companyCode;
  }

  public void setCompanyCode(String companyCode) {
    this.companyCode = companyCode;
  }

  @Override
  public String login() {
    log.info("###### CUSTOM LOGIN CALLED ######");
    return super.login();
  }
}
```

WARNING

An **`Identity`** component must be marked **`@Startup`** so that it is available immediately after the **`SESSION`** context begins. Failing to do this may render certain Seam functionality inoperable in your application.

15.13. OPENID

WARNING

Technology Preview features are not fully supported under Red Hat subscription level agreements (SLAs), may not be functionally complete, and are not intended for production use. However, these features provide early access to upcoming product innovations, enabling customers to test functionality and provide feedback during the development process. As Red Hat considers making future iterations of Technology Preview features generally available, we will provide commercially reasonable efforts to resolve any reported issues that customers experience when using these features.

OpenID is a community standard for external web-based authentication. Any web application can supplement (or replace) its local authentication handling by delegating responsibility to an external OpenID server selected by the user. This benefits both user and developer — the user (who no longer needs to remember login details for multiple web applications), and the developer (who need not maintain an entire complex authentication system).

When using OpenID, the user selects an OpenID provider, and the provider assigns the user an OpenID. The ID takes the form of a URL — **`http://maximoburrito.myopenid.com`**, for example. (The **`http://`** portion of the identifier can be omitted when logging into a site.) The web application (known as a *relying party*) determines which OpenID server to contact and redirects the user to the remote site for authentication. When authentication succeeds, the user is given the (cryptographically secure) token proving his identity and is redirected back to the original web application. The local web application can then assume that the user accessing the application owns the OpenID presented.

However, authentication does not imply authorization. The web application must still determine how to treat the OpenID authentication. The web application can choose to treat the user as instantly logged in and grant full access to the system, or it can attempt to map the OpenID to a local user account and prompt unregistered users to register. This is a design decision for the local application.

15.13.1. Configuring OpenID

Seam uses the **`openid4java`** package, and requires four additional JARs to make use of Seam integration. These are **`htmlparser.jar`**, **`openid4java.jar`**, **`openxri-client.jar`** and **`openxri-syntax.jar`**.

OpenID processing requires the **`OpenIdPhaseListener`**, which should be added to your **`faces-config.xml`** file. The phase listener processes the callback from the OpenID provider, allowing re-entry into the local application.

```
<lifecycle>
  <phase-listener>
    org.jboss.seam.security.openid.OpenIdPhaseListener
  </phase-listener>
</lifecycle>
```

This configuration makes OpenID support available to your application. The OpenID support component, **`org.jboss.seam.security.openid.openid`**, is installed automatically if the **`openid4java`** classes are on the classpath.

15.13.2. Presenting an OpenIdLogin form

To initiate an OpenID log in, present a form to the user requesting the user's OpenID. The **`#{openid.id}`** value accepts the user's OpenID and the **`#{openid.login}`** action initiates an authentication request.

```
<h:form>
  <h:inputText value="#{openid.id}" />
  <h:commandButton action="#{openid.login}" value="OpenID Login"/>
</h:form>
```

When the user submits the login form, they are redirected to their OpenID provider. The user eventually returns to your application through the Seam pseudo-view **`/openid.xhtml`**, provided by the **`OpenIdPhaseListener`**. Your application can handle the OpenID response with a **`pages.xml`** navigation from that view, just as if the user had never left your application.

15.13.3. Logging in immediately

The simplest strategy is to simply log the user in immediately. The following navigation rule shows how to handle this using the **#{openid.loginImmediately()}** action.

```
<page view-id="/openid.xhtml">
  <navigation evaluate="#{openid.loginImmediately()}">
    <rule if-outcome="true">
      <redirect view-id="/main.xhtml">
        <message>OpenID login successful...</message>
      </redirect>
    </rule>
    <rule if-outcome="false">
      <redirect view-id="/main.xhtml">
        <message>OpenID login rejected...</message>
      </redirect>
    </rule>
  </navigation>
</page>
```

The **loginImmediately()** action checks whether the OpenID is valid. If it is valid, an **OpenIdPrincipal** is added to the identity component, and the user is marked as logged in (that is, **#{identity.loggedIn}** is marked **true**), and the **loginImmediately()** action returns **true**. If the OpenID is not validated, the method returns **false**, and the user re-enters the application un-authenticated. If the user's OpenID is valid, it will be accessible using the expression **#{openid.validatedId}** and **#{openid.valid}** will be true.

15.13.4. Deferring log in

If you do not want the user to be immediately logged in to your application, your navigation should check the **#{openid.valid}** property, and redirect the user to a local registration or processing page. Here, you can ask for more information and create a local user account, or present a CAPTCHA to avoid programmatic registrations. When your process is complete, you can log the user in by calling the **loginImmediately** method, either through EL (as shown previously), or direct interaction with the **org.jboss.seam.security.openid.OpenId** component. You can also write custom code to interact with the Seam identity component to create even more customized behavior.

15.13.5. Logging out

Logging out (forgetting an OpenID association) is done by calling **#{openid.logout}**. You can call this method directly if you are not using Seam Security. If you are using Seam Security, you should continue to use **#{identity.logout}** and install an event handler to capture the log out event, calling the OpenID log out method.

```
<event type="org.jboss.seam.security.loggedOut">
  <action execute="#{openid.logout}" />
</event>
```

It is important to include this, or the user will not be able to log in again in the same session.

CHAPTER 16. INTERNATIONALIZATION, LOCALIZATION AND THEMES

There are several stages required to internationalize and localize your application.

NOTE

Internationalization features are available only in the JSF context.

16.1. INTERNATIONALIZING YOUR APPLICATION

A Java EE 5 application involves many components, all of which must be configured properly to localize your application.

Before you begin, ensure that your database server and client use the correct character encoding for your locale. Normally, you will want UTF-8 encoding. (Setting the correct encoding falls outside the scope of this tutorial.)

16.1.1. Application server configuration

You will need to configure the Tomcat connector to ensure that the application server receives the request parameters from client requests in the correct encoding. Add the **`URIEncoding="UTF-8"`** attribute to the connector configuration in **`$JBOSS_HOME/server/$PROFILE/deploy/jboss-web.deployer/server.xml`**:

```
<Connector port="8080" URIEncoding="UTF-8"/>
```

Alternatively, you can tell JBoss Enterprise Application Platform that the correct encoding for the request parameters will be taken from the request:

```
<Connector port="8080" useBodyEncodingForURI="true"/>
```

16.1.2. Translated application strings

You will also need localized strings for all of the *messages* in your application (for example, field labels on your views). First, ensure that your resource bundle is encoded with the desired character encoding. ASCII is used by default. Although ASCII is enough for many languages, it does not provide characters for all languages.

Resource bundles must either be created in ASCII, or use Unicode escape codes to represent Unicode characters. Since you do not compile a property file to byte code, there is no way to tell JVM which character set to use. Therefore, you must use either ASCII characters or escape characters not in the ASCII character set. You can represent a Unicode character in any Java file with **`\uXXXX`**, where *XXXX* is the hexadecimal representation of the character.

You can write your translation of labels (Section 16.3, “Labels”) to your message resource bundle in the native coding. The **`native2ascii`** tool provided in the JDK lets you convert the contents of a file written in your native encoding into one that represents non-ASCII characters as Unicode escape sequences.

Usage of this tool is described here for Java 6. For example, to convert a file from UTF-8:

```
 $ native2ascii -encoding UTF-8 messages_cs.properties >
messages_cs_escaped.properties
```

16.1.3. Other encoding settings

We need to make sure that the view displays your localized data and messages in the correct character set, and that any data submitted uses the correct encoding.

Use the **`<f:view locale="cs_CZ"/>`** tag to set the display character encoding. (Note that this **`locale`** value sets JSF to use the Czech locale.) If you want to embed localized strings in the XML, you may want to change the XML document's encoding. To do so, alter the **`encoding`** attribute value in the XML declaration **`<?xml version="1.0" encoding="UTF-8"?>`**.

JSF/Facelets should submit any requests with the specified character encoding, but to ensure that requests that do not specify an encoding are submitted, you can force the request encoding using a servlet filter. Configure this in **`components.xml`**:

```
<web:character-encoding-filter encoding="UTF-8" override-client="true"
     url-pattern="*.seam" />
```

16.2. LOCALES

Each user log in session has an associated instance of **`java.util.Locale`**, which is available to the application as a component named **`locale`**. Under normal circumstances, setting the locale requires no special configuration. Seam delegates to JSF to determine the active locale as follows:

- If a locale is associated with the HTTP request (the browser locale), and that locale is in the list of supported locales from **`faces-config.xml`**, use that locale for the rest of the session.
- Otherwise, if a default locale was specified in the **`faces-config.xml`**, use that locale for the rest of the session.
- Otherwise, use the default locale of the server.

You *can* set the locale manually through the Seam configuration properties **`org.jboss.seam.international.localeSelector.language`**, **`org.jboss.seam.international.localeSelector.country`** and **`org.jboss.seam.international.localeSelector.variant`**, but there is no good reason to use this method over those described above.

It is useful to allow the user to set the locale manually via the application user interface. Seam provides built-in functionality to override the locale determined by the default algorithm. Do this by adding the following fragment to a form in your JSP or Facelets page:

```
<h:selectOneMenu value="#{localeSelector.language}">
  <f:selectItem itemLabel="English" itemValue="en"/>
  <f:selectItem itemLabel="Deutsch" itemValue="de"/>
  <f:selectItem itemLabel="Francais" itemValue="fr"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
                 value="#{messages['ChangeLanguage']}"/>
```

Or, if you want a list of all supported locales from **`faces-config.xml`**, use:

```
<h:selectOneMenu value="#{localeSelector.localeString}">
  <f:selectItems value="#{localeSelector.supportedLocales}"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
                 value="#{messages['ChangeLanguage']}"/>
```

When the user selects an item from the drop-down, then clicks the command button, the Seam and JSF locales will be overridden for the rest of the session.

You can configure the supported locales and the default locale of the server with the built-in **`org.jboss.seam.international.localeConfig`** component. First, declare an XML namespace for Seam's international package in the Seam component descriptor. Then, define the default locale and supported locales as follows:

```
<international:locale-config default-locale="fr_CA"
               supported-locales="en fr_CA fr_FR"/>
```

Remember that supported locales must have matching resource bundles. Next, define your language-specific labels.

16.3. LABELS

JSF supports the internationalization of user interface labels and descriptive text with the **`<f:loadBundle />`**. In Seam applications, you can either take this approach, or use the Seam **`messages`** component to display templated labels with embedded EL expressions.

16.3.1. Defining labels

Make your internationalized labels available with Seam's **`java.util.ResourceBundle`**, available to the application as a **`org.jboss.seam.core.resourceBundle`**. By default, Seam uses a resource bundle named **`messages`**, so you will need to define your labels in files named **`messages.properties`**, **`messages_en.properties`**, **`messages_en_AU.properties`**, etc. These files usually belong in the **`WEB-INF/classes`** directory.

So, in **`messages_en.properties`**:

```
Hello=Hello
```

And in **`messages_en_AU.properties`**:

```
Hello=G'day
```

You can select a different name for the resource bundle by setting the Seam configuration property named **`org.jboss.seam.core.resourceLoader.bundleNames`**. You can even specify a list of resource bundle names to be searched (depth first) for messages.

```
<core:resource-loader>
  <core:bundle-names>
    <value>mycompany_messages</value>
    <value>standard_messages</value>
  </core:bundle-names>
</core:resource-loader>
```

To define a message for one particular page, specify it in a resource bundle with the same name as the JSF view ID, with the leading `/` and trailing file extension removed. So, we could put our message in **`welcome/hello_en.properties`** if we only needed to display the message on **`/welcome/hello.jsp`**.

You can even specify an explicit bundle name in **`pages.xml`**:

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

Then we could use messages defined in **`HelloMessages.properties`** on **`/welcome/hello.jsp`**.

16.3.2. Displaying labels

If you define your labels with the Seam resource bundle, you can use them without having to type **`<f:loadBundle... />`** on each page. Instead, you can type:

```
<h:outputText value="#{messages['Hello']}"/>
```

or:

```
<h:outputText value="#{messages.Hello}"/>
```

Even better, the messages themselves may contain EL expressions:

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G'day, #{user.firstName}
```

You can even use the messages in your code:

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

16.3.3. Faces messages

The **`facesMessages`** component is a convenient way to display success or failure messages to the user. The functionality we just described also works for Faces messages:

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
  @In FacesMessages facesMessages;
  public String sayIt() {
    facesMessages.addFromResourceBundle("Hello");
  }
}
```

This will display **`Hello, Gavin King`** or **`G'day, Gavin`**, depending upon the user's locale.

16.4. TIMEZONES

There is also a session-scoped instance of **`java.util.Timezone`**, named **`org.jboss.seam.international.timezone`**, and a Seam component for changing the timezone named **`org.jboss.seam.international.timezoneSelector`**. By default, the timezone is the default timezone of the server. Unfortunately, the JSF specification assumes all dates and times are UTC, and displayed as UTC, unless a different timezone is explicitly specified with **`<f:convertDateTime>`**.

Seam overrides this behavior, and defaults all dates and times to the Seam timezone. In addition, Seam provides the **`<s:convertDateTime>`** tag, which always performs conversions in the Seam timezone.

Seam also provides a default date converter to convert a string value to a date. This saves you from having to specify a converter on input fields that capture dates. The pattern is selected according to the user's locale and the time zone is selected as described above.

16.5. THEMES

Seam applications are also very easily skinnable. The theme API is very similar to the localization API, but of course these two concerns are orthogonal, and some applications support both localization and themes.

First, configure the set of supported themes:

```
<theme:theme-selector cookie-enabled="true">
  <theme:available-themes>
    <value>default</value>
    <value>accessible</value>
    <value>printable</value>
  </theme:available-themes>
</theme:theme-selector>
```

The first theme listed is the default theme.

Themes are defined in a properties file with the same name as the theme. For example, the **`default`** theme is defined as a set of entries in **`default.properties`**, which might define:

```
css ../screen.css template /template.xhtml
```

The entries in a theme resource bundle are usually paths to CSS styles or images and names of Facelets templates (unlike localization resource bundles which are usually text).

Now we can use these entries in our JSP or Facelets pages. For example, to theme the stylesheet in a Facelets page:

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

Or, where the page definition resides in a subdirectory:

```
<link href="#{facesContext.externalContext.requestContextPath}#
{theme.css}"
      rel="stylesheet" type="text/css" />
```

Most powerfully, Facelets lets us theme the template used by a **`<ui:composition>`**:

```
<ui:composition 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"
    template="#{theme.template}">
```

Just like the locale selector, there is a built-in theme selector to allow the user to freely switch themes:

```
<h:selectOneMenu value="#{themeSelector.theme}">
  <f:selectItems value="#{themeSelector.themes}"/>
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme"/>
```

16.6. PERSISTING LOCALE AND THEME PREFERENCES VIA COOKIES

The locale selector, theme selector and timezone selector all support persistence of locale and theme preference to a cookie. Simply set the **`cookie-enabled`** property in **`components.xml`**:

```
<theme:theme-selector cookie-enabled="true">
  <theme:available-themes>
    <value>default</value>
    <value>accessible</value>
    <value>printable</value>
  </theme:available-themes>
</theme:theme-selector>

<international:locale-selector cookie-enabled="true"/>
```

CHAPTER 17. SEAM TEXT

Collaboration-oriented websites require a human-friendly markup language so that formatted text can be entered easily in forum posts, wiki pages, blogs, comments, etc. Seam provides the **<s:formattedText/>** control to display formatted text that conforms to the *Seam Text* language. Seam Text is implemented with an ANTLR-based parser. (Experience with ANTLR is not required.)

17.1. BASIC FORMATTING

Here is a simple example: **It's easy to make *emphasized*, |monospaced|, ~deleted~, super^scripted^ or _underlined_ text.**

If we display this using **<s:formattedText/>**, the following HTML will be produced:

```
<p>
  It's easy to make <i>emphasized</i>, <tt>monospaced</tt>,
  <del>deleted</del>, super<sup>scripted</sup> or
  <u>underlined</u> text.
</p>
```

We can use a blank line to indicate a new paragraph, and **+** to indicate a heading:

```
        +This is a big heading
        You /must/ have some text following a heading!

        ++This is a smaller heading
        This is the first paragraph. We can split it across multiple
        lines, but we must end it with a blank line.

        This is the second paragraph.
```

A simple new line is ignored — you need an additional blank line to wrap text into a new paragraph. This is the HTML that results:

```
<h1>This is a big heading</h1>
<p>
  You <i>must</i> have some text following a heading!
</p>

<h2>This is a smaller heading</h2>
<p>
  This is the first paragraph. We can split it across multiple
  lines, but we must end it with a blank line.
</p>

<p>
  This is the second paragraph.
</p>
```

The # character creates items in an ordered list. Unordered lists use the **=** character:

```
        An ordered list:
```

```
        #first item
        #second item
        #and even the /third/ item

        An unordered list:

        =an item
        =another item
```

```
<p>
  An ordered list:
</p>

<ol>
  <li>first item</li>
  <li>second item</li>
  <li>and even the <i>third</i> item</li>
</ol>

<p>
  An unordered list:
</p>

<ul>
  <li>an item</li>
  <li>another item</li>
</ul>
```

Quoted sections should be surrounded in double quotes:

```
        He said:

        "Hello, how are /you/?"

        She answered, "Fine, and you?"
```

```
<p>
  He said:
</p>

<q>Hi, how are
<i>you</i>?</q>

<p>
  She answered, <q>Fine, and you?</q>
</p>
```

17.2. ENTERING CODE AND TEXT WITH SPECIAL CHARACTERS

Special characters such as *****, **|** and **#**, and HTML characters such as **<**, **>** and **&** can be escaped with ****:

```
You can write down equations like 2\*3\=6 and HTML tags like \<body\>
using the escape character: \\.
```

```
<p>
  You can write down equations like 2*3=6 and HTML tags
  like &lt;body&gt; using the escape character: \.
</p>
```

And we can quote code blocks with backticks:

```
My code does not work:
  `for (int i=0; i<100; i--)
       {
       doSomething();
       }`
Any ideas?
```

```
<p>
  My code does not work:
</p>

<pre>for (int i=0; i&lt;100; i--)
{
    doSomething();
}</pre>

<p>
  Any ideas?
</p>
```

Since most monospace-formatted text is either code, or involves special characters, inline monospace formatting always escapes. So, you can write:

```
This is a |<tag attribute="value"/>| example.
```

without escaping any of the characters inside the monospace bars. This also means that inline monospace text cannot be formatted in any other way.

17.3. LINKS

You can create a link like so:

```
Go to the Seam website at [=>http://jboss.com/products/seam].
```

If you want to specify the link text:

```
Go to [the Seam website=>http://jboss.com/products/seam].
```

For advanced users, you can also customize the Seam Text parser to understand wikiword links written in this syntax.

17.4. ENTERING HTML

Text can include a certain limited subset of HTML. (The subset was selected to remain safe from cross-site scripting attacks.) This is useful for creating links:

```
You might want to link to
      <a href="http://jboss.com/products/seam">something cool</a>,
      or even include an image: <img src="/logo.jpg"/>
```

And for creating tables:

```
<table>
   <tr><td>First name:</td><td>Gavin</td></tr>
   <tr><td>Last name:</td><td>King</td></tr>
</table>
```

17.5. USING THE SEAMTEXTPARSER

The **<s:formattedText/>** JSF component uses the **org.jboss.seam.text.SeamTextParser** internally. You can use this class directly to implement your own text parsing, rendering, and HTML sanitation procedures. If you have a custom front-end interface for entering rich text, such as a JavaScript-based HTML editor, this can be useful for validating user input in order to defend against Cross-Site Scripting (XSS) attacks. You could also use it as a custom Wiki text-parsing and rendering engine.

The following example defines a custom text parser, which overrides the default HTML sanitizer:

```
public class MyTextParser extends SeamTextParser {

    public MyTextParser(String myText) {
        super(new SeamTextLexer(new StringReader(myText)));

        setSanitizer(
                    new DefaultSanitizer() {
                        @Override
                        public void validateHtmlElement(Token element)
throws SemanticException {
                            // TODO: I want to validate HTML elements
myself!
                        }
                    }
                    );
    }

    // Customizes rendering of Seam text links such as [Some
Text=&gt;http://example.com]
    @Override
    protected String linkTag(String descriptionText, String linkText) {
        return "&lt;a href=\"" + linkText + "\"&gt;My Custom Link: " +
            descriptionText + "&lt;/a&gt;";
    }

    // Renders a &lt;p&gt; or equivalent tag
    @Override
    protected String paragraphOpenTag() {
        return "&lt;p class=\"myCustomStyle\"&gt;";
    }

    public void parse() throws ANTLRException {
```

```
            startRule();
        }

    }
```

`linkTag()` and **`paragraphOpenTag()`** methods are two of the methods you can override in order to customize rendered output. These methods usually return **`String`** output. For further details, refer to the Java Documentation. The **`org.jboss.seam.text.SeamTextParser.DefaultSanitizer`** Java Documentation also contains more information about the HTML elements, attributes, and attribute values that are filtered by default.

CHAPTER 18. ITEXT PDF GENERATION

Seam now includes a component set for generating documents with iText. The primary focus of Seam's iText document support is for the generation of PDF documents, but Seam also offers basic support for RTF document generation.

18.1. USING PDF SUPPORT

iText support is provided by **jboss-seam-pdf.jar**. This **JAR** contains the iText JSF controls (which construct views that can render to PDF) and the DocumentStore component (which serves the rendered documents to the user). To include PDF support in your application, place **jboss-seam-pdf.jar** in your **WEB-INF/lib** directory along with the iText **JAR** file. No further configuration is required to use Seam's iText support.

The Seam iText module requires that Facelets be used as the view technology. Future versions of the library may also support the use of JSP. It also requires the use of the **seam-ui** package.

The **examples/itext** project contains an example of the PDF support in action. It demonstrates proper deployment packaging, and contains several examples demonstrating the key PDF-generation features currently supported.

18.1.1. Creating a document

<p:document>

Description

Documents are generated by Facelet XHTML files using tags in the **http://jboss.com/products/seam/pdf** namespace. Documents should always have the **document** tag at the root of the document. The **document** tag prepares Seam to generate a document into the DocumentStore and renders an HTML redirect to that stored content.

Attributes

type

The type of the document to be produced. Valid values are **PDF**, **RTF** and **HTML**. Seam defaults to PDF generation, and many features only work correctly when generating PDF documents.

pageSize

The size of the page to be generated. The most commonly used values are **LETTER** and **A4**. A full list of supported pages sizes can be found in the **com.lowagie.text.PageSize** class. Alternatively, pageSize can provide the width and height of the page directly. The value "612 792", for example, is equivalent to the LETTER page size.

orientation

The orientation of the page. Valid values are **portrait** and **landscape**. Landscape mode swaps the height and width page values.

margins

The left, right, top and bottom margin values.

marginMirroring

Indicates that margin settings should be reversed on alternating pages.

disposition

When generating PDFs in a web browser, this determines the HTTP **Content-Disposition** of the document. Valid values are **inline**, which indicates the document should be displayed in the browser window if possible, and **attachment**, which indicates that the document should be treated as a download. The default value is **inline**.

fileName

For attachments, this value overrides the downloaded file name.

Metadata Attributes

- **title**
- **subject**
- **keywords**
- **author**
- **creator**

Usage

```
<p:document
xmlns:p="http://jboss.com/products/seam/pdf">
  The document goes here.
</p:document>
```

18.1.2. Basic Text Elements

Seam provides special UI components for generating content suitable to PDF. **<p:image>** and **<p:paragraph>** tags form the foundations of simple documents. Tags like **<p:font>** provide style information.

<p:paragraph>

Description

For most purposes, text should be sectioned into paragraphs so that text fragments can be assigned a logical flow, format and style.

Attributes

- **firstLineIndent**
- **extraParagraphSpace**
- **leading**
- **multipliedLeading**
- **spacingBefore** — The blank space to be inserted before the element.
- **spacingAfter** — The blank space to be inserted after the element.
- **indentationLeft**
- **indentationRight**
- **keepTogether**

Usage

```
<p:paragraph alignment="justify">
  This is a simple document. It is not very fancy.
</p:paragraph>
```

<p:text>

Description

The **text** tag lets application data produce text fragments using normal JSF converter mechanisms. It is very similar to the **outputText** tag used when rendering HTML documents.

Attributes

- **value** — The value to be displayed. This is typically a value binding expression.

Usage

```
<p:paragraph>
  The item costs <p:text value="#{product.price}">
    <f:convertNumber type="currency"
                     currencySymbol="$"/>
  </p:text>
</p:paragraph>
```

<p:html>

Description

The **html** tag renders HTML content into the PDF.

Attributes

- **value** — The text to be displayed.

Usage

```
<p:html value="This is HTML with <b>some markup</b>"
/>
<p:html>
  <h1>This is more complex HTML</h1>
  <ul>
    <li>one</li>
    <li>two</li>
    <li>three</li>
  </ul>
</p:html>

<p:html>
    <s:formattedText value="*This* is |Seam Text| as
                            HTML.
        It's very^cool^." />
</p:html>
```

<p:font>

Description

The **font** tag defines the default font to be used for all text inside of it.

Attributes

- **name** — The font name, for example: **COURIER**, **HELVETICA**, **TIMES-ROMAN**, **SYMBOL** or **ZAPFDINGBATS**.
- **size** — The point size of the font.
- **style** — The font styles. Any combination of : **NORMAL**, **BOLD**, **ITALIC**, **OBLIQUE**, **UNDERLINE**, **LINE-THROUGH**
- **encoding** — The character set encoding.

Usage

```
<p:font name="courier" style="bold" size="24">
  <p:paragraph>My Title</p:paragraph>
</p:font>
```

<p:textcolumn>	*Description* **p:textcolumn** inserts a text column that can be used to control the flow of text. The most common case is to support right to left direction fonts. *Attributes* • **left** — The left bounds of the text column • **right** — The right bounds of the text column • **direction** — The run direction of the text in the column: **RTL**, **LTR**, **NO-BIDI**, **DEFAULT** *Usage* `<p:textcolumn left="400" right="600" direction="rtl"> <p:font name="/Library/Fonts/Arial Unicode.ttf" encoding="Identity-H" embedded="true">#{phrases.arabic}</p:font> </p:textcolumn>`
<p:newPage>	*Description* **p:newPage** inserts a page break. *Usage* `<p:newPage />`

<p:image>	*Description*
	
	p:image inserts an image into the document. Images can be loaded from the classpath or from the web application context using the **value** attribute.
	
	Resources can also be dynamically generated by application code. The **imageData** attribute can specify a value binding expression whose value is a **java.awt.Image** object.
	
	Attributes
	
	• **value** — A resource name or a method expression that binds to an application-generated image.
	• **rotation** — The rotation of the image in degrees.
	• **height** — The height of the image.
	• **width** — The width of the image.
	• **alignment** — The alignment of the image. (See Section 18.1.7.2, "Alignment Values" for possible values.)
	• **alt** — Alternative text representation for the image.
	• **indentationLeft**
	• **indentationRight**
	• **spacingBefore** — The blank space to be inserted before the element.
	• **spacingAfter** — The blank space to be inserted after the element.
	• **widthPercentage**
	• **initialRotation**
	• **dpi**
	• **scalePercent** — The scaling factor (as a percentage) to use for the image. This can be expressed as a single percentage value or as two percentage values representing separate x and y scaling percentages.
	• **wrap**
	• **underlying**
	
	Usage
	
	`<p:image value="/jboss.jpg" />`
	
	`<p:image value="#{images.chart}" />`

<p:anchor>	*Description*
	
	p:anchor defines clickable links from a document. It supports the following attributes:
	
	Attributes
	
	• **name** — The name of an in-document anchor destination.
	
	• **reference** — The destination the link refers to. Links to other points in the document should begin with a "#". For example, "#link1" to refer to an anchor position with a **name** of **link1**. To point to a resource outside the document, links must be a complete URL.
	
	Usage
	
	`<p:listItem>`
	`  <p:anchor reference="#reason1">Reason 1</p:anchor>`
	`</p:listItem>`
	`...`
	`<p:paragraph>`
	`  <p:anchor name="reason1">`
	`  It's the quickest way to get "rich"`
	`  </p:anchor>`
	`  ...`
	`</p:paragraph>`

18.1.3. Headers and Footers

<p:header>
 	
<p:footer>
	Description
	
	The **p:header** and **p:footer** components let you place header and footer text on each page of a generated document. Header and footer declarations should appear at the beginning of a document.
	
	Attributes
	
	• **alignment** — The alignment of the header/footer box section. (See Section 18.1.7.2, "Alignment Values" for alignment values.)
	
	• **backgroundColor** — The background color of the header/footer box. (See Section 18.1.7.1, "Color Values" for color values.)
	
	• **borderColor** — The border color of the header/footer box. Individual border sides can be set using **borderColorLeft**, **borderColorRight**, **borderColorTop** and **borderColorBottom**. (See Section 18.1.7.1, "Color Values" for color values.)
	
	• **borderWidth** — The width of the border. Individual border sides can be specified using **borderWidthLeft**, **borderWidthRight**, **borderWidthTop** and **borderWidthBottom**.
	
	Usage
	
	`<p:facet name="header">`
	`  <p:font size="12">`
	`  <p:footer borderWidthTop="1"`
	`borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`  </p:footer>`
	`  </p:font>`
	`</f:facet>`

<p:pageNumber>	*Description*
	
	The current page number can be placed inside a header or footer with the **p:pageNumber** tag. The page number tag can only be used in the context of a header or footer and can only be used once.
	
	Usage
	
	`<p:footer borderWidthTop="1" borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`</p:footer>`

18.1.4. Chapters and Sections

<p:chapter>
 	
<p:section>
	Description
	
	If the generated document follows a book/article structure, the **p:chapter** and **p:section** tags can be used to provide structure. Sections can only be used inside chapters, but they may be nested to any depth required. Most PDF viewers provide easy navigation between chapters and sections in a document.
	
	Attributes
	
	• **alignment** — The alignment of the header/footer box section. (See Section 18.1.7.2, "Alignment Values" for alignment values.)
	
	• **number** — The chapter number. Every chapter should be assigned a chapter number.
	
	• **numberDepth** — The depth of numbering for section. All sections are numbered relative to their surrounding chapters/sections. The fourth section of the first section of chapter three would be section 3.1.4, if displayed at the default number depth of **3**. To omit the chapter number, a number depth of **2** should be used — this would display the section number as 1.4.
	
	Usage
	
	`<p:document`
	`xmlns:p="http://jboss.com/products/seam/pdf"`
	`title="Hello">`
	`<p:chapter number="1">`
	
	`<p:title><p:paragraph>Hello</p:paragraph></p:title>`
	`<p:paragraph>Hello #{user.name}!</p:paragraph>`
	`</p:chapter>`
	
	`<p:chapter number="2">`
	`<p:title>`
	`<p:paragraph>`
	`Goodbye`
	`</p:paragraph>`
	`</p:title>`
	`<p:paragraph>Goodbye #{user.name}.</p:paragraph>`
	`</p:chapter>`
	
	`</p:document>`

<p:header>	*Description* Any chapter or section can contain a **p:title**. The title will be displayed next to the chapter or section number. The body of the title may contain raw text or may be a **p:paragraph**.

18.1.5. Lists

List structures can be displayed with the **p:list** and **p:listItem** tags. Lists may contain arbitrarily-nested sublists. List items may not be used outside of a list. The following document uses the **ui:repeat** tag to display a list of values retrieved from a Seam component.

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
```

```
    xmlns:ui="http://java.sun.com/jsf/facelets" title="Hello">

  <p:list style="numbered">
    <ui:repeat value="#{documents}" var="doc">
      <p:listItem>#{doc.name}</p:listItem>
    </ui:repeat>
  </p:list>

</p:document>
```

<p:list>	*Attributes* • **style** — The ordering/bulleting style of the list. One of: **NUMBERED**, **LETTERED**, **GREEK**, **ROMAN**, **ZAPFDINGBATS**, **ZAPFDINGBATS_NUMBER**. If no style is given, the list items are bulleted by default. • **listSymbol** — For bulleted lists, specifies the bullet symbol. • **indent** — The indentation level of the list. • **lowerCase** — For list styles using letters, indicates whether the letters should be lower case. • **charNumber** — For **ZAPFDINGBATS**, indicates the character code of the bullet character. • **numberType** — For **ZAPFDINGBATS_NUMBER**, indicates the numbering style. *Usage* `<p:list style="numbered"> <ui:repeat value="#{documents}" var="doc"> <p:listItem>#{doc.name}</p:listItem> </ui:repeat> </p:list>`

<p:listItem>	*Description* **p:listItem** supports the following attributes: *Attributes* • **alignment** — The alignment of the header/footer box section. (See Section 18.1.7.2, "Alignment Values" for alignment values.) • **alignment** — The alignment of the list item. (See Section 18.1.7.2, "Alignment Values" for possible values.) • **indentationLeft** — The left indentation amount. • **indentationRight** — The right indentation amount. • **listSymbol** — Overrides the default list symbol for this list item. *Usage* `...`

18.1.6. Tables

Table structures can be created using the **p:table** and **p:cell** tags. Unlike many table structures, there is no explicit row declaration. If a table has three columns, then every three cells will automatically form a row. Header and footer rows can be declared, and the headers and footers will be repeated in the event a table structure spans multiple pages.

<p:table>	*Description* **p:table** supports the following attributes. *Attributes* • **columns** — The number of columns (cells) that make up a table row. • **widths** — The relative widths of each column. There should be one value for each column. For example: **widths="2 1 1"** would indicate that there are three columns and the first column should be twice the size of the second and third column. • **headerRows** — The initial number of rows considered to be header and footer rows, which should be repeated if the table spans multiple pages. • **footerRows** — The number of rows considered to be footer rows. This value is subtracted from the **headerRows** value. If document has two rows which make up the header and one row that makes up the footer, **headerRows** should be set to **3** and **footerRows** should be set to **1**. • **widthPercentage** — The percentage of the page width spanned by the table. • **horizontalAlignment** — The horizontal alignment of the table. (See Section 18.1.7.2, "Alignment Values" for possible values.) • **skipFirstHeader**

- **runDirection**
- **lockedWidth**
- **splitRows**
- **spacingBefore** — The blank space to be inserted before the element.
- **spacingAfter** — The blank space to be inserted after the element.
- **extendLastRow**
- **headersInEvent**
- **splitLate**
- **keepTogether**

Usage

```
<p:table columns="3" headerRows="1">
  <p:cell>name</p:cell>
  <p:cell>owner</p:cell>
  <p:cell>size</p:cell>
  <ui:repeat value="#{documents}" var="doc">
    <p:cell>#{doc.name}</p:cell>
    <p:cell>#{doc.user.name}</p:cell>
    <p:cell>#{doc.size}</p:cell>
  </ui:repeat>
</p:table>
```

<p:cell>	*Description* **p:cell** supports the following attributes: *Attributes* • **colspan** — Cells can span more than one column by declaring a **colspan** greater than one. Cells cannot span multiple rows. • **horizontalAlignment** — The horizontal alignment of the cell. (See Section 18.1.7.2, “Alignment Values” for possible values.) • **verticalAlignment** — The vertical alignment of the cell. (See Section 18.1.7.2, “Alignment Values” for possible values.) • **padding** — Specify padding on a particular side using **paddingLeft**, **paddingRight**, **paddingTop** and **paddingBottom**. • **useBorderPadding** • **leading** • **multipliedLeading** • **indent** • **verticalAlignment** • **extraParagraphSpace** • **fixedHeight** • **noWrap** • **minimumHeight** • **followingIndent** • **rightIndent** • **spaceCharRatio** • **runDirection** • **arabicOptions** • **useAscender** • **grayFill** • **rotation** *Usage* `<p:cell>...</p:cell>`

18.1.7. Document Constants

This section documents some of the constants shared by attributes on multiple tags.

18.1.7.1. Color Values

Seam documents do not yet support a full color specification. Currently, only named colors are supported. They are: **white**, **gray**, **lightgray**, **darkgray**, **black**, **red**, **pink**, **yellow**, **green**, **magenta**, **cyan** and **blue**.

18.1.7.2. Alignment Values

Seam PDF supports the following horizontal alignment values: **left**, **right**, **center**, **justify** and **justifyall**. The vertical alignment values are **top**, **middle**, **bottom**, and **baseline**.

18.2. CHARTING

Charting support is also provided with **jboss-seam-pdf.jar**. Charts can be used in PDF documents, or as images in an HTML page. To use charting, you will need to add the JFreeChart library (**jfreechart.jar** and **jcommon.jar**) to the **WEB-INF/lib** directory. Three types of charts are currently supported: pie charts, bar charts and line charts.

<p:chart>	*Description* Displays a chart already created in Java by a Seam component. *Attributes* • **chart** -- The chart object to display • **height** -- The height fo the chart • **width** -- The width of the chart *Usage* `<p:chart chart="#{mycomponent.chart}" width="500" height="500" />`
<p:barchart>	*Description* Displays a bar chart. *Attributes* • **borderVisible** — Controls whether or not a border is displayed around the entire chart. • **borderPaint** — The color of the border, if visible. • **borderBackgroundPaint** — The default background color of the chart. • **borderStroke** • **domainAxisLabel** — The text label for the domain axis. • **domainLabelPosition** — The angle of the domain axis category labels. Valid values are **STANDARD**, **UP_45**, **UP_90**, **DOWN_45** and **DOWN_90**. The value can also be a positive or negative angle in radians. • **domainAxisPaint** — The color of the domain axis label.

- **`domainGridlinesVisible`**— Controls whether or not gridlines for the domain axis are shown on the chart.
- **`domainGridlinePaint`**— The color of the domain gridlines, if visible.
- **`domainGridlineStroke`** — The stroke style of the domain gridlines, if visible.
- **`height`** — The height of the chart.
- **`width`** — The width of the chart.
- **`is3D`** — A Boolean value indicating that the chart should be rendered in 3D instead of 2D.
- **`legend`** — A Boolean value indicating whether or not the chart should include a legend.
- **`legendItemPaint`**— The default color of the text labels in the legend.
- **`legendItemBackgoundPaint`**— The background color for the legend, if different from the chart background color.
- **`legendOutlinePaint`**— The color of the border around the legend.
- **`orientation`** — The orientation of the plot, either **`vertical`** (the default) or **`horizontal`**.
- **`plotBackgroundPaint`** — The color of the plot background.
- **`plotBackgroundAlpha`** — The alpha (transparency) level of the plot background. This should be a number between 0 (completely transparent) and 1 (completely opaque).
- **`plotForegroundAlpha`** — The alpha (transparency) level of the plot. This should be a number between 0 (completely transparent) and 1 (completely opaque).
- **`plotOutlinePaint`** — The color of the range gridlines, if visible.
- **`plotOutlineStroke`** — The stroke style of the range gridlines, if visible.
- **`rangeAxisLabel`** — The text label for the range axis.
- **`rangeAxisPaint`** — The color of the range axis label.
- **`rangeGridlinesVisible`** — Controls whether or not gridlines for the range axis are shown on the chart.
- **`rangeGridlinePaint`** — The color of the range gridlines, if visible.
- **`rangeGridlineStroke`** — The stroke style of the range gridlines, if visible.
- **`title`** — The chart title text.
- **`titlePaint`** — The color of the chart title text.
- **`titleBackgroundPaint`** — The background color around the chart title.
- **`width`** — The width of the chart.

Usage

```
<p:barchart title="Bar Chart" legend="true"
width="500" height="500">
  <p:series key="Last Year">
    <p:data columnKey="Joe" value="100" />
    <p:data columnKey="Bob" value="120" />
  </p:series>
  <p:series key="This Year">
    <p:data columnKey="Joe" value="125" />
    <p:data columnKey="Bob" value="115" />
  </p:series>
</p:barchart>
```

<p:linechart>

Description

Displays a line chart.

Attributes

- **borderVisible** — Controls whether or not a border is displayed around the entire chart.
- **borderPaint** — The color of the border, if visible.
- **borderBackgroundPaint** — The default background color of the chart.
- **borderStroke** —
- **domainAxisLabel** — The text label for the domain axis.
- **domainLabelPosition** — The angle of the domain axis category labels. Valid values are **STANDARD**, **UP_45**, **UP_90**, **DOWN_45** and **DOWN_90**. The value can also be a positive or negative angle in radians.
- **domainAxisPaint** — The color of the domain axis label.
- **domainGridlinesVisible**— Controls whether or not gridlines for the domain axis are shown on the chart.
- **domainGridlinePaint**— The color of the domain gridlines, if visible.
- **domainGridlineStroke** — The stroke style of the domain gridlines, if visible.
- **height** — The height of the chart.
- **width** — The width of the chart.
- **is3D** — A Boolean value indicating that the chart should be rendered in 3D instead of 2D.
- **legend** — A Boolean value indicating whether or not the chart should include a legend.
- **legendItemPaint** — The default color of the text labels in the legend.
- **legendItemBackgoundPaint** — The background color for the legend, if different from the chart background color.

- **`legendOutlinePaint`** — The color of the border around the legend.
- **`orientation`** — The orientation of the plot, either **`vertical`** (the default) or **`horizontal`**.
- **`plotBackgroundPaint`** — The color of the plot background.
- **`plotBackgroundAlpha`** — The alpha (transparency) level of the plot background. It should be a number between 0 (completely transparent) and 1 (completely opaque).
- **`plotForegroundAlpha`** — The alpha (transparency) level of the plot. It should be a number between 0 (completely transparent) and 1 (completely opaque).
- **`plotOutlinePaint`** — The color of the range gridlines, if visible.
- **`plotOutlineStroke`** — The stroke style of the range gridlines, if visible.
- **`rangeAxisLabel`** — The text label for the range axis.
- **`rangeAxisPaint`** — The color of the range axis label.
- **`rangeGridlinesVisible`** — Controls whether or not gridlines for the range axis are shown on the chart.
- **`rangeGridlinePaint`** — The color of the range gridlines, if visible.
- **`rangeGridlineStroke`** — The stroke style of the range gridlines, if visible.
- **`title`** — The chart title text.
- **`titlePaint`** — The color of the chart title text.
- **`titleBackgroundPaint`** — The background color around the chart title.
- **`width`** — The width of the chart.

Usage

```
<p:linechart title="Line Chart" width="500"
height="500">
  <p:series key="Prices">
    <p:data columnKey="2003" value="7.36" />
    <p:data columnKey="2004" value="11.50" />
    <p:data columnKey="2005" value="34.625" />
    <p:data columnKey="2006" value="76.30" />
    <p:data columnKey="2007" value="85.05" />
  </p:series>
</p:linechart>
```

`<p:piechart>`

Description

Displays a pie chart.

Attributes

- **`title`** — The chart title text.

- **`label`** — The default label text for pie sections.
- **`legend`** — A Boolean value indicating whether or not the chart should include a legend. Default value is **`true`**.
- **`is3D`** — A Boolean value indicating that the chart should be rendered in 3D instead of 2D.
- **`labelLinkMargin`** — The link margin for labels.
- **`labelLinkPaint`** — The paint used for the label linking lines.
- **`labelLinkStroke`** — The stroke used for the label linking lines.
- **`labelLinksVisible`** — A flag that controls whether or not the label links are drawn.
- **`labelOutlinePaint`** — The paint used to draw the outline of the section labels.
- **`labelOutlineStroke`** — The stroke used to draw the outline of the section labels.
- **`labelShadowPaint`** — The paint used to draw the shadow for the section labels.
- **`labelPaint`** — The color used to draw the section labels.
- **`labelGap`** — The gap between the labels and the plot, as a percentage of the plot width.
- **`labelBackgroundPaint`** — The color used to draw the background of the section labels. If this is null, the background is not filled.
- **`startAngle`** — The starting angle of the first section.
- **`circular`** — A Boolean value indicating that the chart should be drawn as a circle. If **`false`**, the chart is drawn as an ellipse. The default is **`true`**.
- **`direction`** — The direction in which the pie sections are drawn. One of: **`clockwise`** or **`anticlockwise`**. The default is **`clockwise`**.
- **`sectionOutlinePaint`** — The outline paint for all sections.
- **`sectionOutlineStroke`** — The outline stroke for all sections.
- **`sectionOutlinesVisible`** — Indicates whether an outline is drawn for each section in the plot.
- **`baseSectionOutlinePaint`** — The base section outline paint.
- **`baseSectionPaint`** — The base section paint.
- **`baseSectionOutlineStroke`** — The base section outline stroke.

Usage

```
<p:piechart title="Pie Chart" circular="false"
            direction="anticlockwise" startAngle="30"
            labelGap="0.1" labelLinkPaint="red">
  <p:series key="Prices">
    <p:data key="2003" columnKey="2003" value="7.36"
```

```
/>
    <p:data key="2004" columnKey="2004" value="11.50"
/>
    <p:data key="2005" columnKey="2005" value="34.625"
/>
    <p:data key="2006" columnKey="2006" value="76.30"
/>
    <p:data key="2007" columnKey="2007" value="85.05"
/>
  </p:series>
</p:piechart>
```

<p:series>

Description

Category data can be broken down into series. The series tag is used to categorize a data set with a series and apply styling to the entire series.

Attributes

- **key** — The series name.
- **seriesPaint** — The color of each item in the series.
- **seriesOutlinePaint** — The outline color for each item in the series.
- **seriesOutlineStroke** — The stroke used to draw each item in the series.
- **seriesVisible** — A Boolean indicating if the series should be displayed.
- **seriesVisibleInLegend** — A Boolean indicating whether the series should be listed in the legend.

Usage

```
<p:series key="data1">
  <ui:repeat value="#{data.pieData1}" var="item">
    <p:data columnKey="#{item.name}"
            value="#{item.value}" />
  </ui:repeat>
</p:series>
```

<p:data>

Description

The data tag describes each data point to be displayed in the graph.

Attributes

- **key** — The name of the data item.
- **series** — The series name, when not embedded inside a **<p:series>**.
- **value** — The numeric data value.
- **explodedPercent** — For pie charts, indicates how exploded from the pie a piece is.
- **sectionOutlinePaint** — For bar charts, the color of the section outline.
- **sectionOutlineStroke** — For bar charts, the stroke type for the section outline.
- **sectionPaint** — For bar charts, the color of the section.

Usage

```
<p:data key="foo" value="20" sectionPaint="#111111"
explodedPercent=".2" />
<p:data key="bar" value="30" sectionPaint="#333333" />
<p:data key="baz" value="40" sectionPaint="#555555"
   sectionOutlineStroke="my-dot-style" />
```

<p:color>

Description

The color component declares a color or gradient for filled shapes.

Attributes

- **color** — The color value. For gradient colors, this indicates the starting color. See Section 18.1.7.1, “Color Values” for color values.
- **color2** — For gradient colors, this is the color that ends the gradient.
- **point** — The coordinates that mark the beginning of the gradient color.
- **point2** — The coordinates that mark the end of the gradient color.

Usage

```
<p:color id="foo" color="#0ff00f"/>
<p:color id="bar" color="#ff00ff" color2="#00ff00"
   point="50 50" point2="300 300"/>
```

<p:stroke>	*Description* Describes a stroke used to draw lines in a chart. *Attributes* • **width** — The width of the stroke. • **cap** — The line cap type. Valid values are **butt**, **round** and **square** • **join** — The line join type. Valid values are **miter**, **round** and **bevel** • **miterLimit** — For miter joins, this value is the limit of the size of the join. • **dash** — The dash value sets the dash pattern used to draw the line. Use space-separated integers to indicate the length of each alternating drawn and undrawn segment. • **dashPhase** — The dash phase indicates the point in the dash pattern that corresponds to the beginning of the stroke. *Usage* `<p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" />`

18.3. BAR CODES

Seam can use iText to generate barcodes in a wide variety of formats. These barcodes can be embedded in a PDF document or displayed as an image on a web page. However, barcodes cannot currently display barcode text when used with HTML images.

<p:barCode>	*Description* Displays a barcode image. *Attributes* • **type** — A barcode type supported by iText. Valid values include: **EAN13**, **EAN8**, **UPCA**, **UPCE**, **SUPP2**, **SUPP5**, **POSTNET**, **PLANET**, **CODE128**, **CODE128_UCC**, **CODE128_RAW** and **CODABAR**. • **code** — The value to be encoded by the barcode. • **xpos** — For PDFs, the absolute **x** position of the barcode on the page. • **ypos** — For PDFs, the absolute **y** position of the barcode on the page. • **rotDegrees** — For PDFs, the rotation factor of the barcode in degrees. • **barHeight** — The height of the bars in the barcode. • **minBarWidth** — The minimum bar width. • **barMultiplier** — The bar multiplier for wide bars or the distance between bars for **POSTNET** and **PLANET** code. • **barColor** — The color the bars should be drawn in. • **textColor** — The color of any text on the barcode. • **textSize** — The size of any text on the barcode. • **altText** — The **alt** text for HTML image links. *Usage* `<p:barCode type="code128" barHeight="80" textSize="20" code="(10)45566(17)040301" codeType="code128_ucc" altText="My BarCode" />`

18.4. FILL-IN-FORMS

If you have a complex, pre-generated PDF with named fields, you can easily populate it with values from your application and present it to the user.

<p:form>	*Description* Defines a form template to populate. *Attributes* • **URL** — A URL that points to the PDF file to use as a template. If the value contains no protocol (**://**), the file is read locally. • **filename** — The filename to use for the generated PDF file. • **exportKey** — If set, no redirect will occur when the generated PDF file is placed in a DocumentData object under the specified key in the event context.

<p:field>	*Description* Connects a field name to its value. *Attributes* • **name** — The name of the field. • **value** — The value of the field. • **readOnly** — Whether the field is read-only. The default is **true**.

```
<p:form
   xmlns:p="http://jboss.com/products/seam/pdf"
   URL="http://localhost/Concept/form.pdf">
  <p:field name="person.name" value="Me, myself and I"/>
</p:form>
```

18.5. RENDERING SWING/AWT COMPONENTS

Seam now provides experimental support to render Swing components into PDF images. Some Swing look and feel supports, specifically those that use native widgets, will not render correctly.

<p:swing>	*Description* Renders a Swing component into a PDF document. *Attributes* • **width** — The width of the component to be rendered. • **height** — The height of the component to be rendered. • **component** — An expression whose value is a Swing or AWT component. *Usage* `<p:swing width="310" height="120" component="#{aButton}" />`

18.6. CONFIGURING ITEXT

Document generation itself requires no additional configuration, but some minor configuration can make your application more user-friendly.

The default implementation serves PDF documents from a generic URL, **/seam-doc.seam**. Many users prefer to see a URL that contains the actual document name and extension — **/myDocument.pdf**, for example. To serve fully named files, the **DocumentStoreServlet** must contain mappings for each document type:

```
<servlet>
  <servlet-name>Document Store Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.document.DocumentStoreServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Document Store Servlet</servlet-name>
  <url-pattern>DOCUMENT_TYPE</url-pattern>
</servlet-mapping>
```

DOCUMENT_TYPE can take on the following values:

- ***.pdf**
- ***.xls**
- ***.csv**

To include multiple document types, add a **<servlet-mapping>** element, with **<servlet-name>** and **<url-pattern>** sub-elements, for each desired document type.

The **use-extensions** option instructs the DocumentStore component to generate URLs with the correct filename extension for the generated document type.

```
<components xmlns="http://jboss.com/products/seam/components"
```

```
            xmlns:document="http://jboss.com/products/seam/document"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
                http://jboss.com/products/seam/document
                http://jboss.com/products/seam/document-2.2.xsd
                http://jboss.com/products/seam/components
                http://jboss.com/products/seam/components-2.2.xsd">
  <document:document-store use-extensions="true"/>
</components>
```

The DocumentStore holds documents in conversation scope, so documents will expire when the conversation ends. At that point, references to the document will be invalid. Specify a default view to show when a document does not exist by editing the **`error-page`** property of **`documentStore`**.

```
<document:document-store use-extensions="true"
          error-page="/documentMissing.seam" />
```

18.7. FURTHER DOCUMENTATION

For further information on iText, see:

- iText Home Page
- iText in Action

CHAPTER 19. THE MICROSOFT® EXCEL® SPREADSHEET APPLICATION

Seam can generate Microsoft® **Excel®** spreadsheets through the JExcelAPI library. The document generated is compatible with **Microsoft Excel** versions 95, 97, 2000, XP, and 2003. At present, only a limited subset of the library functionality is available. Refer to the JExcelAPI documentation for more information about limitations and capabilities.

19.1. MICROSOFT EXCEL SUPPORT

To include **Microsoft Excel** support in your application, you must include **`jboss-seam-excel.jar`** and **`jxl.jar`** in your **`WEB-INF/lib`** directory. **`jboss-seam-excel.jar`** contains the **Microsoft Excel** JSF controls used to construct views for document rendering, and the DocumentStore component, which serves the rendered document to the user. You will also need to configure the DocumentStore Servlet in your **`web.xml`** file. The **Microsoft Excel** Seam module requires the **`seam-ui`** package, and that Facelets be used as the view technology.

You can see an example of **Microsoft Excel** support in action in the **`examples/excel`** project. This demonstrates the exposed functionality of the support, as well as correct deployment packaging.

You can easily customize the module to support other kinds of **Microsoft Excel** spreadsheet. Implement the **`ExcelWorkbook`** interface and register the following in **`components.xml`**:

```
<excel:excelFactory>
  <property name="implementations">
    <key>myExcelExporter</key>
    <value>my.excel.exporter.ExcelExport</value>
  </property>
</excel:excelFactory>
```

Register the **Microsoft Excel** namespace in the components tag like so:

```
xmlns:excel="http://jboss.com/products/seam/excel"
```

Then set the UIWorkbook type to **`myExcelExporter`** to use your own preferred exporter. The default here is **`jxl`**, but you can also use CSV with the **`csv`** type.

See Section 18.6, “Configuring iText” for information about how to configure the document servlet for serving documents with an .xls extension.

If you have trouble accessing the generated file under Microsoft® **Internet Explorer®**, especially with HTTPS, check that your **`web.xml`** or browser security constraints (see http://www.nwnetworks.com/iezones.htm/) are not too strict.

19.2. CREATING A SIMPLE WORKBOOK

The worksheet support is used like a **`<h:dataTable>`**, and can be bound to a **`List`**, **`Set`**, **`Map`**, **`Array`** or **`DataModel`**.

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet>
```

```
    <e:cell column="0" row="0" value="Hello world!"/>
  </e:worksheet>
</e:workbook>
```

The following is a more common use case:

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet value="#{data}" var="item">
    <e:column>
      <e:cell value="#{item.value}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

The top-level **workbook** element serves as the container, and has no attributes. The child element, **worksheet**, has two attributes: **value="#{data}"** is the EL-binding to the data, and**var="item"** is the name of the current item. The worksheet contains a single **column**. Within this is the **cell**, which is the final bind to the data in the currently iterated item.

Now you can bind your data into spreadsheets.

19.3. WORKBOOKS

Workbooks are the top-level parents of worksheets and stylesheet links.

<e:workbook>	*Attributes* • **type** — Defines the export model. The value is a string and can be either **jxl** or **csv**. The default is **jxl**. • **templateURI** — A template that forms the basis of the workbook. The value is a string (URI). • **arrayGrowSize** — The amount of memory (in bytes) by which the workbook data storage space should be increased. If your process reads many small workbooks inside a web application server, you may need to reduce the default size. The default value is 1 MB. • **autoFilterDisabled** — A Boolean value determining whether autofiltering is disabled. • **cellValidationDisabled** — A Boolean value determining whether cell validation is ignored. • **characterSet** — The character set used to read the spreadsheet. Has no effect on the spreadsheet being written. The value is a string (character set encoding). • **drawingsDisabled** — A Boolean value determining whether drawings are disabled. • **excelDisplayLanguage** — The language that the generated file will display in. The value is a string (two character ISO 3166 country code).

- **`excelRegionalSettings`** — The regional settings for the generated file. The value is a string (two character ISO 3166 country code).
- **`formulaAdjust`** — A Boolean determining whether formulas are adjusted.
- **`gcDisabled`** — A Boolean determining whether garbage collection is disabled.
- **`ignoreBlanks`** — A Boolean value determining whether blanks are ignored.
- **`initialFileSize`** — The initial amount of memory (in bytes) allocated to workbook data storage when reading a worksheet. If your process reads many small workbooks inside a web application server, you may need to reduce the default size. The default value is 5 MB.
- **`locale`** — The locale JExcelAPI uses to generate the spreadsheet. This value has no effect on the language or region of the generated file. The value is a string.
- **`mergedCellCheckingDisabled`** — A Boolean determining whether merged cell checking is disabled.
- **`namesDisabled`** — A Boolean determining whether name handling is disabled.
- **`propertySets`** — A Boolean determining whether property sets (such as macros) are copied with the workbook. If this feature is enabled, the JXL process will use more memory.
- **`rationalization`** — A Boolean determining whether cell formats are rationalized before the sheet is written. Defaults to **`true`**.
- **`supressWarnings`** — A Boolean determining whether warnings are suppressed. Depending on the type of logger used, this sets the warning behavior across the JVM.
- **`temporaryFileDuringWriteDirectory`** — A string value containing the target directory for temporary files. Used in conjunction with **`useTemporaryFileDuringWrite`**. If set to **`NULL`**, the default temporary directory is used instead.
- **`useTemporaryFileDuringWrite`** — A Boolean determining whether a temporary file is used during workbook generation. If not set, the workbook will be generated entirely in memory. Setting this flag involves an assessment of the trade-offs between memory usage and performance.
- **`workbookProtected`** — A Boolean determining whether the workbook is protected.
- **`filename`** — A string value to be used as the download's filename. If you map the DocumentServlet to some pattern, its file extension must match.
- **`exportKey`** — A key to store event-scoped data in a DocumentData object. If used, there is no redirection.

Child elements

- **`<e:link/>`** — Zero or more stylesheet links. (See Section 19.14.1, “Stylesheet links”.)

- **`<e:worksheet/>`** — Zero or more worksheets. (See Section 19.4, "Worksheets".)

Facets

- **none**

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

This defines a workbook with a worksheet and a greeting at cell A1.

19.4. WORKSHEETS

Worksheets are the children of workbooks and the parent of columns and worksheet commands. They can also contain explicitly placed cells, formulas, images and hyperlinks. They are the pages that make up the workbook.

`<e:worksheet>`

- **`value`** — An EL-expression string to the backing data. The target of this expression is examined for an Iterable. If the target is a Map, the iteration is done over the Map.Entry entrySet(), so use a .key or .value to target in your references.
- **`var`** — The current row iterator variable name to be referenced in cell value attributes. The value is a string.
- **`name`** — The name of the worksheet. The value is a string. Defaults to **`Sheet<replaceable>#</replaceable>`** where # is the worksheet index. If the given worksheet name exists, that sheet is selected. This can be used to merge several data sets into a single worksheet by defining each worksheet with the same name — use **`startRow`** and **`startCol`** to make sure they do not occupy the same space.
- **`startRow`** — A number value that defines the starting row for the data. This is used to position data from places other than the upper-left corner. This is particularly useful when using multiple data sets for a single worksheet. The default value is **`0`**.
- **`startColumn`** — A number value that defines the starting column for the data. This is used to position data from places other than the upper-left corner. This is particularly useful when using multiple data sets for a single worksheet. The default value is **`0`**.
- **`automaticFormulaCalculation`** — A Boolean determining whether formulas are calculated automatically.
- **`bottomMargin`** — A number value determining the bottom margin in inches.
- **`copies`** — A number value determining the number of copies.
- **`defaultColumnWidth`** — A number value defining the default column width, in characters * 256.

- **`defaultRowHeight`** — A number value defining the default row height, in 1/20ths of a point.
- **`displayZeroValues`** — A Boolean determining whether zero values are displayed.
- **`fitHeight`** — A number value defining the number of pages vertically that this sheet will print into.
- **`fitToPages`** — A Boolean determining whether printing fits to pages.
- **`fitWidth`** — A number value defining the number of pages across that this sheet will print into.
- **`footerMargin`** — A number value defining the margin for any page footer in inches.
- **`headerMargin`** — A number value defining the margin for any page header in inches.
- **`hidden`** — A Boolean determining whether the worksheet is hidden.
- **`horizontalCentre`** — A Boolean determining whether the worksheet is centred horizontally.
- **`horizontalFreeze`** — A number value defining the column at which the pane is frozen horizontally.
- **`horizontalPrintResolution`** — A number value defining the horizontal print resolution.
- **`leftMargin`** — A number value defining the left margin in inches.
- **`normalMagnification`** — A number value defining the normal magnification factor as a percentage. This is not the zoom or scale factor.
- **`orientation`** — A string value that determines the paper orientation when this sheet is printed. Can be either **`landscape`** or **`portrait`**.
- **`pageBreakPreviewMagnification`** — A number value defining the page break preview magnification factor as a percentage.
- **`pageBreakPreviewMode`** — A Boolean determining whether the page is shown in preview mode.
- **`pageStart`** — A number value defining the page number at which to commence printing.
- **`paperSize`** — A string value determining the paper size to be used when printing. Possible values are **`a4`**, **`a3`**, **`letter`**, **`legal`**, etc. For a full list, see jxl.format.PaperSize.
- **`password`** — A string value determining the password for this sheet.
- **`passwordHash`** — A string value determining the password hash. This is used only when copying sheets.
- **`printGridLines`** — A Boolean determining whether grid lines are printed.
- **`printHeaders`** — A Boolean determining whether headers are printed.

- **`sheetProtected`** — A Boolean determining whether the sheet is read-only.
- **`recalculateFormulasBeforeSave`** — A Boolean determining whether formulas are recalculated when the sheet is saved. The default value is **`false`**.
- **`rightMargin`** — A number value defining the right margin in inches.
- **`scaleFactor`** — A number value defining the scale factor (as a percentage) used when this sheet is printed.
- **`selected`** — A Boolean value determining whether the sheet is selected automatically when the workbook opens.
- **`showGridLines`** — A Boolean determining whether grid lines are shown.
- **`topMargin`** — A number value defining the top margin in inches.
- **`verticalCentre`** — A Boolean determining whether the sheet is vertically centred.
- **`verticalFreeze`** — A number value determining the row at which the pane is frozen vertically.
- **`verticalPrintResolution`** — A number value determining the vertical print resolution.
- **`zoomFactor`** — A number value determining the zoom factor. This relates to on-screen view, and should not be confused with the scale factor.

Child elements

- **`<e:printArea/>`** — Zero or more print area definitions. (See Section 19.11, “Print areas and titles”.)
- **`<e:printTitle/>`** — Zero or more print title definitions. (See Section 19.11, “Print areas and titles”.)
- **`<e:headerFooter/>`** — Zero or more header/footer definitions. (See Section 19.10, “Headers and footers”.)
- Zero or more worksheet commands. (See Section 19.12, “Worksheet Commands”.)

Facets

- **`header`**— Contents placed at the top of the data block, above the column headers (if any).
- **`footer`**— Contents placed at the bottom of the data block, below the column footers (if any).

```
<e:workbook>
  <e:worksheet name="foo" startColumn="1" startRow="1">
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
```

```
    </e:column>
  </e:worksheet>
<e:workbook>
```

This defines a worksheet with the name "foo", starting at B2.

19.5. COLUMNS

Columns are the children of worksheets and the parents of cells, images, formulas and hyperlinks. They control the iteration of the worksheet data. See Section 19.14.5, “Column settings” for formatting.

<e:column>	*Attributes* • **none** *Child elements* • **<e:cell/>** — Zero or more cells. (See Section 19.6, “Cells”.) • **<e:formula/>** — Zero or more formulas. (See Section 19.7, “Formulas”.) • **<e:image/>** — Zero or more images. (See Section 19.8, “Images”.) • **<e:hyperLink/>** — Zero or more hyperlinks (see Section 19.9, “Hyperlinks”). *Facets* • **header** — This facet can/will contain one **<e:cell>**, **<e:formula>**, **<e:image>** or **<e:hyperLink>**, which will be used as header for the column. • **footer** — This facet can/will contain one **<e:cell>**, **<e:formula>**, **<e:image>** or **<e:hyperLink>**, which will be used as footer for the column.

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
<e:workbook>
```

This defines a column with a header and an iterated output.

19.6. CELLS

Cells are nested within columns (for iteration) or inside worksheets (for direct placement using the **column** and **row** attributes) and are responsible for outputting the value, usually through an EL-expression involving the **var** attribute of the datatable. See Section 19.14.6, “Cell settings”.

<e:cell>	*Attributes* • **column** — A number value denoting the column that the cell belongs to. The default is the internal counter. Note that the value is 0-based. • **row** — A number value denoting the row where to place the cell. The default is the internal counter. Note that the value is 0-based. • **value** — A string defining the display value. Usually an EL-expression referencing the var-attribute of the containing datatable. • **comment** — A string value defining a comment attached to the cell. • **commentHeight** — The comment height in pixels. • **commentWidth** — The comment width in pixels. *Child elements* • Zero or more validation conditions. (See Section 19.6.1, “Validation”.) *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

This defines a column with a header and an iterated output.

19.6.1. Validation

Validations are nested inside cells or formulas. They add constraints to cell data.

<e:numericValidation>	*Attributes* • **value** — A number value denoting the limit (or lower limit where applicable) of the validation. • **value2** — A number value denoting the upper limit (where applicable) of the validation. • **condition** — A string value defining the validation condition. ◦ **equal** — requires the cell value to match the one defined in the value-attribute. ◦ **greater_equal** — requires the cell value to be greater than or equal to the value defined in the value-attribute. ◦ **less_equal** — requires the cell value to be less than or equal to the value defined in the value-attribute. ◦ **less_than** — requires the cell value to be less than the value defined in the value-attribute. ◦ **not_equal** — requires the cell value to not match the one defined in the value-attribute. ◦ **between** — requires the cell value to be between the values defined in the value and value2 attributes. ◦ **not_between** — requires the cell value not to be between the values defined in the value- and value2 attributes. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.age">
        <e:numericValidation condition="between" value="4" value2="18"/>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

This adds numeric validation to a cell, specifying that the value must be between 4 and 18.

<e:rangeValidation>	*Attributes* • **startColumn** — A number value denoting the first column to validate against. • **startRow** — A number value denoting the first row to validate against. • **endColumn** — A number value denoting the last column to validate against. • **endRow** — A number value denoting the last row to validate against. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.position">
        <e:rangeValidation startColumn="0" startRow="0" endColumn="0"
           endRow="10"/>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

This adds validation to a cell, specifying that the value must exist within the values specified in range A1:A10.

<e:listValidation>	*Attributes* • **none** *Child elements* • **Zero or more list validation items.** *Facets* • **none**

e:listValidation is a just a container for holding multiple e:listValidationItem tags.

<e:listValidationItem>	*Attributes* • **value** — A value to validate against. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.position">
        <e:listValidation>
          <e:listValidationItem value="manager"/>
          <e:listValidationItem value="employee"/>
        </e:listValidation>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

This adds validation to a cell, specifying that the value must be "manager" or "employee".

19.6.2. Format masks

Format masks are defined in the **mask** attribute in a cell or formula. There are two types of format masks: one for numbers, and one for dates.

19.6.2.1. Number masks

When a format mask is encountered, a check is executed to see if the mask follows an internal form, for example, **format1**, **accounting_float**, etc. (See jxl.write.NumberFormats.)

If the mask is not part of the internal list, it is treated as a custom mask (for example, **0.00**), and automatically converted to the closest match. (See java.text.DecimalFormat.)

19.6.2.2. Date masks

When a format mask is encountered, a check is executed to see if the mask follows an internal form, for example, **format1**, **format2**, etc. (See jxl.write.DecimalFormats.)

If the mask is not part of the internal list, it is treated as a custom mask (for example, **dd.MM.yyyy**), and automatically converted to the closest match. (See java. text.DateFormat.)

19.7. FORMULAS

Formulas are nested within columns (for iteration) or inside worksheets (for direct placement using the **column** and **row** attributes), and add calculations or functions to ranges of cells. They are essentially

cells, see Section 19.6, “Cells” for available attributes. They can apply templates and have their own font definitions, etc., just as normal cells can.

The formula of the cell is placed in the **value** attribute as a normal **Microsoft Excel** notation. When doing cross-sheet formulas, the worksheets must exist before referencing a formula against them. The value is a string.

```
<e:workbook>
  <e:worksheet name="fooSheet">
    <e:cell column="0" row="0" value="1"/>
  </e:worksheet>
  <e:worksheet name="barSheet">
  <e:cell column="0" row="0" value="2"/>
    <e:formula column="0" row="1" value="fooSheet!A1+barSheet1!A1">
      <e:font fontSize="12"/>
    </e:formula>
  </e:worksheet>
</e:workbook>
```

This defines a formula in B2 summing cells A1 in worksheets FooSheet and BarSheet.

19.8. IMAGES

Images are nested within columns (for iteration) or inside worksheets (for direct placement using the **startColumn/startRow** and **rowSpan/columnSpan** attributes). Span tags are optional, and the image will be inserted without resizing if they are omitted.

<e:image>	*Attributes* • **startColumn** — A number value denoting the column in which the image starts. The default is the internal counter. The number value is 0-based. • **startRow** — A number value denoting the row in which the image starts. The default is the internal counter. The number value is 0-based. • **columnSpan** — A float value denoting the column span of the image. The default uses the default width of the image. • **rowSpan** — A float value denoting the row span of the image. The default uses the default height of the image. • **URI** — A string value denoting the URI to the image. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
```

```
  <e:worksheet>
    <e:image startRow="0" startColumn="0" rowSpan="4" columnSpan="4"
       URI="http://foo.org/logo.jpg"/>
  </e:worksheet>
</e:workbook>
```

This defines an image in A1:E5 based on the given data.

19.9. HYPERLINKS

Hyperlinks are nested within columns (for iteration) or inside worksheets (for direct placement using the **`startColumn/startRow`** and **`endColumn/endRow`** attributes). They add link navigation to URIs.

`<e:hyperlink>`	*Attributes* • **`startColumn`** — A number value denoting the column in which the hyperlink starts. The default is the internal counter. The number value is 0-based. • **`startRow`** — A number value denoting the row in which the hyperlink starts. The default is the internal counter. The number value is 0-based. • **`endColumn`** — A number value denoting the column in which the hyperlink ends. The default is the internal counter. The number value is 0-based. • **`endRow`** — A number value denoting the row in which the hyperlink ends. The default is the internal counter. The number value is 0-based. • **`URL`** — A string value denoting the URL to link. • **`description`** — A string value describing the link. string. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:hyperLink startRow="0" startColumn="0" endRow="4" endColumn="4"
       URL="http://seamframework.org" description="The Seam Framework"/>
  </e:worksheet>
</e:workbook>
```

This defines a described hyperlink pointing to Seam Framework in the area A1:E5.

19.10. HEADERS AND FOOTERS

Headers and footers are children of worksheets, and contain facets, which contain strings to be parsed as commands.

<e:header>	*Attributes* • **none** *Child elements* • **none** *Facets* • **left** — The contents of the left header part. • **center** — The contents of the central header part. • **right** — The contents of the right header part.

<e:footer>	*Attributes* • **none** *Child elements* • **none** *Facets* • **left** — The contents of the left footer part. • **center** — The contents of the central footer part. • **right** — The contents of the right footer part.

Because facets contain string values, they can contain various #-delimited commands, like the following:

#date#	Inserts the current date.
#page_number#	Inserts the current page number.
#time#	Inserts the current time.
#total_pages#	Inserts the total page count.
#worksheet_name#	Inserts the worksheet name.
#workbook_name#	Inserts the workbook name.
#bold#	Toggles bold font. One use turns bold text on; a second use turns bold text off.
#italics#	Toggles italic font. One use turns italic text on; a second use turns italic text off.
#underline#	Toggles underlined font. One use turns underlined text on; a second use turns underlined text off.

#double_underline#	Toggles double-underlined font. One use turns double-underlying text on; a second use turns double-underlined text off.
#outline#	Toggles outlined font. One use turns outlined text on; a second use turns outlined text off.
#shadow#	Toggles shadowed font. One use turns shadowed text on; a second use turns shadowed text off.
#strikethrough#	Toggles struck-through font. One use turns struck-through text on; a second use turns struck-through text off.
#subscript#	Toggles subscript font. One use turns subscript text on; a second use turns subscript text off.
#superscript#	Toggles superscript font. One use turns superscript text on; a second use turns superscript text off.
#font_name#	Sets font name. To set Verdana as the font, use **#font_name=Verdana#**.
#font_size#	Sets font size. To set 12 as the font size, use **#font_size=12#**.

```
<e:workbook>
  <e:worksheet>
    <e:header>
      <f:facet name="left">
        This document was made on #date# and has #total_pages# pages.
      </f:facet>
      <f:facet name="right"> #time# </f:facet>
    </e:header>
  <e:worksheet>
</e:workbook>
```

19.11. PRINT AREAS AND TITLES

Print areas and titles are the children of worksheets and worksheet templates, and provide print areas and titles.

<e:printArea>	*Attributes* • **firstColumn** — A number value denoting column that holds the top-left corner of the area. The value is 0-based. • **firstRow** — A number value denoting the row that holds the top left corner of the area.The value is 0-based. • **lastColumn** — A number value denoting the column that holds the bottom-right corner of the area. The value is 0-based. • **lastRow** — A number value denoting the row that holds the bottom-right corner of the area. The value is 0-based. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:printTitles firstRow="0" firstColumn="0" lastRow="0"
       lastColumn="9"/>
    <e:printArea firstRow="1" firstColumn="0" lastRow="9"
lastColumn="9"/>
  </e:worksheet>
</e:workbook>
```

This defines a print title between A1:A10 and a print area between B2:J10.

19.12. WORKSHEET COMMANDS

Worksheet commands are the children of workbooks and are usually executed only once.

19.12.1. Grouping

Provides grouping of columns and rows.

<e:groupRows>	*Attributes* • **startRow** — A number value denoting the row at which to begin the grouping. The value is 0-based. • **endRow** — A number value denoting the row at which to end the grouping. The value is 0-based. • **collapse** — A Boolean determining whether the grouping is initially collapsed. *Child elements* • **none** *Facets* • **none**

<e:groupColumns>	*Attributes* • **startColumn** — A number value denoting the column at which to begin the grouping. The value is 0-based. • **endColumn** — A number value denoting the column at which to end the grouping. The value is 0-based. • **collapse** — A Boolean determining whether the grouping is initially collapsed. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:groupRows startRow="4" endRow="9" collapse="true"/>
    <e:groupColumns startColumn="0" endColumn="9" collapse="false"/>
  </e:worksheet>
</e:workbook>
```

This groups rows 5 through 10 and columns 5 through 10 so that the rows are initially collapsed (but not the columns).

19.12.2. Page breaks

Provides page breaks

<e:rowPageBreak>	*Attributes* • **row** — A number value denoting the row at which a page break should occur. The value is 0-based. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:rowPageBreak row="4"/>
  </e:worksheet>
</e:workbook>
```

This causes a page break at row 5.

19.12.3. Merging

Provides cell merging

<e:mergeCells>	*Attributes* • **startRow** — A number value denoting the row at which to begin the merge. The value is 0-based. • **startColumn** — A number value denoting the column at which to begin the merge. The value is 0-based. • **endRow** — A number value denoting the row at which to end the merge. The value is 0-based. • **endColumn** — A number value denoting the column at which to end the merge. The value is 0-based. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:worksheet>
    <e:mergeCells startRow="0" startColumn="0" endRow="9" endColumn="9"/>
  </e:worksheet>
</e:workbook>
```

This merges the cells in the range A1:J10.

19.13. DATATABLE EXPORTER

If you prefer to export an existing JSF datatable instead of writing a dedicated XHTML document, you can execute the **`org.jboss.seam.excel.excelExporter.export`** component, passing in the ID of the datatable as an Seam EL parameter. For example, say you have the following datatable:

```
<h:form id="theForm">
  <h:dataTable id="theDataTable" value="#{personList.personList}"
     var="person">
    ...
  </h:dataTable>
</h:form>
```

If you want to view this as a **Microsoft Excel** spreadsheet, place the following in the form:

```
<h:commandLink value="Export"
   action="#{excelExporter.export('theForm:theDataTable')}" />
```

You can also execute the exporter with a button, s:link, or other preferred method.

See Section 19.14, “Fonts and layout” for formatting.

19.14. FONTS AND LAYOUT

Output appearance is controlled with a combination of CSS style and tag attributes. CSS style attributes flow from parent to child, and let you use one tag to apply all attributes defined for that tag in the **`styleClass`** and **`style`** sheets.

If you have format masks or fonts that use special characters, such as spaces and semicolons, you can escape the CSS string with **`''`** characters such as **`xls-format-mask:'$;$'`**.

19.14.1. Stylesheet links

External stylesheets are referenced with the **`e:link`** tag. They are placed within the document as if they are children of the **`workbook`** tag.

`<e:link>`	*Attributes* • **URL** — The URL of the stylesheet. *Child elements* • **none** *Facets* • **none**

```
<e:workbook>
  <e:link URL="/css/excel.css"/>
</e:workbook>
```

This references a stylesheet located at **`/css/excel.css`**.

19.14.2. Fonts

This group of XLS-CSS attributes define a font and its attributes.

xls-font-family	The name of the font. Make sure the font you enter here is supported by your system.
xls-font-size	A plain number value denoting the font size.
xls-font-color	The color of the font. (See jxl.format.Colour.)
xls-font-bold	A Boolean determining whether the font is bold. Valid values are **`true`** and **`false`**.
xls-font-italic	A Boolean determining whether the font is italicized. Valid values are **`true`** and **`false`**.
xls-font-script-style	The script style of the font. (See jxl.format.ScriptStyle.)
xls-font-underline-style	The underline style of the font. (See jxl.format.UnderlineStyle.)
xls-font-struck-out	A Boolean determining whether the font is struck-through. Valid values are **`true`** and **`false`**.
xls-font	A shorthand notation for setting all values associated with font. Place the font name last. (If you wish to use a font with spaces in its name, use tick marks to surround the font. For example, **`'Times New Roman'`**.) Here, defined italicized, bold, or struck-through text with **`italic`**, **`bold`**, or **`struckout`**. For example: **`style="xls-font: red bold italic 22 Verdana"`**

19.14.3. Borders

This group of XLS-CSS attributes defines the borders of the cell.

xls-border-left-color	The border color of the left edge of the cell. (See jxl.format.Colour.)
xls-border-left-line-style	The border line style of the left edge of the cell. (See jxl.format.LineStyle.)
xls-border-left	A shorthand notation for setting the line style and color of the left edge of the cell. Use like so: **`style="xls-border-left: thick red"`**
xls-border-top-color	The border color of the top edge of the cell. (See jxl.format.Colour.)

xls-border-top-line-style	The border line style of the top edge of the cell. (See jxl.format.LineStyle.)
xls-border-top	A shorthand notation for setting the line style and color of the top edge of the cell. Use like so: `style="xls-border-top: red thick"`
xls-border-right-color	The border color of the right edge of the cell. (See jxl.format.Colour.)
xls-border-right-line-style	The border line style of the right edge of the cell. (See jxl.format.LineStyle.)
xls-border-right	A shorthand notation for setting the line style and color of the right edge of the cell. Use like so: `style="xls-border-right: thick red"`
xls-border-bottom-color	The border color of the bottom edge of the cell. (See jxl.format.Colour.)
xls-border-bottom-line-style	The border line style of the bottom edge of the cell. (See jxl.format.LineStyle.)
xls-border-bottom	A shorthand notation for setting the line style and color of the bottom edge of the cell. Use like so: `style="xls-border-bottom: thick red"`
xls-border	A shorthand notation for setting the line style and color for all edges of the cell. Use like so: `style="xls-border: thick red"`

19.14.4. Background

This group of XLS-CSS attributes defines the background of the cell.

xls-background-color	The color of the background. (See jxl.format.LineStyle.)
xls-background-pattern	The pattern of the background. (See jxl.format.Pattern.)
xls-background	A shorthand for setting the background color and pattern.

19.14.5. Column settings

This group of XLS-CSS attributes defines column properties.

xls-column-width	The width of a column. We recommend beginning with values of approximately 5000, and adjusting as required. Used by the `e:column` in XHTML mode.
xls-column-widths	The width of each column, respectively. We recommend beginning with values of approximately 5000, and adjusting as required. Used by the excel exporter, and placed in the datatable `style` attribute. Use numerical values, or * to bypass a column. For example: `style="xls-column-widths: 5000, 5000, *, 10000"`

xls-column-autosize	Determines whether the column should be autosized. Valid values are **`true`** and **`false`**.
xls-column-hidden	Determines whether the column is hidden. Valid values are **`true`** and **`false`**.
xls-column-export	Determines whether the column is shown in export. Valid values are **`true`** and **`false`**. Defaults to **`true`**.

19.14.6. Cell settings

This group of XLS-CSS attributes defines the cell properties.

xls-alignment	The alignment of the cell value. (See jxl.format.Alignment.)
xls-force-type	A string value determining the forced type of data in the cell. Valid values are **`general`**, **`number`**, **`text`**, **`date`**, **`formula`**, and **`bool`**. The type is automatically detected so there is rarely any use for this attribute.
xls-format-mask	The format mask of the cell. (See Section 19.6.2, “Format masks”.)
xls-indentation	A number value determining the indentation of the cell's contents.
xls-locked	Determines whether a cell is locked. Used with workbook level **`locked`**. Valid values are **`true`** or **`false`**.
xls-orientation	The orientation of the cell value. (See jxl.format.Orientation.)
xls-vertical-alignment	The vertical alignment of the cell value. (See jxl.format.VerticalAlignment.)
xls-shrink-to-fit	Determines whether cell values shrink to fit. Valid values are **`true`** and **`false`**.
xls-wrap	Determines whether the cell wraps new lines. Valid values are **`true`** and **`false`**.

19.14.7. The datatable exporter

The datatable exporter uses the same XLS-CSS attributes as the XHTML document, with the exception that column widths are defined with the **`xls-column-widths`** attribute on the datatable (since the UIColumn doesn't support the **`style`** or **`styleClass`** attributes).

19.14.8. Limitations

There are some known limitations to CSS support in the current version of Seam.

- When using **`.xhtml`** documents, stylesheets must be referenced through the **`<e:link>`** tag.
- When using the datatable exporter, CSS must be entered through style-attributes — external stylesheets are not supported.

19.15. INTERNATIONALIZATION

Only two resource bundle keys are used. Both are for invalid data formats, and both take a parameter that defines the invalid value.

- **`org.jboss.seam.excel.not_a_number`** — When a value thought to be a number could not be treated as such.
- **`org.jboss.seam.excel.not_a_date`** — When a value thought to be a date could not be treated as such.

19.16. LINKS AND FURTHER DOCUMENTATION

The core of **Microsoft Excel** functionality in Seam is based on the JExcelAPI library, which can be found on http://jexcelapi.sourceforge.net/. Most features and limitations are inherited from the JExcelAPI.

NOTE

JExcelAPI is not Seam. Any Seam-based issues are best reported in the JBoss Seam JIRA under the *Excel* module.

CHAPTER 20. EMAIL

Seam now includes an optional component for templating and sending email.

Email support is provided by **jboss-seam-mail.jar**. This **JAR** contains the mail JSF controls, used to construct emails, and the **mailSession** manager component.

For a demonstration of the email support available in Seam, see the **examples/mail** project. This demonstrates proper packaging, and contains a number of currently-supported key features.

You can test your mail system with Seam's integration testing environment. See Section 35.3.4, “Integration Testing Seam Mail” for details.

20.1. CREATING A MESSAGE

Seam uses Facelets to template emails.

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
           xmlns:m="http://jboss.com/products/seam/mail"
           xmlns:h="http://java.sun.com/jsf/html">

  <m:from name="Peter" address="peter@example.com" />
  <m:to name="#{person.firstname} #{person.lastname}">
    #{person.address}
  </m:to>
  <m:subject>Try out Seam!</m:subject>

  <m:body>
    <p><h:outputText value="Dear #{person.firstname}" />,</p>
    <p>You can try out Seam by visiting
    <a href="http://labs.jboss.com/jbossseam">
      http://labs.jboss.com/jbossseam
      </a>.
    </p>
    <p>Regards,</p>
    <p>Pete</p>
  </m:body>

</m:message>
```

The **<m:message>** tag wraps the whole message, and tells Seam to start rendering an email. Inside the **<m:message>** tag, we use an **<m:from>** tag to specify the sender, a **<m:to>** tag to specify a recipient, and a **<m:subject>** tag. (Note that EL is used as it would be in a normal Facelet.)

The **<m:body>** tag wraps the body of the email. You can use regular HTML tags inside the body, as well as JSF components.

Once the **m:message** is rendered, the **mailSession** is called to send the email. To send your email, have Seam render the view:

```
@In(create=true)
private Renderer renderer;

public void send() {
```

```
    try {
        renderer.render("/simple.xhtml");
        facesMessages.add("Email sent successfully");
    } catch (Exception e) {
        facesMessages.add("Email sending failed: " + e.getMessage());
    }
}
```

If, for example, you entered an invalid email address, then an exception is thrown, caught and then displayed to the user.

20.1.1. Attachments

Seam supports most standard Java types when working with files, so it is easy to attach files to an email.

For example, to email the **`jboss-seam-mail.jar`**:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam loads the file from the classpath and attaches it to the email. By default, this file is attached as **`jboss-seam-mail.jar`**, but you can change the attachment name by adding and editing the **`fileName`** attribute:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"
      fileName="this-is-so-cool.jar"/>
```

You can also attach a **`java.io.File`**, a **`java.net.URL`**:

```
<m:attachment value="#{numbers}"/>
```

Or a **`byte[]`** or a **`java.io.InputStream`**:

```
<m:attachment value="#{person.photo}" contentType="image/png"/>
```

For **`byte[]`** and **`java.io.InputStream`**, you will need to specify the MIME type of the attachment, since this information is not carried as part of the file.

You can attach a Seam-generated PDF, or any standard JSF view, by wrapping a **`<m:attachment>`** tag around your normal tags:

```
<m:attachment fileName="tiny.pdf">
  <p:document>
    A very tiny PDF
  </p:document>
</m:attachment>
```

To attach a set of files — for example, a set of pictures loaded from a database — you can use a **`<ui:repeat>`**:

```
<ui:repeat value="#{people}" var="person">
  <m:attachment value="#{person.photo}" contentType="image/jpeg"
                fileName="#{person.firstname}_#{person.lastname}.jpg"/>
</ui:repeat>
```

To display an attached image inline:

```
<m:attachment value="#{person.photo}" contentType="image/jpeg"
              fileName="#{person.firstname}_#{person.lastname}.jpg"
              status="personPhoto" disposition="inline" />
<img src="cid:#{personPhoto.contentId}" />
```

The **`cid:#{...}`** tag specifies that the attachments will be examined when attempting to locate the image. The **`cid`** — **`Content-ID`** — must match.

You must declare the attachment before trying to access the status object.

20.1.2. HTML/Text alternative part

Although most mail readers support HTML, some do not. You can add a plain text alternative to your email body:

```
<m:body>
  <f:facet name="alternative">
    Sorry, your email reader can not show our fancy email. Please go to
    http://labs.jboss.com/jbossseam to explore Seam.
  </f:facet>
</m:body>
```

20.1.3. Multiple recipients

Often you will want to send an email to a group of recipients, such as your users. All recipient mail tags can be placed inside a **`<ui:repeat>`**:

```
<ui:repeat value="#{allUsers} var="user">
  <m:to name="#{user.firstname} #{user.lastname}"
        address="#{user.emailAddress}"/>
</ui:repeat>
```

20.1.4. Multiple messages

Sometimes — for example, during a password reset — you will need to send a slightly different message to each recipient. The best way to do this is to place the whole message inside a **`<ui:repeat>`**:

```
<ui:repeat value="#{people}" var="p">
  <m:message>
    <m:from name="#{person.firstname} #{person.lastname}">
      #{person.address}
    </m:from>
    <m:to name="#{p.firstname}">#{p.address}</m:to>
    ...
  </m:message>
</ui:repeat>
```

20.1.5. Templating

The mail templating example shows that Facelets templating works with the Seam mail tags.

Our **template.xhtml** contains:

```
<m:message>
  <m:from name="Seam" address="do-not-reply@jboss.com" />
  <m:to name="#{person.firstname} #{person.lastname}">
    #{person.address}
  </m:to>
  <m:subject>#{subject}</m:subject>
  <m:body>
    <html>
      <body>
        <ui:insert name="body">
          This is the default body, specified by the template.
        </ui:insert>
      </body>
    </html>
  </m:body>
</m:message>
```

Our **templating.xhtml** contains:

```
<ui:param name="subject" value="Templating with Seam Mail"/>
<ui:define name="body">
  <p>
    This example demonstrates that you can easily use
    <i>facelets templating</i> in email!
  </p>
</ui:define>
```

You can also use Facelets source tags in your email. These must be placed in a JAR in **WEB-INF/lib** because referencing the **.taglib.xml** from **web.xml** is not reliable when using Seam Mail. (When mail is sent asynchronously, Seam Mail cannot access the full JSF or Servlet context, so it does not acknowledge **web.xml** configuration parameters.)

To configure Facelets or JSF further when sending mail, you will need to override the Renderer component and perform the configuration programmatically. This should only be done by advanced users.

20.1.6. Internationalization

Seam supports sending internationalized messages. By default, Seam uses encoding provided by JSF, but this can be overridden on the template:

```
<m:message charset="UTF-8">
  ...
</m:message>
```

The body, subject, and recipient and sender names are encoded. You will need to make sure that Facelets parses your page with the correct character set by setting the encoding of the template:

```
<?xml version="1.0" encoding="UTF-8"?>
```

20.1.7. Other Headers

Seam also provides support for some additional email headers. (See Section 20.4, “Tags”.) You can set the importance of the email, and ask for a read receipt:

```
<m:message xmlns:m="http://jboss.com/products/seam/mail"
           importance="low" requestReadReceipt="true"/>
```

Otherwise, you can add any header to the message by using the **<m:header>** tag:

```
<m:header name="X-Sent-From" value="JBoss Seam"/>
```

20.2. RECEIVING EMAILS

If you use Enterprise JavaBeans (EJB), you can use a MDB (Message Driven Bean) to receive email. JBoss provides a JCA adaptor (**mail-ra.rar**). You can configure **mail-ra.rar** like this:

```
@MessageDriven(activationConfig={
@ActivationConfigProperty(propertyName="mailServer",
                          propertyValue="localhost"),
@ActivationConfigProperty(propertyName="mailFolder",
                          propertyValue="INBOX"),
@ActivationConfigProperty(propertyName="storeProtocol",
                          propertyValue="pop3"),
@ActivationConfigProperty(propertyName="userName",
                          propertyValue="seam"),
@ActivationConfigProperty(propertyName="password",
                          propertyValue="seam")
})
@ResourceAdapter("mail-ra.rar")
@Name("mailListener")
public class MailListenerMDB implements MailListener {

    @In(create=true)
    private OrderProcessor orderProcessor;

    public void onMessage(Message message) {
        // Process the message
        orderProcessor.process(message.getSubject());
    }

}
```

Each message received calls **onMessage(Message message)**. Most Seam annotations work inside a MDB, but you must not access the persistence context.

20.3. CONFIGURATION

Include **jboss-seam-mail.jar** in your **WEB-INF/lib** directory to include email support in your application. If you use JBoss Enterprise Application Platform, no further configuration is required. If you do not use EAP, make sure you have the JavaMail API and a copy of the Java Active Framework. The versions distributed with Seam are **lib/mail.jar** and **lib/activation.jar** respectively.)

NOTE

The Seam Mail module requires both the use of the **seam-ui** package, and that Facelets be used as the view technology. Future versions of the library may also support the use of JSP.

The **mailSession** component uses JavaMail to talk to a 'real' SMTP server.

20.3.1. mailSession

If you are working in a Java EE 5 environment, a JavaMail session may be available through a JNDI look up. Otherwise, you can use a Seam-configured session.

The **mailSession** component's properties are described in more detail in Section 30.9, “Mail-related components”.

20.3.1.1. JNDI look up in EAP

The JBossAS **deploy/mail-service.xml** configures a JavaMail session binding into JNDI. The default service configuration must be altered for your network. http://wiki.jboss. org/wiki/Wiki.jsp?page=JavaMail describes the service in more detail.

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:mail="http://jboss.com/products/seam/mail">
  <mail:mail-session session-jndi-name="java:/Mail"/>
</components>
```

Here, we tell Seam to retrieve the mail session bound to **java:/Mail** from JNDI.

20.3.1.2. Seam-configured Session

A mail session can be configured via **components.xml**. Here we tell Seam to use **smtp.example.com** as the SMTP server:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:mail="http://jboss.com/products/seam/mail">
  <mail:mail-session host="smtp.example.com"/>
</components>
```

20.4. TAGS

Emails are generated using tags in the **http://jboss.com/products/seam/mail** namespace. Documents should always have the **message** tag at the root of the message. The message tag prepares Seam to generate an email.

Facelets standard templating tags can be used as normal. Inside the **body**, you can use any JSF tag. If the tag requires access to external resources such as stylesheets or JavaScript, be sure to set the **urlBase**.

<m:message>

Root tag of a mail message.

- **importance** — Sets the importance of the mail message. Valid values are **low**, **normal**, or **high**. Defaults to **normal**.
- **precedence** — Sets the precedence of the message, for example, **bulk**.
- **requestReadReceipt** — If set, a read receipt request will be added, and the read receipt will be sent to the **From:** address. Defaults to **false**.
- **urlBase** — If set, the value is prepended to the **requestContextPath**, allowing you to use components such as **<h:graphicImage>** in your emails.
- **messageId** — Explicitly sets the Message-ID.

<m:from>

Sets the **From:** address for the email. Only one exists per email.

- **name** — The name that the email comes from.
- **address** — The email address that the email comes from.

<m:replyTo>

Sets the **Reply-to:** address for the email. Only one exists per email.

- **address** — the email address the email comes from.

<m:to>

Adds a recipient to the email. Use multiple **<m:to>** tags for multiple recipients. This tag can be safely placed inside a repeat tag such as **<ui:repeat>**.

- **name** — The name of the recipient.
- **address** — The email address of the recipient.

<m:cc>

Adds a CC recipient to the email. Use multiple **<m:cc>** tags for multiple CCs. This tag can be safely placed inside a iterator tag such as <ui:repeat>.

- **name** — The name of the recipient.
- **address** — The email address of the recipient.

<m:bcc>

Adds a BCC recipient to the email. Use multiple **<m:bcc>** tags for multiple bccs. This tag can be safely placed inside a repeat tag such as **<ui:repeat>**.

- **name** — The name of the recipient.
- **address** — The email address of the recipient.

<m:header>

Adds a header to the email. (For example, **`X-Sent-From: JBoss Seam`**.)

- **`name`** — The name of the header to add. (For example, **`X-Sent-From`**.)
- **`value`** — The value of the header to add. (For example, **`JBoss Seam`**.)

<m:attachment>

Adds an attachment to the email.

- **`value`** — The file to attach:
  - **`String`** — A **`String`** is interpreted as a path to file within the classpath.
  - **`java.io.File`** — An EL expression can reference a **`File`** object.
  - **`java.net.URL`** — An EL expression can reference a **`URL`** object.
  - **`java.io.InputStream`** — An EL expression can reference an **`InputStream`**. In this case both a **`fileName`** and a **`contentType`** must be specified.
  - **`byte[]`** — An EL expression can reference a **`byte[]`**. In this case both a **`fileName`** and a **`contentType`** must be specified.

  If the value attribute is omitted:

  - If this tag contains a **`<p:document>`** tag, the document described will be generated and attached to the email. A **`fileName`** should be specified.
  - If this tag contains other JSF tags, a HTML document will be generated from them and attached to the email. A **`fileName`** should be specified.
- **`fileName`** — Specifies the file name to use for the attached file.
- **`contentType`** — Specifies the MIME type of the attached file.

<m:subject>

Sets the subject for the email.

<m:body>

Sets the body for the email. Supports an **`alternative`** facet which, if a HTML email is generated, can contain alternative text for a mail reader which does not support HTML.

- **`type`** — If set to **`plain`**, a plain text email will be generated. Otherwise, a HTML email is generated.

CHAPTER 21. ASYNCHRONICITY AND MESSAGING

Seam makes it easy to perform work asynchronously from a web request. Asynchronicity in Java EE is usually linked with JMS, and where your quality of service requirements are strict and well-defined, this is logical. It is easy to send JMS messages through Seam components.

However, for many use cases, JMS is more powerful than necessary. Seam layers a simple, asynchronous method and event facility over your choice of *dispatchers*:

- **`java.util.concurrent.ScheduledThreadPoolExecutor`** (by default)
- the EJB timer service (for EJB 3.0 environments)
- Quartz

21.1. ASYNCHRONICITY

Asynchronous events and method calls have the same quality of service expectations as the underlying dispatcher mechanism. The default dispatcher, based upon a **`ScheduledThreadPoolExecutor`** performs efficiently but provides no support for persistent asynchronous tasks, and hence no guarantee that a task will ever actually be executed. If you are working in an environment that supports EJB 3.0, add the following line to **`components.xml`** to ensure that your asynchronous tasks are processed by the container's EJB timer service:

```
<async:timer-service-dispatcher/>
```

If you want to use asynchronous methods in Seam, you do not need to interact directly with the Timer service. However, it is important that your EJB3 implementation has the option of using persistent timers, which give some guarantee that the task will eventually be processed.

Alternatively, you can use the open source Quartz library to manage asynchronous method. To do so, bundle the Quartz library **`JAR`** (found in the **`lib`** directory) in your **`EAR`**, and declare it as a Java module in **`application.xml`**. You can configure the Quartz dispatcher by adding a Quartz property file to the classpath —this file must be named **`seam.quartz.properties`**. To install the Quartz dispatcher, you will also need to add the following line to **`components.xml`**:

```
<async:quartz-dispatcher/>
```

Since the Seam API for the default **`ScheduledThreadPoolExecutor`**, the EJB3 **`Timer`**, and the Quartz **`Scheduler`** are very similar, you can "plug and play" by adding a line to **`components.xml`**.

21.1.1. Asynchronous methods

An asynchronous call allows a method call to be processed asynchronously (in a different thread) to the caller. Usually, asynchronous calls are used when we want to send an immediate response to the client, and simultaneously process expensive work in the background. This pattern works well in AJAX applications, where the client can automatically poll the server for the result of the work.

For EJB components, annotate the implementation of the bean to specify that a method be processed asynchronously. For JavaBean components, annotate the component implementation class:

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
```

```
{
  @Asynchronous
  public void processPayment(Payment payment) {
    //do some work!
  }
}
```

Asynchronicity is transparent to the bean class. It is also transparent to the client:

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String pay() {
    paymentHandler.processPayment( new Payment(bill) );
    return "success";
  }
}
```

The asynchronous method is processed in a fresh event context, and has no access to the session or conversation context state of the caller. However, the business process context *is* propagated.

You can schedule asynchronous method calls for delayed execution with the **`@Duration`**, **`@Expiration`** and **`@IntervalDuration`** annotations.

```
@Local
public interface PaymentHandler {
  @Asynchronous
  public void processScheduledPayment(Payment payment,
                                      @Expiration Date date);

  @Asynchronous
  public void processRecurringPayment(Payment payment,
                                      @Expiration Date date,
                                      @IntervalDuration Long interval);
}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String schedulePayment() {
    paymentHandler.processScheduledPayment(new Payment(bill),
                                           bill.getDueDate() );
    return "success";
  }

  public String scheduleRecurringPayment() {
    paymentHandler.processRecurringPayment(new Payment(bill),
```

```
                                                bill.getDueDate(), ONE_MONTH
);
    return "success";
  }
}
```

Both client and server can access the **`Timer`** object associated with the invocation. The **`Timer`** shown below is the EJB3 timer used with the EJB3 dispatcher. For the default **`ScheduledThreadPoolExecutor`**, the timer returns **`Future`** from the JDK. For the Quartz dispatcher, it returns **`QuartzTriggerHandle`**, which will be discussed in the next section.

```
@Local
public interface PaymentHandler
{
  @Asynchronous
  public Timer processScheduledPayment(Payment payment,
                                       @Expiration Date date);
}
```

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler {
  @In Timer timer;

  public Timer processScheduledPayment(Payment payment,
                                       @Expiration Date date) {
    //do some work!
    return timer; //note that return value is completely ignored
  }
}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String schedulePayment() {
    Timer timer =
      paymentHandler.processScheduledPayment(new Payment(bill),
                                             bill.getDueDate());
    return "success";
  }
}
```

Asynchronous methods cannot return any other value to the caller.

21.1.2. Asynchronous methods with the Quartz Dispatcher

The Quartz dispatcher lets you use the **`@Asynchronous`**, **`@Duration`**, **`@Expiration`**, and **`@IntervalDuration`** annotations, as above, but it also supports several additional annotations.

The **@FinalExpiration** annotation specifies an end date for a recurring task. Note that you can inject the **QuartzTriggerHandle**.

```
@In QuartzTriggerHandle timer;

// Defines the method in the "processor" component
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalDuration Long
interval,
                                           @FinalExpiration Date
endDate,
                                           Payment payment) {
  // do the repeating or long running task until endDate
}

... ...

// Schedule the task in the business logic processing code
// Starts now, repeats every hour, and ends on May 10th, 2010
Calendar cal = Calendar.getInstance ();
cal.set (2010, Calendar.MAY, 10);
processor.schedulePayment(new Date(), 60*60*1000, cal.getTime(), payment);
```

Note that this method returns the **QuartzTriggerHandle** object, which can be used to stop, pause, and resume the scheduler. The **QuartzTriggerHandle** object is serializable, so it can be saved into the database if required for an extended period of time.

```
QuartzTriggerHandle handle=
    processor.schedulePayment(payment.getPaymentDate(),
                              payment.getPaymentCron(),
                              payment);
payment.setQuartzTriggerHandle( handle );
// Save payment to DB

// later ...

// Retrieve payment from DB
// Cancel the remaining scheduled tasks
payment.getQuartzTriggerHandle().cancel();
```

The **@IntervalCron** annotation supports Unix cron job syntax for task scheduling. For example, the following asynchronous method runs at 2:10pm and at 2:44pm every Wednesday in the month of March.

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalCron String cron,
                                           Payment payment) {
  // do the repeating or long running task
}

... ...
```

```
// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
     processor.schedulePayment(new Date(), "0 10,44 14 ? 3 WED", payment);
```

The **@IntervalBusinessDay** annotation supports invocation in the "nth Business Day" scenario. For instance, the following asynchronous method runs at 14:00 on the 2nd business day of each month. All weekends and US Federal holidays are excluded from the business days by default.

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                  @IntervalBusinessDay NthBusinessDay
nth,
                                  Payment payment) {
  // do the repeating or long running task
}

... ...

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
    processor.schedulePayment(new Date(),
                              new NthBusinessDay(2, "14:00", WEEKLY),
                              payment);
```

The **NthBusinessDay** object contains the configuration of the invocation trigger. You can specify more holidays (company holidays and non-US holidays, for example) in the **additionalHolidays** property.

```
public class NthBusinessDay implements Serializable {
  int n;
  String fireAtTime;
  List<Date> additionalHolidays;
  BusinessDayIntervalType interval;
  boolean excludeWeekends;
  boolean excludeUsFederalHolidays;

  public enum BusinessDayIntervalType { WEEKLY, MONTHLY, YEARLY }

  public NthBusinessDay () {
    n = 1;
    fireAtTime = "12:00";
    additionalHolidays = new ArrayList<Date> ();
    interval = BusinessDayIntervalType.WEEKLY;
    excludeWeekends = true;
    excludeUsFederalHolidays = true;
  }
      ... ...
}
```

The **@IntervalDuration**, **@IntervalCron**, and **@IntervalNthBusinessDay** annotations are mutually exclusive. Attempting to use them in the same method will cause a **RuntimeException** error.

21.1.3. Asynchronous events

Component-driven events can also be asynchronous. To raise an event for asynchronous processing,

call the **raiseAsynchronousEvent()** method of the **Events** class. To schedule a timed event, call the **raisedTimedEvent()** method and pass a schedule object. (For the default dispatcher or timer service dispatcher, use **TimerSchedule**.) Components can observe asynchronous events as usual, but only business process context is propagated to the asynchronous thread.

21.1.4. Handling exceptions from asynchronous calls

Each asynchronous dispatcher behaves differently when an exception propagates through it. For example, the **java.util.concurrent** suspends further executions of a repeating call, and the EJB3 timer service swallows the exception, so Seam catches any exception that propagates from the asynchronous call before it reaches the dispatcher.

By default, any exception that propagates from an asynchronous execution will be caught and logged at error level. You can customize this behavior globally by overriding the **org.jboss.seam.async.asynchronousExceptionHandler** component:

```
@Scope(ScopeType.STATELESS)
@Name("org.jboss.seam.async.asynchronousExceptionHandler")
public class MyAsynchronousExceptionHandler
             extends AsynchronousExceptionHandler {
  @Logger Log log;

  @In Future timer;

  @Override
  public void handleException(Exception exception) {
    log.debug(exception);
    timer.cancel(false);
  }

}
```

Here, with **java.util.concurrent** dispatcher, we inject its control object and cancel all future invocations when an exception is encountered.

You can alter this behavior for an individual component by implementing the **public void handleAsynchronousException(Exception exception);** method on that component, like so:

```
public void handleAsynchronousException(Exception exception) {
  log.fatal(exception);
}
```

21.2. MESSAGING IN SEAM

It is easy to send and receive JMS messages to and from Seam components.

21.2.1. Configuration

To configure Seam infrastructure to send JMS messages, you must first tell Seam about the topics and queues you want to send messages to, and where to find the **QueueConnectionFactory** and **TopicConnectionFactory**, depending on your requirements.

By default, Seam uses **`UIL2ConnectionFactory`**, the default connection factory with JBossMQ. If you use another JMS provider, you must set one or both of **`queueConnection.queueConnectionFactoryJndiName`** and **`topicConnection.topicConnectionFactoryJndiName`**, in either **`seam.properties`**, **`web.xml`**, or **`components.xml`**.

To install Seam-managed **`TopicPublisher`**s and **`QueueSender`**s, you must also list topics and queues in **`components.xml`**:

```
<jms:managed-topic-publisher name="stockTickerPublisher"
     auto-create="true" topic-jndi-name="topic/stockTickerTopic"/>

<jms:managed-queue-sender name="paymentQueueSender"
     auto-create="true" queue-jndi-name="queue/paymentQueue"/>
```

21.2.2. Sending messages

Once configuration is complete, you can inject a JMS **`TopicPublisher`** and **`TopicSession`** into any component:

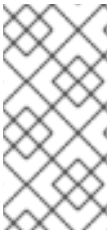
```
@Name("stockPriceChangeNotifier")
public class StockPriceChangeNotifier {
  @In private TopicPublisher stockTickerPublisher;

  @In private TopicSession topicSession;

  public void publish(StockPrice price) {
    try {
      stockTickerPublisher.publish(topicSession
                                   .createObjectMessage(price));
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Or, to work with a queue:

```
@Name("paymentDispatcher")
public class PaymentDispatcher {
  @In private QueueSender paymentQueueSender;

  @In private QueueSession queueSession;

  public void publish(Payment payment) {
    try {
      paymentQueueSender.send(queueSession.createObjectMessage(payment));
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

21.2.3. Receiving messages using a message-driven bean

You can process messages with any EJB3 message-driven bean. Message-driven beans can sometimes be Seam components, in which case, you can inject other event- and application-scoped Seam components. The following is an example of the payment receiver, which delegates to the payment processor.



NOTE

You may need to set the **`create`** attribute on the **`@In`** annotation to **`true`** so that Seam can create an instance of the component to be injected. (This is necessary only if the component does not support auto-creation — that is, it is not annotated with **`@Autocreate`**.)

First, create a message-driven bean to receive the message:

```
@MessageDriven(activationConfig =
    {@ActivationConfigProperty(propertyName = "destinationType",
                               propertyValue = "javax.jms.Queue"),
     @ActivationConfigProperty(propertyName = "destination",
                               propertyValue = "queue/paymentQueue")
})

@Name("paymentReceiver")
public class PaymentReceiver implements MessageListener
{
  @Logger private Log log;

  @In(create = true) private PaymentProcessor paymentProcessor;

  @Override
  public void onMessage(Message message)
  {
    try {
      paymentProcessor.processPayment((Payment) ((ObjectMessage)
                                      message).getObject());
    } catch (JMSException ex) {
      log.error("Message payload did not contain a Payment object", ex);
    }
  }
}
```

Next, implement the Seam component to which the receiver will delegate payment processing:

```
@Name("paymentProcessor")
public class PaymentProcessor {
  @In private EntityManager entityManager;

  public void processPayment(Payment payment) {
    // perhaps do something more fancy
    entityManager.persist(payment);
  }
}
```

If you want to perform transaction operations in your message-driven bean, ensure that you are working with an XA datasource, or you will not be able to roll back database changes in the event that a database transaction commits, but a subsequent message operation fails.

21.2.4. Receiving messages in the client

Seam Remoting lets you subscribe to a JMS topic from client-side JavaScript. You can find more information in Chapter 24, *Remoting*.

CHAPTER 22. CACHING

The database is the primary bottleneck in almost all enterprise applications, and the least-scalable tier of the runtime environment, so anything we can do to reduce the number of times the database is accessed can dramatically improve application performance.

A well-designed Seam application will feature a rich, multi-layered caching strategy that impacts every layer of the application, including:

- A cache for the database. This is vital, but cannot scale like a cache in the application tier.
- A secondary cache of data from the database, provided by your ORM solution (Hibernate, or another JPA implementation). In a clustered environment, keeping cache data transactionally consistent with both the database and the rest of the cluster can be very expensive to implement effectively. Therefore, this secondary cache is best used to store data that is rarely updated, and shared between many users. In traditional stateless architectures, this space is often used (ineffectively) to store conversational state.
- The Seam conversational context, which is a cache of conversational state. Components in the conversation context store state relating to the current user interaction.
- The Seam-managed persistence context, which acts as a cache of data read in the current conversation. (An Enterprise JavaBean [EJB] container-managed persistence context associated with a conversation-scoped stateful session bean can be used in place of a Seam-managed persistence context.) Seam optimizes the replication of Seam-managed persistence contexts in a clustered environment, and optimistic locking provides sufficient transactional consistency with the database. Unless you read thousands of objects into a single persistence context, the performance implications of this cache are minimal.
- The Seam application context, which can be used to cache non-transactional state. State held here is not visible to other nodes in the cluster.
- The Seam **`cacheProvider`** component within the application, which integrates JBossCache, or Ehcache into the Seam environment. State held here is visible to other nodes if your cache supports running in clustered mode.
- Finally, Seam can cache rendered fragments of a JSF page. Unlike the ORM secondary cache, this is not automatically invalidated when data is updated, so you will need to write application code to perform explicit invalidation, or set appropriate expiry policies.

For more information about the secondary cache, you will need to refer to the documentation of your ORM solution, since this can be quite complex. In this section, we discuss the use of caching directly via the **`cacheProvider`** component, or caching as stored page fragments, via the **`<s:cache>`** control.

22.1. USING CACHING IN SEAM

The built-in **`cacheProvider`** component manages an instance of:

JBoss Cache 3.2.x

`org.jboss.cache.Cache`

EhCache

`net.sf.ehcache.CacheManager`

Any immutable Java object placed in the cache will be stored there and replicated across the cluster (if replication is supported and enabled). To keep mutable objects in the cache, read the underlying caching project documentation for information about notifying the cache of changes made to stored objects.

To use **`cacheProvider`**, you need to include the JARs of the cache implementation in your project:

JBoss Cache 3.2.x

- **`jbosscache-core.jar`** — JBoss Cache 3.2.x
- **`jgroups.jar`** — JGroups 2.6.x

Ehcache

- **`ehcache.jar`** — Ehcache 1.2.3

In **EAR** deployments of Seam, it is recommended that cache **JAR**s and configuration go directly into the **EAR**.

You will also need to provide a configuration file for JBossCache. Place **`cache-configuration.xml`** with an appropriate cache configuration into the classpath — for example, the EJB JAR or **`WEB-INF/classes`**. Refer to the JBossCache documentation for more information about configuring the JBossCache.

You can find a sample **`cache-configuration.xml`** in **`examples/blog/resources/META-INF/cache-configuration.xml`**.

Ehcache will run in its default configuration without a configuration file.

To alter the configuration file in use, configure your cache in **`components.xml`**:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider
        configuration="META-INF/cache/cache-configuration.xml" />
</components>
```

Now you can inject the cache into any Seam component:

```
@Name("chatroomUsers")
@Scope(ScopeType.STATELESS)

public class ChatroomUsers {
  @In CacheProvider cacheProvider;
  @Unwrap public Set<String> getUsers() throws CacheException   {
    Set<String> userList =
      (Set<String>) cacheProvider.get("chatroom", "userList");
    if (userList==null) {
      userList = new HashSet<String>();
      cacheProvider.put("chatroom", "userList", userList);
    } return userList;
  }
}
```

If you want multiple cache configurations available to your application, use **components.xml** to configure multiple cache providers:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache3-provider name="myCache"
        configuration="myown/cache.xml"/>
  <cache:jboss-cache3-provider name="myOtherCache"
        configuration="myother/cache.xml"/>
</components>
```

22.2. PAGE FRAGMENT CACHING

The **<s:cache>** tag is Seam's solution to the problem of page fragment caching in JSF. **<s:cache>** uses **pojoCache** internally, so you will need to follow the previous steps — place the **JAR**s in the **EAR** and edit additional configuration options — before you can use it.

<s:cache> stores some rendered content that is rarely updated. For example, the welcome page of our blog displays recent blog entries:

```
<s:cache key="recentEntries-#{blog.id}" region="welcomePageFragments">
  <h:dataTable value="#{blog.recentEntries}" var="blogEntry">
    <h:column>
      <h3>#{blogEntry.title}</h3>
      <div>
        <s:formattedText value="#{blogEntry.body}"/>
      </div>
    </h:column>
  </h:dataTable>
</s:cache>
```

The **key** lets you store multiple versions of each page fragment. In this case, there is one cached version per blog. The **region** determines the cache or region node where all versions are stored. Different nodes may have differing expiry policies.

The **<s:cache>** cannot tell when the underlying data is updated, so you will need to manually remove the cached fragment when a change occurs:

```
public void post() {
  ...
  entityManager.persist(blogEntry);
  cacheProvider.remove("welcomePageFragments",
                       "recentEntries-" + blog.getId());
}
```

If changes need not be immediately visible to the user, you can set up a short expiry period on the cache node.

CHAPTER 23. WEB SERVICES

Seam integrates with JBossWS (JWS) to allow standard Java EE web services to take full advantage of Seam's contextual framework, including conversational web service support. This chapter guides you through web service configuration for a Seam environment.

23.1. CONFIGURATION AND PACKAGING

For Seam to create contexts for web service requests, it must first have access to those requests. **`org.jboss.seam.webservice.SOAPRequestHandler`** is a **`SOAPHandler`** implementation that manages the Seam component life cycle during the scope of a web service request.

`standard-jaxws-endpoint-config.xml` (a configuration file) should be placed in the **`META-INF`** directory of the **`JAR`** file that contains the web service classes. This file contains the following SOAP handler configuration:

```
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:javaee="http://java.sun.com/xml/ns/javaee"
              xsi:schemaLocation=
                "urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd">

  <endpoint-config>
    <config-name>Seam WebService Endpoint</config-name>

    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings>
          ##SOAP11_HTTP
        </javaee:protocol-bindings>
        <javaee:handler>
          <javaee:handler-name>
            SOAP Request Handler
          </javaee:handler-name>
          <javaee:handler-class>
            org.jboss.seam.webservice.SOAPRequestHandler
          </javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>

    </pre-handler-chains>

  </endpoint-config>

</jaxws-config>
```

23.2. CONVERSATIONAL WEB SERVICES

Seam uses a SOAP header element in both SOAP request and response messages to carry the conversation ID between the consumer and the service. One example of a web service request containing a conversation ID is:

```
<soapenv:Envelope
         xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
          xmlns:seam="http://seambay.example.seam.jboss.org/">

  <soapenv:Header>
    <seam:conversationId
xmlns:seam='http://www.jboss.org/seam/webservice'>
      2
    </seam:conversationId>
  </soapenv:Header>

  <soapenv:Body>
    <seam:confirmAuction/>
  </soapenv:Body>

</soapenv:Envelope>
```

The above SOAP message contains a **`conversationId`** element, which contains the conversation ID for the request — in this case, **`2`**. Because web services can be consumed by a variety of web service clients written in a variety of languages, the developer is responsible for implementing conversation ID propagation between individual web services to be used in a single conversation's scope.

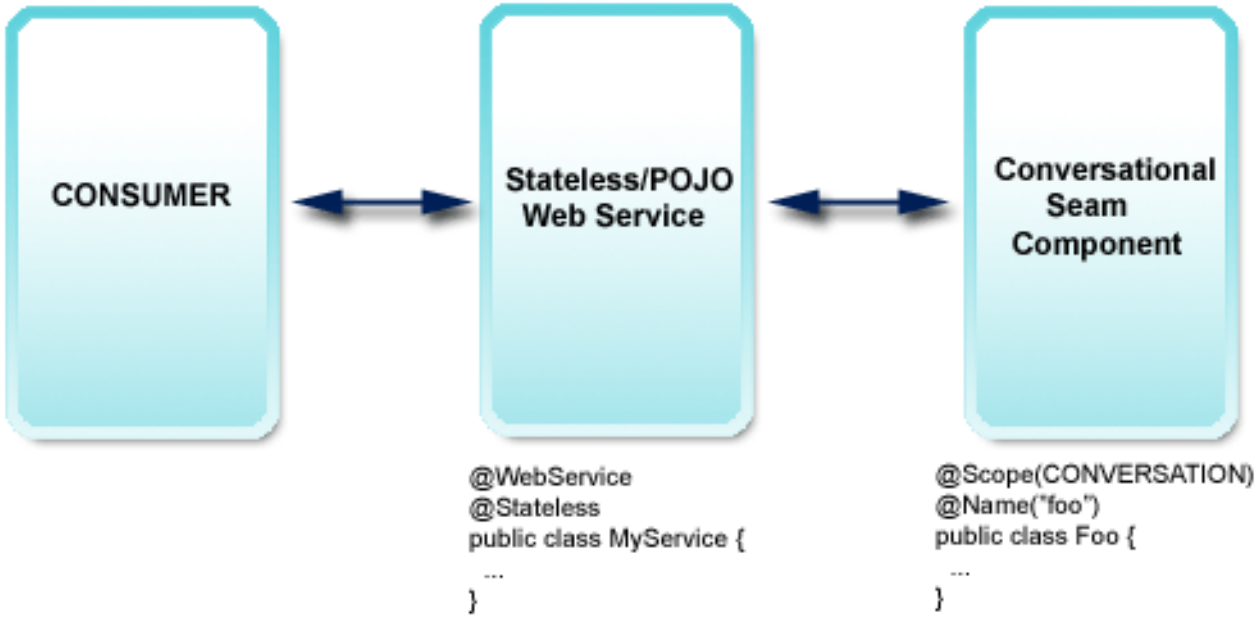
The **`conversationId`** header element must be qualified with a namespace of **`http://www.jboss.org/seam/webservice`**, or Seam will be unable to read the conversation ID from the request. An example response to the above request message is:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header>
    <seam:conversationId
xmlns:seam='http://www.jboss.org/seam/webservice'>
      2
    </seam:conversationId>
  </env:Header>
  <env:Body>
    <confirmAuctionResponse
       xmlns="http://seambay.example.seam.jboss.org/"/>
  </env:Body>
</env:Envelope>
```

Note that the response message contains the same **`conversationId`** element as the request.

23.2.1. A Recommended Strategy

Since web services must be implemented as either stateless session beans or POJOs, we recommend that conversational web services implement the web service as a facade for a conversational Seam component.



If the web service is written as a stateless session bean, it can be transformed into a Seam component by annotating it with **@Name**. This allows Seam bijection, and other features, to be used in the web service class itself.

23.3. AN EXAMPLE WEB SERVICE

The example code that follows is from the seamBay example application, which can be found in Seam's **/examples** directory, and follows the recommended strategy outlined in the previous section. First, we will look at the web service class and one of its web service methods:

```
@Stateless
@WebService(name = "AuctionService", serviceName = "AuctionService")
public class AuctionService implements AuctionServiceRemote
{
  @WebMethod
  public boolean login(String username, String password)
  {
    Identity.instance().setUsername(username);
    Identity.instance().setPassword(password);
    Identity.instance().login();
    return Identity.instance().isLoggedIn();
  }

  // snip
}
```

Here, the web service is a stateless session bean annotated with the JWS annotations from the **javax.jws** package, as defined by JSR-181. The **@WebService** annotation tells the container that this class implements a web service. The **@WebMethod** annotation on the **login()** method identifies the method as a web service method. The **name** and **serviceName** attributes in the **@WebService** annotation are optional.

When the web service is a stateless session bean, each method that will be exposed as a web service method must also be declared in the remote interface of the web service class. In the previous example, since the **AuctionServiceRemote** interface is annotated as a **@WebService**, it must declare the **login()** method.

In the previous example, the web service implements a **login()** method that delegates to Seam's built-in **Identity** component. As our recommended strategy suggests, the web service is written as a simple facade. The real work takes place in a Seam component. This means that business logic is reused efficiently between web services and other clients.

In the following example, the web service method begins a new conversation by delegating to the **AuctionAction.createAuction()** method:

```
@WebMethod
public void createAuction(String title, String description, int
categoryId)
{
  AuctionAction action =
    (AuctionAction) Component.getInstance(AuctionAction.class, true);
  action.createAuction();
  action.setDetails(title, description, categoryId);
}
```

The code from **AuctionAction** is as follows:

```
@Begin
public void createAuction()
{
  auction = new Auction();
  auction.setAccount(authenticatedAccount);
  auction.setStatus(Auction.STATUS_UNLISTED);
  durationDays = DEFAULT_AUCTION_DURATION;
}
```

Here, we see how web services can participate in long-running conversations by acting as a facade and delegating the real work to a conversational Seam component.

23.4. RESTFUL HTTP WEB SERVICES WITH RESTEASY

Seam integrates the RESTEasy implementation of the JAX-RS specification (JSR 311). You can decide which of the following features are integrated with your Seam application:

- RESTEasy bootstrap and configuration, with automatic resource detection. and providers.
- SeamResourceServlet-served HTTP/REST requests, without the need for an external servlet or configuration in **web.xml**.
- Resources written as Seam components with full Seam life cycle management and bijection.

23.4.1. RESTEasy configuration and request serving

First, download the RESTEasy libraries and the **jaxrs-api.jar**, and deploy them alongside the integration library (**jboss-seam-resteasy.jar**) and any other libraries your application requires.

In seam-gen based projects, this can be done by appending **jaxrs-api.jar**, **resteasy-jaxrs.jar** and **jboss-seam-resteasy.jar** to the **deployed-jars.list** (war deployment) or **deployed-jars-ear.list** (ear deployment) file. For a JBDS based project, copy the libraries mentioned above to the **EarContent/lib** (ear deployment) or **WebContent/WEB-INF/lib** (war deployment) folder and reload the project in the IDE.

All classes annotated with **`@javax.ws.rs.Path`** will automatically be discovered and registered as HTTP resources at start up. Seam automatically accepts and serves HTTP requests with its built-in **`SeamResourceServlet`**. The URI of a resource is built like so:

- The URI begins with the pattern mapped in **`web.xml`** for the **`SeamResourceServlet`** — in the examples provided, **`/seam/resource`**. Change this setting to expose your RESTful resources under a different base. Remember that this is a *global* change, and other Seam resources (**`s:graphicImage`**) will also be served under this base path.
- Seam's RESTEasy integration then appends a configurable string to the base path (**`/rest`** by default). So, in the example, the full base path of your resources would be **`/seam/resource/rest`**. We recommend changing this string in your application to something more descriptive — add a version number to prepare for future REST API upgrades. This allows old clients to keep the old URI base.
- Finally, the resource is made available under the defined **`@Path`**. For example, a resource mapped with **`@Path("/customer")`** would be available under **`/seam/resource/rest/customer`**.

The following resource definition would return a plain text representation for any GET request using the URI **`http://your.hostname/seam/resource/rest/customer/123`**:

```
@Path("/customer")
public class MyCustomerResource {

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
    public String getCustomer(@PathParam("customerId") int id) {
         return ...;
    }

}
```

If these defaults are acceptable, there is no need for additional configuration. However, if required, you can configure RESTEasy in your Seam application. First, import the **`resteasy`** namespace into your XML configuration file header:

```
<components
   xmlns="http://jboss.com/products/seam/components"
   xmlns:resteasy="http://jboss.com/products/seam/resteasy"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
     "http://jboss.com/products/seam/resteasy
      http://jboss.com/products/seam/resteasy-2.2.xsd
      http://jboss.com/products/seam/components
      http://jboss.com/products/seam/components-2.2.xsd">
```

```
<resteasy:application resource-path-prefix="/restv1"/>
```

The full base path to your resources is now **`/seam/resource/restv1/{resource}`**. Note that your **`@Path`** definitions and mappings do not change. This is an application-wide switch, usually used for versioning of the HTTP API.

If you want to map the full path in your resources, you can disable base path stripping:

```
<resteasy:application strip-seam-resource-path="false"/>
```

Here, the path of a resource is now mapped with **@Path("/seam/resource/rest/customer")**. Disabling this feature binds your resource class mappings to a particular deployment scenario. This is *not* recommended.

Seam scans your classpath for any deployed **@javax.ws.rs.Path** resources or **@javax.ws.rs.ext.Provider** classes. You can disable scanning and configure these classes manually like so:

```
<resteasy:application
     scan-providers="false"
     scan-resources="false"
     use-builtin-providers="true">

  <resteasy:resource-class-names>
    <value>org.foo.MyCustomerResource</value>
    <value>org.foo.MyOrderResource</value>
         <value>org.foo.MyStatelessEJBImplementation</value>
  </resteasy:resource-class-names>

  <resteasy:provider-class-names>
    <value>org.foo.MyFancyProvider</value>
  </resteasy:provider-class-names>

 </resteasy:application>
```

The **use-built-in-providers** switch enables (default) or disables the RESTEasy built-in providers. Since these provide plain text, JSON and JAXB marshaling, we recommend that these are left enabled.

RESTEasy supports plain EJBs (EJBs that are not Seam components) as resources. Instead of configuring the JNDI names in a non-portable fashion in **web.xml** (see RESTEasy documentation), you can simply list the EJB implementation classes, not the business interfaces, in **components.xml** as shown above. Note that you have to annotate the **@Local** interface of the EJB with **@Path**, **@GET**, and so on - not the bean implementation class. This allows you to keep your application deployment-portable with the global Seam **jndi-pattern** switch on **<core:init/>**. Note that plain (non-Seam component) EJB resources will not be found even if scanning of resources is enabled, you always have to list them manually. Again, this whole paragraph is only relevant for EJB resources that are not also Seam components and that do not have an **@Name** annotation.

Finally, you can configure media type and language URI extensions:

```
<resteasy:application>

    <resteasy:media-type-mappings>
      <key>txt</key>
      <value>text/plain</value>
    </resteasy:media-type-mappings>

    <resteasy:language-mappings>
       <key>deutsch</key><value>de-DE</value>
```

```
    </resteasy:language-mappings>

</resteasy:application>
```

This definition would map the URI suffix of **`.txt.deutsch`** to the additional **`Accept`** and **`Accept-Language`** header values, **`text/plain`** and **`de-DE`**.

23.4.2. Resources and providers as Seam components

Resource and provider instances are, by default, managed by RESTEasy. A resource class will be instantiated by RESTEasy and serve a single request, after which it will be destroyed. This is the default JAX-RS life cycle. Providers are instantiated once for the entire application. These are stateless singletons.

Resources and providers can also be written as Seam components to take advantage of Seam's richer life cycle management, and bijection and security abilities. Make your resource class into a Seam component like so:

```
@Name("customerResource")
@Path("/customer")
public class MyCustomerResource {

    @In
    CustomerDAO customerDAO;

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
    public String getCustomer(@PathParam("customerId") int id) {
         return customerDAO.find(id).getName();
    }

}
```

A **`customerResource`** instance is now handled by Seam when a request hits the server. This component is event-scoped, so its life cycle is identical to that of the JAX-RS. However, the Seam JavaBean component gives you full injection support, and full access to all other components and contexts. Session, application, and stateless resource components are also supported. These three scopes allow you to create an effectively stateless Seam middle-tier HTTP request-processing application.

You can annotate an interface and keep the implementation free from JAX-RS annotations:

```
@Path("/customer")
public interface MyCustomerResource {

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
    public String getCustomer(@PathParam("customerId") int id);

}
```

```
@Name("customerResource")
@Scope(ScopeType.STATELESS)
```

```
public class MyCustomerResourceBean implements MyCustomerResource {

    @In
    CustomerDAO customerDAO;

    public String getCustomer(int id) {
         return customerDAO.find(id).getName();
    }

}
```

You can use **SESSION**-scoped Seam components. By default, the session will however be shortened to a single request. In other words, when an HTTP request is being processed by the RESTEasy integration code, an HTTP session will be created so that Seam components can utilize that context. When the request has been processed, Seam will look at the session and decide if the session was created only to serve that single request (no session identifier has been provided with the request, or no session existed for the request). If the session has been created only to serve this request, the session will be destroyed after the request!

Assuming that your Seam application only uses event, application, or stateless components, this procedure prevents exhaustion of available HTTP sessions on the server. The RESTEasy integration with Seam assumes by default that sessions are not used, hence anemic sessions would add up as every REST request would start a session that will only be removed when timed out.

If your RESTful Seam application has to preserve session state across REST HTTP requests, disable this behavior in your configuration file:

```
<resteasy:application destroy-session-after-request="false"/>
```

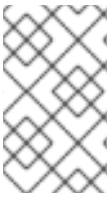
Every REST HTTP request will now create a new session that will only be removed by timeout or explicit invalidation in your code through **Session.instance().invalidate()**. It is your responsibility to pass a valid session identifier along with your HTTP requests, if you want to utilize the session context across requests.

Conversation-scoped resource components and conversation mapping are not currently supported, but are planned for future versions of Seam.

Provider classes can also be Seam components. They must be either application-scoped or stateless.

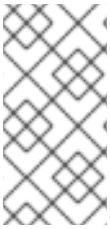
Resources and providers can be EJBs or JavaBeans, like any other Seam component.

EJB Seam components are supported as REST resources. Always annotate the local business interface, not the EJB implementation class, with JAX-RS annotations. The EJB has to be **STATELESS**.



NOTE

RESTEasy components do not support hot redeployment. As a result, the components should never be placed in the **src/hot** folder. The **src/main** folder should be used instead.



NOTE

Sub-resources as defined in the JAX RS specification, section 3.4.1, can not be Seam component instances at this time. Only root resource classes can be registered as Seam components. In other words, do not return a Seam component instance from a root resource method.

23.4.3. Securing resources

You can enable the Seam authentication filter for HTTP Basic and Digest authentication in **`components.xml`**:

```
<web:authentication-filter url-pattern="/seam/resource/rest/*" auth-
type="basic"/>
```

See the Seam security chapter on how to write an authentication routine.

After successful authentication, authorization rules with the common **`@Restrict`** and **`@PermissionCheck`** annotations are in effect. You can also access the client **`Identity`**, work with permission mapping, and so on. All regular Seam security features for authorization are available.

23.4.4. Mapping exceptions to HTTP responses

Section 3.3.4 of the JAX-RS specification defines how JAX RS handles checked and unchecked exceptions. Integrating RESTEasy with Seam allows you to map exceptions to HTTP response codes within Seam's **`pages.xml`**. If you use **`pages.xml`** already, this is easier to maintain than many JAX RS exception mapper classes.

For exceptions to be handled within Seam, the Seam filter must be executed for your HTTP request. You must filter all requests in your **`web.xml`**, *not* as a request URI pattern that does not cover your REST requests. The following example intercepts all HTTP requests and enables Seam exception handling:

```
<filter>
    <filter-name>Seam Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Seam Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

To convert the unchecked **`UnsupportedOperationException`** thrown by your resource methods to a **`501 Not Implemented`** HTTP status response, add the following to your **`pages.xml`** descriptor:

```
<exception class="java.lang.UnsupportedOperationException">
    <http-error error-code="501">
        <message>The requested operation is not supported</message>
    </http-error>
</exception>
```

Custom or checked exceptions are handled in the same way:

```
<exception class="my.CustomException" log="false">
```

```
    <http-error error-code="503">
        <message>Service not available: #
{org.jboss.seam.handledException.message}</message>
    </http-error>
</exception>
```

You do not have to send a HTTP error to the client if an exception occurs. Seam lets you map the exception as a redirect to a view of your Seam application. Since this feature is typically used for human clients (web browsers) and not for REST API remote clients, you should pay attention to conflicting exception mappings in **`pages.xml`**.

The HTTP response does pass through the servlet container, so an additional mapping may apply if you have **`<error-page>`** mappings in your **`web.xml`** configuration. The HTTP status code would then be mapped to a rendered HTML error page with status **`200 OK`**.

23.4.5. Exposing entities via RESTful API

Seam makes it really easy to use a RESTful approach for accessing application data. One of the improvements that Seam introduces is the ability to expose parts of your SQL database for remote access via plain HTTP calls. For this purpose, the Seam/RESTEasy integration module provides two components: **`ResourceHome`** and **`ResourceQuery`**, which benefit from the API provided by the Seam Application Framework (Chapter 13, *The Seam Application Framework*). These components allow you to bind domain model entity classes to an HTTP API.

23.4.5.1. ResourceQuery

ResourceQuery exposes entity querying capabilities as a RESTful web service. By default, a simple underlying Query component, which returns a list of instances of a given entity class, is created automatically. Alternatively, the ResourceQuery component can be attached to an existing Query component in more sophisticated cases. The following example demonstrates how easily ResourceQuery can be configured:

```
<resteasy:resource-query
   path="/user"
   name="userResourceQuery"
   entity-class="com.example.User"/>
```

With this single XML element, a ResourceQuery component is set up. The configuration is straightforward:

- The component will return a list of **`com.example.User`** instances.
- The component will handle HTTP requests on the URI path **`/user`**.
- The component will by default transform the data into XML or JSON (based on client's preference). The set of supported mime types can be altered by using the **`media-types`** attribute, for example:

```
<resteasy:resource-query
   path="/user"
   name="userResourceQuery"
   entity-class="com.example.User"
   media-types="application/fastinfoset"/>
```

Alternatively, if you do not like configuring components using XML, you can set up the component by extension:

```
@Name("userResourceQuery")
@Path("user")
public class UserResourceQuery extends ResourceQuery<User>
{
}
```

Queries are read-only operations, the resource only responds to GET requests. Furthermore, ResourceQuery allows clients of a web service to manipulate the resultset of a query using the following path parameters:

Parameter name	Example	Description
start	/user?start=20	Returns a subset of a database query result starting with the 20th entry.
show	/user?show=10	Returns a subset of the database query result limited to 10 entries.

For example, you can send an HTTP GET request to **`/user?start=30&show=10`** to get a list of entries representing 10 rows starting with row 30.

NOTE

RESTEasy uses JAXB to marshall entities. Thus, in order to be able to transfer them over the wire, you need to annotate entity classes with **`@XMLRootElement`**. Consult the JAXB and RESTEasy documentation for more information.

23.4.5.2. ResourceHome

Just as ResourceQuery makes Query's API available for remote access, so does ResourceHome for the Home component. The following table describes how the two APIs (HTTP and Home) are bound together.

Table 23.1. Bindings in ResourceHome

HTTP method	Path	Function	ResourceHome method
GET	{path}/{id}	Read	getResource()
POST	{path}	Create	postResource()
PUT	{path}/{id}	Update	putResource()
DELETE	{path}/{id}	Delete	deleteResource()

- You can GET, PUT, and DELETE a particular user instance by sending HTTP requests to /user/{userId}
- Sending a POST request to **`/user`** creates a new user entity instance and persists it. Usually, you leave it up to the persistence layer to provide the entity instance with an identifier value and thus an URI. Therefore, the URI is sent back to the client in the **`Location`** header of the HTTP response.

The configuration of ResourceHome is very similar to ResourceQuery except that you need to explicitly specify the underlying Home component and the Java type of the entity identifier property.

```
<resteasy:resource-home
   path="/user"
   name="userResourceHome"
   entity-home="#{userHome}"
   entity-id-class="java.lang.Integer"/>
```

Again, you can write a subclass of ResourceHome instead of XML:

```
@Name("userResourceHome")
@Path("user")
public class UserResourceHome extends ResourceHome<User, Integer>
{

   @In
   private EntityHome<User> userHome;

   @Override
   public Home<?, User> getEntityHome()
   {
      return userHome;
   }
}
```

For more examples of ResourceHome and ResourceQuery components, take a look at the *Seam Tasks* example application, which demonstrates how Seam/RESTEasy integration can be used together with a jQuery web client. In addition, you can find more code example in the *Restbay* example, which is used mainly for testing purposes.

23.4.6. Testing resources and providers

Seam includes a unit testing utility class that helps you create unit tests for a RESTful architecture. Extend the **`SeamTest`** class as usual and use the **`ResourceRequestEnvironment.ResourceRequest`** to emulate HTTP requests/response cycles:

```
import org.jboss.seam.mock.ResourceRequestEnvironment;
import org.jboss.seam.mock.EnhancedMockHttpServletRequest;
import org.jboss.seam.mock.EnhancedMockHttpServletResponse;
import static
org.jboss.seam.mock.ResourceRequestEnvironment.ResourceRequest;
import static org.jboss.seam.mock.ResourceRequestEnvironment.Method;

public class MyTest extends SeamTest {

   ResourceRequestEnvironment sharedEnvironment;
```

```
   @BeforeClass
   public void prepareSharedEnvironment() throws Exception {
       sharedEnvironment = new ResourceRequestEnvironment(this) {
            @Override
            public Map<String, Object> getDefaultHeaders() {
               return new HashMap<String, Object>() {{
                   put("Accept", "text/plain");
               }};
            }
         };
   }

   @Test
   public void test() throws Exception
   {
      //Not shared: new ResourceRequest(new
ResourceRequestEnvironment(this), Method.GET, "/my/relative/uri)

      new ResourceRequest(sharedEnvironment, Method.GET,
"/my/relative/uri)
      {
         @Override
         protected void prepareRequest(EnhancedMockHttpServletRequest
request)
         {
            request.addQueryParameter("foo", "123");
            request.addHeader("Accept-Language", "en_US, de");
         }

         @Override
         protected void onResponse(EnhancedMockHttpServletResponse
response)
         {
            assert response.getStatus() == 200;
            assert response.getContentAsString().equals("foobar");
         }

      }.run();
   }
}
```

This test only executes local calls, it does not communicate with the **`SeamResourceServlet`** through TCP. The mock request is passed through the Seam servlet and filters and the response is then available for test assertions. Overriding the **`getDefaultHeaders()`** method in a shared instance of **`ResourceRequestEnvironment`** allows you to set request headers for every test method in the test class.

Note that a **`ResourceRequest`** has to be executed in a **`@Test`** method or in a **`@BeforeMethod`** callback. You can not execute it in any other callback, such as **`@BeforeClass`**.

CHAPTER 24. REMOTING

Seam uses Asynchronous JavaScript and XML (AJAX) to remotely access components from a web page. The framework for this functionality requires very little development effort — you can make your components AJAX-accessible with simple annotations. This chapter describes the steps required to build an AJAX-enabled web page, and explains the Seam Remoting framework in further detail.

24.1. CONFIGURATION

To use remoting, you must first configure your Seam Resource Servlet in your **web.xml** file:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.servlet.SeamResourceServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

Next, import the necessary JavaScript into your web page. A minimum of two scripts must be imported. The first contains all client-side framework code, which enables remoting functionality:

```
<script type="text/javascript"
        src="seam/resource/remoting/resource/remote.js">
</script>
```

The second contains the stubs and type definitions for the components you wish to call. This is generated dynamically, based on the local interface of your components, and includes type definitions for all classes that can be used to call the remotable methods of the interface. The script name reflects your component name. For example, if you annotate a stateless session bean with **@Name("customerAction")**, your script tag should look like this:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction">
</script>
```

If you want to access more than one component from the same page, include them all as parameters of your script tag:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?
customerAction&accountAction">
</script>
```

You can also use the **s:remote** tag to import the required JavaScript. Separate each component or class name that you want to import with a comma:

```
<s:remote include="customerAction,accountAction"/>
```

24.2. THE SEAM OBJECT

Client-side component interaction is performed with the **Seam** JavaScript object defined in **remote.js**. This is used to make asynchronous calls against your component. It is split into two areas of functionality: **Seam.Component** contains methods for working with components and **Seam.Remoting** contains methods for executing remote requests. The easiest way to become familiar with this object is to start with a simple example.

24.2.1. A Hello World example

Procedure 24.1. Hello World Example

1. To show you how the **Seam** object works, we will first create a new Seam component called **helloAction**:

```
@Stateless
@Name("helloAction")
public class HelloAction implements HelloLocal {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

2. We will also need to create a local interface for our new component. In particular, note the **@WebRemote** annotation, as this is required to make our method accessible via remoting:

```
@Local
public interface HelloLocal {
    @WebRemote
    public String sayHello(String name);
}
```

3. This is all the server-side code we require. Next, create a new web page and import the **helloAction** component:

```
<s:remote include="helloAction"/>
```

4. Add a button to the page to make this an interactive user experience:

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

5. You will also need script that performs an action when the button is clicked:

```
<script type="text/javascript">
function sayHello() {
     var name = prompt("What is your name?");
     Seam.Component.getInstance("helloAction").sayHello(name,
sayHelloCallback);
}

function sayHelloCallback(result) {
```

```
    alert(result);
}
</script>
```

6. Now deploy your application and browse to your page. Click the button, and enter a name when prompted. A message box will display the "Hello" message, confirming the call's success. (You can find the full source code for this Hello World example in Seam's **`/examples/remoting/helloworld`** directory.)

You can see from the JavaScript code listing that we have implemented two methods. The first method prompts the user for their name, and makes a remote request. Look at the following line:

```
Seam.Component.getInstance("helloAction").sayHello(name,
sayHelloCallback);
```

The first section of this line (**`Seam.Component.getInstance("helloAction")`**) returns a proxy, or *stub*, for our **`helloAction`** component. The remainder of the line (**`sayHello(name,sayHelloCallback);`**) invokes our component methods against the stub.

The whole line invokes the **`sayHello`** method of our component, passing in **`name`** as a parameter. The second parameter, **`sayHelloCallback`**, is not a parameter of our component's **`sayHello`** method — it tells the Seam Remoting framework that, once a response to the request is received, the response should be passed to the **`sayHelloCallback`** JavaScript method. (This callback parameter is optional; you can leave it out if you are calling a method with a **`void`** return type, or if the result of the request is not important.)

When the **`sayHelloCallback`** method receives the response to our remote request, it displays an alert message with the result of our method call.

24.2.2. Seam.Component

The **`Seam.Component`** JavaScript object provides a number of client-side methods for working with your Seam components. The two main methods, **`newInstance()`** and **`getInstance()`** are documented more thoroughly in the sections following. The main difference between them is that **`newInstance()`** will always create a new instance of a component type, and **`getInstance()`** will return a singleton instance.

24.2.2.1. Seam.Component.newInstance()

Use this method to create a new instance of an entity or JavaBean component. The object returned will have the same getter/setter methods as its server-side counterpart. You can also access its fields directly. For example:

```
@Name("customer")
@Entity
public class Customer implements Serializable
{
    private Integer customerId;
    private String firstName;
    private String lastName;

    @Column public Integer getCustomerId() {
        return customerId;
    }
```

```
    public void setCustomerId(Integer customerId} {
        this.customerId = customerId;
    }

    @Column public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

To create a client-side Customer you would write the following code:

```
var customer = Seam.Component.newInstance("customer");
```

From here, you can set the fields of the customer object.

```
customer.setFirstName("John"); // Or you can set the fields directly
// customer.lastName = "Smith";
```

24.2.2.2. Seam.Component.getInstance()

The **`getInstance()`** method is used to refer to a Seam session bean component stub, which can then be used to remotely execute methods against your component. This method returns a singleton for the specified component, so calling it twice in a row with the same component name will return the same instance of the component.

To continue the previous example, if we have created a new **`customer`** and we want to save it, we pass it to the **`saveCustomer()`** method of our **`customerAction`** component:

```
Seam.Component.getInstance("customerAction").saveCustomer( customer);
```

24.2.2.3. Seam.Component.getComponentName()

Passing an object into this method returns the component name, if it is a component, or **`null`** if it is not.

```
if (Seam.Component.getComponentName(instance) == "customer")
    alert("Customer");
else if (Seam.Component.getComponentName(instance) == "staff")
    alert("Staff member");
```

24.2.3. Seam.Remoting

Most of the client side functionality for Seam Remoting is held within the **`Seam.Remoting`** object. You should not need to directly call many of its methods, but there are several that are useful:

24.2.3.1. Seam.Remoting.createType()

If your application contains or uses JavaBean classes that are not Seam components, you may need to create these types on the client side to pass as parameters into your component method. Use the **`createType()`** method to create an instance of your type. Pass in the fully-qualified Java class name as a parameter:

```
var widget = Seam.Remoting.createType("com.acme.widgets.MyWidget");
```

24.2.3.2. Seam.Remoting.getTypeName()

This method is the non-component equivalent of **`Seam.Component.getComponentName()`**. It returns the name of the type for an object instance, or **`null`** if the type is not known. The name is the fully-qualified name of the type's Java class.

24.3. EVALUATING EL EXPRESSIONS

Seam Remoting also supports EL expression evaluation, which is another convenient method of retrieving data from the server. The **`Seam.Remoting.eval()`** function lets the EL expression be remotely evaluated on the server, and returns the resulting value to a client-side callback method. This function accepts two parameters: the EL expression to evaluate, and the callback method to invoke with the expression value. For example:

```
function customersCallback(customers) {
    for (var i = 0; i < customers.length; i++) {
        alert("Got customer: " + customers[i].getName());
    }
}

Seam.Remoting.eval("#{customers}", customersCallback);
```

Here, Seam evaluates the **`#{customers}`** expression, and the value of the expression (in this case, a list of **`Customer`** objects) is returned to the **`customersCallback()`** method. Remember, objects returned this way must have their types imported with **`s:remote`** for you to work with them in JavaScript. To work with a list of **`customer`** objects, you must be able to import the **`customer`** type:

```
<s:remote include="customer"/>
```

24.4. CLIENT INTERFACES

In the previous configuration section, the stub for our component is imported into our page with either **`seam/resource/remoting/interface.js`**, or with the **`s:remote`** tag:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction">
</script>
```

```
<s:remote include="customerAction"/>
```

Including this script generates the interface definitions for our component, plus any other components or types required to execute the methods of our component, and makes them available for the remoting framework's use.

Two types of stub can be generated: *executable* stubs, and *type* stubs. Executable stubs are behavioral, and execute methods against your session bean components. Type stubs contain state, and represent the types that can be passed in as parameters or returned as results.

The type of stub that is generated depends upon the type of your Seam component. If the component is a session bean, an executable stub will be generated. If it is an entity or JavaBean, a type stub will be generated. However, if your component is a JavaBean and any of its methods are annotated with **`@WebRemote`**, an executable stub will be generated. This lets you call your JavaBean component's methods in a non-EJB environment, where you do not have access to session beans.

24.5. THE CONTEXT

The Seam Remoting Context contains additional information that is sent and received as part of a remoting request or response cycle. At this point, it contains only the conversation ID, but may be expanded in future.

24.5.1. Setting and reading the Conversation ID

If you intend to use remote calls within a conversation's scope, then you must be able to read or set the conversation ID in the Seam Remoting context. To read the conversation ID after making a remote request, call **`Seam.Remoting.getContext().getConversationId()`**. To set the conversation ID before making a request, call **`Seam.Remoting.getContext().setConversationId()`**.

If the conversation ID has not been explicitly set with **`Seam.Remoting.getContext().setConversationId()`**, then the first valid conversation ID returned by any remoting call is assigned automatically. If you are working with multiple conversations within your page, you may need to set your conversation ID explicitly before each call. Single conversations do not require explicit ID setting.

24.5.2. Remote calls within the current conversation scope

Under some circumstances, you may need to make a remote call within the scope of the current view's conversation. To do so, you must explicitly set the conversation ID to that of the view before making the remote call. The following JavaScript will set the conversation ID being used for remote calls to the current view's conversation ID:

```
Seam.Remoting.getContext().setConversationId( #{conversation.id} );
```

24.6. BATCH REQUESTS

Seam Remoting lets you execute multiple component calls with a single request. We recommend using this feature when you need to reduce network traffic.

The **`Seam.Remoting.startBatch()`** method starts a new batch. Any component calls executed after starting a batch are queued, rather than being sent immediately. When all the desired component calls have been added to the batch, the **`Seam.Remoting.executeBatch()`** method sends a single request

containing all of the queued calls to the server, where they will be executed in order. After the calls have been executed, a single response containing all return values is returned to the client, and the callback functions are triggered in their execution order.

If you begin a batch, and then decide you do not want to send it, the **`Seam.Remoting.cancelBatch()`** method discards any queued calls and exits the batch mode.

For an example of batch use, see **`/examples/remoting/chatroom`**.

24.7. WORKING WITH DATA TYPES

24.7.1. Primitives / Basic Types

This section describes the support for basic data types. On the server side, these values are generally compatible with either their primitive type, or their corresponding wrapper class.

24.7.1.1. String

Use JavaScript String objects to set String parameter values.

24.7.1.2. Number

Seam supports all Java-supported number types. On the client side, number values are always serialized as their String representation. They are converted to the correct destination type on the server side. Conversion into either a primitive or wrapper type is supported for **`Byte`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** and **`Short`** types.

24.7.1.3. Boolean

Booleans are represented client-side by JavaScript Boolean values, and server-side by a Java Boolean.

24.7.2. JavaBeans

In general, these are either Seam entity or JavaBean components, or some other non-component class. Use the appropriate method to create a new instance of the object — **`Seam.Component.newInstance()`** for Seam components, or **`Seam.Remoting.createType()`** for anything else.

Only objects created by either of these two methods should be used as parameter values, where the parameter is not one of the preexisting valid types. You may encounter component methods where the exact parameter type cannot be determined, such as:

```
@Name("myAction")
public class MyAction implements MyActionLocal {
    public void doSomethingWithObject(Object obj) {
        // code
    }
}
```

In this case, the interface for **`myAction`** will not include **`myWidget`**, because it is not directly referenced by any of its methods. Therefore, you cannot pass in an instance of your **`myWidget`** component unless you import it explicitly:

```
<s:remote include="myAction,myWidget"/>
```

This allows a **myWidget** object to be created with **Seam.Component.newInstance("myWidget")**, which can then be passed to **myAction.doSomethingWithObject()**.

24.7.3. Dates and Times

Date values are serialized into a String representation that is accurate to the millisecond. On the client side, use a JavaScript Date object to work with date values. On the server side, use any **java.util.Date** class (or a descendant class, such as **java.sql.Date** or **java.sql.Timestamp**.)

24.7.4. Enums

On the client side, enums are treated similarly to Strings. When setting the value for an enum parameter, use the String representation of the enum. Take the following component as an example:

```
@Name("paintAction")
public class paintAction implements paintLocal {
    public enum Color {red, green, blue, yellow, orange, purple};
    public void paint(Color color) {
        // code
    }
}
```

To call the **paint()** method with the color **red**, pass the parameter value as a String literal:

```
Seam.Component.getInstance("paintAction").paint("red");
```

The inverse is also true. That is, if a component method returns an enum parameter (or contains an enum field anywhere in the returned object graph), then on the client-side it will be represented as a String.

24.7.5. Collections

24.7.5.1. Bags

Bags cover all collection types, including arrays, collections, lists, and sets, but excluding maps — see the section following. They are implemented client-side as a JavaScript array, both when called and returned. The remoting framework on the server side can convert the bag to an appropriate type for the component method call.

24.7.5.2. Maps

The Seam Remoting framework provides simple map support where no native support is available in JavaScript. To create a map that can be used as a parameter to a remote call, create a new **Seam.Remoting.Map** object:

```
var map = new Seam.Remoting.Map();
```

This JavaScript implementation provides basic methods for working with Maps: **size()**, **isEmpty()**, **keySet()**, **values()**, **get(key)**, **put(key, value)**, **remove(key)** and **contains(key)**. Each of these methods is equivalent to the Java method of the same name. Where the method returns a

collection, as in **`keySet()`** and **`values()`**, a JavaScript array object will be returned that contains the key or value objects (respectively).

24.8. DEBUGGING

To help you track down bugs, you can enable a debug mode, which displays the contents of all packets sent between client and server in a pop-up window. To enable debug mode, either execute the **`setDebug()`** method in JavaScript, like so:

```
Seam.Remoting.setDebug(true);
```

Or configure it in **`components.xml`**:

```
<remoting:remoting debug="true"/>
```

To turn off debug mode, call **`setDebug(false)`**. If you want to write your own messages to the debug log, call **`Seam.Remoting.log(message)`**.

24.9. HANDLING EXCEPTIONS

When invoking a remote component method, you can specify an exception handler to process the response in the event of an exception during component invocation. To specify an exception handler function, include a reference to it after the callback parameter in your JavaScript:

```
var callback = function(result) {
    alert(result);
};
var exceptionHandler = function(ex) {
    alert("An exception occurred: " + ex.getMessage());
};
Seam.Component.getInstance("helloAction")
    .sayHello(name, callback, exceptionHandler);
```

If you do not have a callback handler defined, you must specify **`null`** in its place:

```
var exceptionHandler = function(ex) {
    alert("An exception occurred: " + ex.getMessage());
};
Seam.Component.getInstance("helloAction")
    .sayHello(name, null, exceptionHandler);
```

The exception object that is passed to the exception handler exposes one method, **`getMessage()`**, which returns the exception message belonging to the exception thrown by the **`@WebRemote`** method.

24.10. THE LOADING MESSAGE

You can modify, define custom rendering for, or even remove the default loading message that appears in the top right corner of the screen.

24.10.1. Changing the message

To change the message from the default "Please Wait...", set the value of **`Seam.Remoting.loadingMessage`**:

```
Seam.Remoting.loadingMessage = "Loading...";
```

24.10.2. Hiding the loading message

To completely suppress the display of the loading message, override the implementation of **`displayLoadingMessage()`** and **`hideLoadingMessage()`** with actionless functions:

```
// do not display the loading indicator
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

24.10.3. A Custom Loading Indicator

It is also possible to override the loading indicator to display an animated icon, or anything else that you want. To do so, override the **`displayLoadingMessage()`** and **`hideLoadingMessage()`** messages with your own implementations:

```
Seam.Remoting.displayLoadingMessage = function() {
    // Write code here to display the indicator
};
Seam.Remoting.hideLoadingMessage = function() {
    // Write code here to hide the indicator
};
```

24.11. CONTROLLING WHAT DATA IS RETURNED

When a remote method is executed, the result is serialized into an XML response, which is returned to the client. This response is then unmarshaled by the client into a JavaScript object. For complex types (such as JavaBeans) that include references to other objects, all referenced objects are also serialized as part of the response. These objects can reference other objects, which can reference other objects, and so on — so, if left unchecked, this object "graph" can be enormous.

For this reason, and to prevent sensitive information being exposed to the client, Seam Remoting lets you constrain the object graph by specifying the **`exclude`** field of the remote method's **`@WebRemote`** annotation. This field accepts a String array containing one or more paths specified with dot notation. When invoking a remote method, the objects in the result's object graph that match these paths are excluded from the serialized result packet.

The examples that follow are all based on this **`Widget`** class:

```
@Name("widget")
public class Widget {
    private String value;
    private String secret;
    private Widget child;
    private Map&lt;String,Widget&gt; widgetMap;
    private List&lt;Widget&gt; widgetList;

    // getters and setters for all fields
```

24.11.1. Constraining normal fields

If your remote method returns an instance of **`Widget`**, but you do not want to expose the **`secret`** field because it contains sensitive information, you would constrain it like so:

```
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

The value "secret" refers to the **`secret`** field of the returned object.

Now, note that the returned **`Widget`** value has a field **`child`** that is also a **`Widget`**. If we want to hide the **`child`**'s **`secret`** value, rather than the field itself, we can use dot notation to specify this field's path within the result object's graph:

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

24.11.2. Constraining Maps and Collections

Objects within an object graph can also exist in a **`Map`** or a Collection (that is, a **`List`**, a **`Set`**, an **`Array`**, etc.). Collections are treated like any other field — for example, if our **`Widget`** contained a list of other **`Widget`**s in its **`widgetList`** field, we would constrain the **`secret`** field of the **`Widget`**s in this list with the following notation:

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

To constrain a **`Map`**'s key or value, the notation is slightly different. Appending **`[key]`** after the **`Map`**'s field name constrains the **`Map`**'s key object values, while **`[value]`** constrains the value object values. The following example demonstrates how the values of the **`widgetMap`** field have their **`secret`** field constrained:

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

24.11.3. Constraining objects of a specific type

You can use square brackets to constrain the fields of an object type regardless of its location in the object graph. If the object is a Seam component, use the name of the component; if not, use the fully-qualified class name, like so:

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

24.11.4. Combining Constraints

Constraints can also be combined to filter objects from multiple paths within the object graph:

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

24.12. TRANSACTIONAL REQUESTS

By default, no transaction is active during a remoting request. If you wish to update the database during a remoting request, you must annotate the **@WebRemote** method with **@Transactional**, like so:

```
@WebRemote
@Transactional(TransactionPropagationType.REQUIRED)
    public void updateOrder(Order order) {
    entityManager.merge(order);
}
```

24.13. JMS MESSAGING

Seam Remoting provides experimental support for JMS Messaging. This section describes currently-implemented JMS support. Note that this may change in the future. At present, we do not recommend using this feature within a production environment.

24.13.1. Configuration

Before you can subscribe to a JMS topic, you must first configure a list of the topics that Seam Remoting can subscribe to. List the topics under **org.jboss.seam.remoting.messaging.subscriptionRegistry. allowedTopics** in **seam.properties**, **web.xml** or **components.xml**:

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

24.13.2. Subscribing to a JMS Topic

The following example demonstrates how to subscribe to a JMS Topic:

```
function subscriptionCallback(message) {
    if (message instanceof Seam.Remoting.TextMessage)
        alert("Received message: " + message.getText());
}
Seam.Remoting.subscribe("topicName", subscriptionCallback);
```

The **Seam.Remoting.subscribe()** method accepts two parameters: the name of the JMS topic to subscribe to, and the callback function to invoke when a message is received.

Two message types are supported: Text messages, and Object messages. To test for the message type that is passed to your callback function, use the **instanceof** operator. This tests whether the message is a **Seam.Remoting.TextMessage** or **Seam.Remoting.ObjectMessage**. A **TextMessage** contains the text value in its **text** field. (You can also fetch this value by calling the object's **getText()** method.) An **ObjectMessage** contains its object value in its **value** field. (You can also fetch this value by calling the object's **getValue()** method.)

24.13.3. Unsubscribing from a Topic

To unsubscribe from a topic, call **Seam.Remoting.unsubscribe()** and pass in the topic name:

```
Seam.Remoting.unsubscribe("topicName");
```

24.13.4. Tuning the Polling Process

Polling can be controlled and modified with two parameters.

`Seam.Remoting.pollInterval` controls how long to wait between subsequent polls for new messages. This parameter is expressed in seconds, and its default setting is **`10`**.

`Seam.Remoting.pollTimeout` is also expressed in seconds. It controls how long a request to the server should wait for a new message before timing out and sending an empty response. Its default is **`0`** seconds, which means that when the server is polled, if there are no messages ready for delivery, an empty response will be immediately returned.

Use caution when setting a high **`pollTimeout`** value. Each request that has to wait for a message uses a server thread until either the message is received, or the request times out. If many such requests are served simultaneously, a large number of server threads will be used.

We recommend setting these options in **`components.xml`**, but they can be overridden with JavaScript if desired. The following example demonstrates a more aggressive polling method. Set these parameters to values that suit your application:

In **`components.xml`**:

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

With Java:

```
// Only wait 1 second between receiving a poll response and sending
// the next poll request.
Seam.Remoting.pollInterval = 1;
// Wait up to 5 seconds on the server for new messages
Seam.Remoting.pollTimeout = 5;
```

CHAPTER 25. SEAM AND THE GOOGLE WEB TOOLKIT

WARNING

Technology Preview features are not fully supported under Red Hat subscription level agreements (SLAs), may not be functionally complete, and are not intended for production use. However, these features provide early access to upcoming product innovations, enabling customers to test functionality and provide feedback during the development process. As Red Hat considers making future iterations of Technology Preview features generally available, we will provide commercially reasonable efforts to resolve any reported issues that customers experience when using these features.

If you prefer to develop dynamic AJAX (Asynchronous Java and XML) applications with the Google Web Toolkit (GWT), Seam provides an integration layer that allows GWT widgets to interact directly with Seam components.

In this section, we assume you are already familiar with GWT Tools, and focus only on the Seam integration. You can find more information at http://code.google.com/webtoolkit/ .

25.1. CONFIGURATION

You do not need to make any configuration changes to use GWT in a Seam application — all you need to do is install the Seam Resource Servlet. See Chapter 28, *Configuring Seam and packaging Seam applications* for details.

25.2. PREPARING YOUR COMPONENT

To prepare a Seam component to be called with GWT, you must first create both synchronous and asynchronous service interfaces for the methods you wish to call. Both interfaces should extend the GWT interface **`com.google.gwt.user.client.rpc.RemoteService`**:

```
public interface MyService extends RemoteService {
  public String askIt(String question);
}
```

The asynchronous interface should be identical, except for an additional **`AsyncCallback`** parameter for each of the methods it declares:

```
public interface MyServiceAsync extends RemoteService {
  public void askIt(String question, AsyncCallback callback);
}
```

The asynchronous interface (in this case, **`MyServiceAsync`**) is implemented by GWT, and should never be implemented directly.

The next step is to create a Seam component that implements the synchronous interface:

```
@Name("org.jboss.seam.example.remoting.gwt.client.MyService")
public class ServiceImpl implements MyService {

  @WebRemote
  public String askIt(String question) {

    if (!validate(question)) {
      throw new IllegalStateException("Hey, this should not happen, " +
                                      "I checked on the client, but " +
                                      "it's always good to double
check.");
    }
    return "42. Its the real question that you seek now.";
  }

  public boolean validate(String q) {
    ValidationUtility util = new ValidationUtility();
    return util.isValid(q);
  }
}
```

The Seam component's name must match the fully-qualified name of the GWT client interface (as shown), or the Seam Resource Servlet will not be able to find it when a client makes a GWT call. Methods that GWT will make accessible must be annotated with **`@WebRemote`**.

25.3. HOOKING UP A GWT WIDGET TO THE SEAM COMPONENT

Next, write a method that returns the asynchronous interface to the component. This method can be located inside the widget class, and will be used by the widget to obtain a reference to the asynchronous client stub:

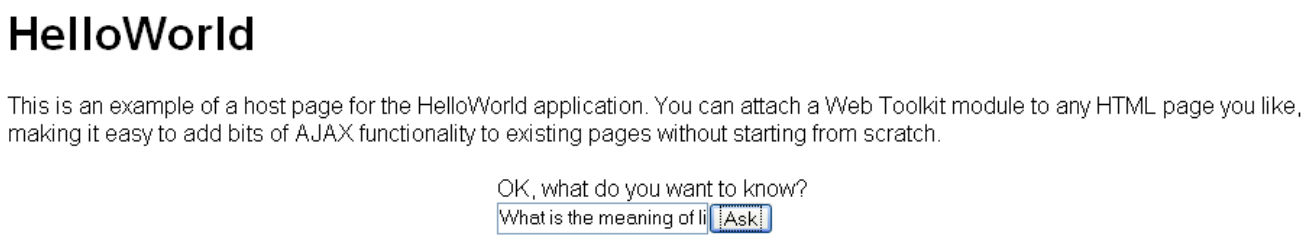
```
private MyServiceAsync getService() {
  String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

  MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
  ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
  return svc;
}
```

Finally, write the widget code that invokes the method on the client stub. The following example creates a simple user interface with a label, text input, and a button:

```
public class AskQuestionWidget extends Composite {
  private AbsolutePanel panel = new AbsolutePanel();

  public AskQuestionWidget() {
    Label lbl = new Label("OK, what do you want to know?");
    panel.add(lbl);
    final TextBox box = new TextBox();
    box.setText("What is the meaning of life?");
    panel.add(box);
    Button ok = new Button("Ask");

    ok.addClickListener(new ClickListener() {
```

```
      public void onClick(Widget w) {
        ValidationUtility valid = new ValidationUtility();
        if (!valid.isValid(box.getText())) {
          Window.alert("A question has to end with a '?'");
        } else {
          askServer(box.getText());
        }
      }
    });
    panel.add(ok);

    initWidget(panel);
  }

  private void askServer(String text) {
    getService().askIt(text, new AsyncCallback() {
    public void onFailure(Throwable t) {
      Window.alert(t.getMessage());
    }

    public void onSuccess(Object data) {
      Window.alert((String) data);
    }
  });
}

...
```

When clicked, this button invokes the **`askServer()`** method, passing the contents of the input text. In this example, it also validates that the input is a valid question. The **`askServer()`** method acquires a reference to the asynchronous client stub (returned by the **`getService()`** method) and invokes the **`askIt()`** method. The result (or error message, if the call fails) is shown in an alert window.



The complete code for this example can be found in the Seam distribution in the **`examples/remoting/gwt`** directory.

25.4. GWT ANT TARGETS

To deploy GWT applications, you must also perform compilation to JavaScript. This compacts and obfuscates the code. You can use an Ant utility instead of the command line or GUI utility provided by GWT. To do so, you must have GWT downloaded, and the Ant task **`JAR`** in your Ant classpath.

Place the following near the top of your Ant file:

```
<taskdef uri="antlib:de.samaflost.gwttasks"
         resource="de/samaflost/gwttasks/antlib.xml"
```

```
          classpath="./lib/gwttasks.jar"/>
<property file="build.properties"/>
```

Create a **build.properties** file containing:

```
gwt.home=/gwt_home_dir
```

This must point to the directory in which GWT is installed. Next, create a target:

```
<!-- the following are handy utilities for doing GWT development.
     To use GWT, you will of course need to download GWT seperately -->

<target name="gwt-compile">
  <!-- in this case, we are "re homing" the gwt generated stuff, so
       in this case we can only have one GWT module - we are doing this
       deliberately to keep the URL short -->
  <delete>
    <fileset dir="view"/>
  </delete>
  <gwt:compile outDir="build/gwt"
               gwtHome="${gwt.home}"
               classBase="${gwt.module.name}"
               sourceclasspath="src"/>
    <copy todir="view">
      <fileset dir="build/gwt/${gwt.module.name}"/>
    </copy>
</target>
```

When called, this target compiles the GWT application and copies it to the specified directory (likely in the **webapp** section of your WAR).

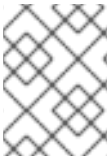
NOTE

Never edit the code generated by **gwt-compile** — if you need to edit, do so in the GWT source directory.

We highly recommend using the hosted mode browser included in the GWT if you plan to develop applications with the GWT.

CHAPTER 26. SPRING FRAMEWORK INTEGRATION

The Spring Framework is part of the Seam inversion-of-control (IoC) module. It allows easy migration of Spring-based projects to Seam, and benefits Spring applications with Seam features, such as conversations and a more sophisticated persistence context management.



NOTE

The Spring integration code is included in the **`jboss-seam-ioc`** library. This library is a required dependency for all Seam-Spring integration techniques covered in this chapter.

Seam's support for Spring gives you:

- Seam component injection into Spring beans,
- Spring bean injection into Seam components,
- Spring bean to Seam component transformation,
- the ability to place Spring beans in any Seam context,
- the ability to start a spring WebApplicationContext with a Seam component,
- support for using Spring PlatformTransactionManagement with your Seam-based applications,
- support for using a Seam-managed replacement for Spring's **`OpenEntityManagerInViewFilter`** and **`OpenSessionInViewFilter`**, and
- support for backing **`@Asynchronous`** calls with Spring **`TaskExecutors`**.

26.1. INJECTING SEAM COMPONENTS INTO SPRING BEANS

Inject Seam component instances into Spring beans with the **`<seam:instance/>`** namespace handler. To enable the Seam namespace handler, the Seam namespace must first be added to the Spring beans definition file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:seam="http://jboss.com/products/seam/spring-seam"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
         "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd
          http://jboss.com/products/seam/spring-seam
          http://jboss.com/products/seam/spring-seam-2.2.xsd">
```

Any Seam component can now be injected into any Spring bean, like so:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="someComponent"/>
  </property>
</bean>
```

You can use an EL expression instead of a component name:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="#{someExpression}"/>
  </property>
</bean>
```

You can inject a Seam component instance into a Spring bean by using a Spring bean ID, like so:

```
<seam:instance name="someComponent" id="someSeamComponentInstance"/>

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty" ref="someSeamComponentInstance">
</bean>
```

However, Spring, unlike Seam, was not designed to support a stateful component model with multiple contexts. Spring injection does not occur at method invocation time, but when the Spring bean is instantiated.

The instance available when the bean is instantiated will be used for the entire life of the bean. Say you inject a Seam conversation-scoped component instance directly into a singleton Spring bean — that singleton will hold a reference to the same instance long after the conversation is over. This is called *scope impedance*.

Seam bijection maintains scope impedance naturally as an invocation flows through the system. In Spring, we must inject a proxy of the Seam component, and resolve the reference when the proxy is invoked.

The **`<seam:instance/>`** tag lets us automatically proxy the Seam component.

```
<seam:instance id="seamManagedEM"
               name="someManagedEMComponent"
               proxy="true"/>

<bean id="someSpringBean" class="SomeSpringBeanClass">
  <property name="entityManager" ref="seamManagedEM">
</bean>
```

Here, we see one example of using a Seam-managed persistence context from a Spring bean. See the section on Section 26.6, “Using a Seam-Managed Persistence Context in Spring” for a more robust way to use Seam-managed persistence contexts as a replacement for the Spring **`OpenEntityManagerInView`** filter.

26.2. INJECTING SPRING BEANS INTO SEAM COMPONENTS

You can inject a Spring bean into a Seam component instance either by using an EL expression, or by making the Spring bean a Seam component.

The simplest approach is to access the Spring beans with EL.

The Spring **DelegatingVariableResolver** assists Spring integration with JavaServer Faces (JSF). This **VariableResolver** uses EL with bean IDs to make Spring beans available to JSF. You will need to add the **DelegatingVariableResolver** to **faces-config.xml**:

```
<application>
  <variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
  </variable-resolver>
</application>
```

You can then inject Spring beans using **@In**:

```
@In("#{bookingService}")
private BookingService bookingService;
```

Spring beans are not limited to injection. They can be used wherever EL expressions are used in Seam: process and pageflow definitions, working memory assertions, etc.

26.3. MAKING A SPRING BEAN INTO A SEAM COMPONENT

The **<seam:component/>** namespace handler can be used to transform any Spring bean into a Seam component. Just add the **<seam:component/>** tag to the declaration of the bean that you want to make into a Seam component:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <seam:component/>
</bean>
```

By default, **<seam:component/>** creates a stateless Seam component with the class and name provided in the bean definition. Occasionally — when a **FactoryBean** is used, for example — the Spring bean class may differ from the class listed in the bean definition. In this case, specify the **class** explicitly. You should also explicitly specify a Seam component name where there is a potential naming conflict.

If you want the Spring bean to be managed in a particular Seam scope, use the **scope** attribute of **<seam:component/>**. If the Seam scope specified is anything other than **STATELESS**, you must scope your Spring bean to **prototype**. Pre-existing Spring beans usually have a fundamentally stateless character, so this attribute is not usually necessary.

26.4. SEAM-SCOPED SPRING BEANS

With the Seam integration package, you can also use Seam's contexts as Spring 2.0-style *custom scopes*, which lets you declare any Spring bean in any Seam context. However, because Spring's component model was not built to support statefulness, this feature should be used with care. In particular, there are problems with clustering session- or conversation-scoped Spring beans, and care must be taken when injecting a bean or component from a wider scope into a bean of narrower scope.

Specify **<seam:configure-scopes/>** in a Spring bean factory configuration to make all Seam scopes available to Spring beans as custom scopes. To associate a Spring bean with a particular Seam scope, specify the desired scope in the **scope** attribute of the bean definition.

```
<!-- Only needs to be specified once per bean factory-->
```

```
<seam:configure-scopes/>

...

<bean id="someSpringBean" class="SomeSpringBeanClass"
      scope="seam.CONVERSATION"/>
```

You can change the scope name's prefix by specifying the **`prefix`** attribute in the **`configure-scopes`** definition. (The default prefix is **`seam.`**.)

By default, a Spring component instance that is registered this way is not created automatically when referenced with **`@In`**. To automatically create an instance, you must either specify **`@In(create=true)`** at the injection point (to auto-create a specific bean), or use the **`default-auto-create`** attribute of **`configure-scopes`** to auto-create all Seam-scoped Spring beans.

The latter approach lets you inject Seam-scoped Spring beans into other Spring beans without using **`<seam:instance/>`**. However, you must be careful to maintain scope impedance. Normally, you would specify **`<aop:scoped-proxy/>`** in the bean definition, but Seam-scoped Spring beans are not compatible with **`<aop:scoped-proxy/>`**. Therefore, to inject a Seam-scoped Spring bean into a singleton, use **`<seam:instance/>`**:

```
<bean id="someSpringBean" class="SomeSpringBeanClass"
scope="seam.CONVERSATION"/>

...

<bean id="someSingleton">
  <property name="someSeamScopedSpringBean">
    <seam:instance name="someSpringBean" proxy="true"/>
  </property>
</bean>
```

26.5. USING SPRING PLATFORMTRANSACTIONMANAGEMENT

Spring's extensible transaction management provides support for many transaction APIs, including the Java Persistence API (JPA), Hibernate, Java Data Objects (JDO), and Java Transaction API (JTA). It also exposes support for many advanced features such as nested transactions. Spring also provides tight integration with many application server TransactionManagers, such as Websphere and Weblogic, and supports full Java EE transaction propagation rules, such as **`REQUIRES_NEW`** and **`NOT_SUPPORTED`**. See the Spring Documentation for further information.

To configure Seam to use Spring transactions, enable the **`SpringTransaction`** component, like so:

```
<spring:spring-transaction
        platform-transaction-manager="#{transactionManager}"/>
```

The **`spring:spring-transaction`** component will utilizes Spring's transaction synchronization capabilities for synchronization callbacks.

26.6. USING A SEAM-MANAGED PERSISTENCE CONTEXT IN SPRING

Some of Seam's most powerful features are its conversation scope, and the ability to keep an **`EntityManager`** open for the life of a conversation. These eliminate many problems associated with

detaching and reattaching entities, and mitigate the occurrence of **`LazyInitializationException`**. Spring does not provide a way to manage persistence contexts beyond the scope of a single web request (**`OpenEntityManagerInViewFilter`**).

Seam brings conversation-scoped persistence context capabilities to Spring applications by allowing Spring developers to access a Seam-managed persistence context with the JPA tools provided with Spring (**`PersistenceAnnotationBeanPostProcessor`**, **`JpaTemplate`**, etc.)

This integration work provides:

- transparent access to a Seam-managed persistence context using Spring-provided tools
- access to Seam conversation-scoped persistence contexts in a non-web request — for example, an asynchronous Quartz job
- the ability to use Seam-managed persistence contexts with Spring-managed transactions. This requires manual flushing of the persistent context.

Spring's persistence context propagation model allows only one open **`EntityManager`** per **`EntityManagerFactory`**, so the Seam integration works by wrapping an **`EntityManagerFactory`** around a Seam-managed persistence context, like so:

```
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
</bean>
```

Here, **`persistenceContextName`** is the name of the Seam-managed persistence context component. By default, this **`EntityManagerFactory`** has a **`unitName`** equal to the Seam component name — in this case, **`entityManager`**. If you wish to provide a different **`unitName`**, you can provide a **`persistenceUnitName`** like so:

```
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
  <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

This **`EntityManagerFactory`** can now be used in any Spring-provided tools; in this case, you can use Spring's **`PersistenceAnnotationBeanPostProcessor`** just as you would in Spring.

```
<bean class="org.springframework.orm.jpa.support
             .PersistenceAnnotationBeanPostProcessor"/>
```

If you define your real **`EntityManagerFactory`** in Spring, but wish to use a Seam-managed persistence context, you can tell the **`PersistenceAnnotationBeanPostProcessor`** your desired default **`persistenctUnitName`** by specifying the **`defaultPersistenceUnitName`** property.

The **`applicationContext.xml`** might look like:

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="bookingDatabase"/>
</bean>
```

```
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
  <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
<bean class="org.springframework.orm.jpa
             .support.PersistenceAnnotationBeanPostProcessor">
  <property name="defaultPersistenceUnitName"
            value="bookingDatabase:extended"/>
</bean>
```

The **component.xml** might look like:

```
<persistence:managed-persistence-context name="entityManager"
    auto-create="true" entity-manager-factory="#{entityManagerFactory}"/>
```

JpaTemplate and **JpaDaoSupport** have an identical configuration in a Spring-based persistence context and in a normal Seam-managed persistence context.

```
<bean id="bookingService"
      class="org.jboss.seam.example.spring.BookingService">
  <property name="entityManagerFactory" ref="seamEntityManagerFactory"/>
</bean>
```

26.7. USING A SEAM-MANAGED HIBERNATE SESSION IN SPRING

Spring integration into Seam also provides support for complete Spring tool access to a Seam-managed Hibernate session. This integration is very similar to the JPA integration — see Section 26.6, “Using a Seam-Managed Persistence Context in Spring” for details.

Spring's propagation model allows only one open **EntityManager** per **EntityManagerFactory** to be available to Spring tools, so Seam integrates by wrapping a proxy **SessionFactory** around a Seam-managed Hibernate session context.

```
<bean id="seamSessionFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedSessionFactoryBean">
  <property name="sessionName" value="hibernateSession"/>
</bean>
```

Here, **sessionName** is the name of the **persistence:managed-hibernate-session** component. This **SessionFactory** can then be used with any Spring-provided tool. The integration also provides support for calls to **SessionFactory.getCurrentInstance()**, provided that **getCurrentInstance()** is called on the **SeamManagedSessionFactory**.

26.8. SPRING APPLICATION CONTEXT AS A SEAM COMPONENT

Although it is possible to use the Spring **ContextLoaderListener** to start your application's Spring **ApplicationContext**, there are some limitations: the Spring **ApplicationContext** must be started after the **SeamListener**, and starting a Spring **ApplicationContext** for use in Seam unit and integration tests can be complicated.

To overcome these limitations, the Spring integration includes a Seam component that can start a Spring **ApplicationContext**. To use this component, place the **<spring:context-loader/>** definition in

the **components.xml** file. Specify your Spring context file location in the **config-locations** attribute. If more than one configuration file is required, you can place them in the nested **<spring:config-locations/>** element, as per standard **components.xml** multi-value practices.

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:spring="http://jboss.com/products/seam/spring"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/spring
               http://jboss.com/products/seam/spring-2.2.xsd">

  <spring:context-loader config-locations=
                           "/WEB-INF/applicationContext.xml"/>

</components>
```

26.9. USING A SPRING TASKEXECUTOR FOR @ASYNCHRONOUS

Spring provides an abstraction for executing code asynchronously, called a **TaskExecutor**. The Spring-Seam integration lets you use a Spring **TaskExecutor** to execute immediate **@Asynchronous** method calls. To enable this functionality, install the **SpringTaskExecutorDispatchor** and provide a Spring -bean defined **taskExecutor** like so:

```
<spring:task-executor-dispatcher
        task-executor="#{springThreadPoolTaskExecutor}"/>
```

Because a Spring **TaskExecutor** does not support scheduling asynchronous events, you can provide handling with a fallback Seam **Dispatcher**, like so:

```
<!--
  Install a ThreadPoolDispatcher to handle scheduled asynchronous event
-->
<core:thread-pool-dispatcher name="threadPoolDispatcher"/>

<!-- Install the SpringDispatcher as default -->
<spring:task-executor-dispatcher
        task-executor="#{springThreadPoolTaskExecutor}"
        schedule-dispatcher="#{threadPoolDispatcher}"/>
```

CHAPTER 27. HIBERNATE SEARCH

27.1. INTRODUCTION

Full text search engines like Apache™ Lucene™ bring full text and efficient queries to applications. Hibernate Search, which makes use of Apache Lucene, can index your domain model with a few added annotations, handle database or index synchronization, and return regular managed objects that are matched by full text queries. There are some limitations to dealing with an object domain model over a text index — such as maintaining index accuracy, consistency between index structure and the domain model, and avoiding query mismatches — but these limitations are far outweighed by the advantages of speed and efficiency.

Hibernate Search has been designed to integrate as naturally as possible with the Java Persistence API (JPA) and Hibernate. As a natural extension, JBoss Seam provides Hibernate Search integration.

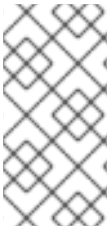
Refer to the Hibernate Search documentation for information specific to the Hibernate Search project.

27.2. CONFIGURATION

Hibernate Search is configured either in the **`META-INF/persistence.xml`** or **`hibernate.cfg.xml`** file.

Hibernate Search configuration has sensible defaults for most configuration parameters. The following is an example of a minimal persistence unit configuration:

```
<persistence-unit name="sample">
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
              value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
              value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit>
```

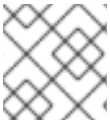


NOTE

When using Hibernate Search 3.1.x, more event listeners are required, but these are registered automatically by Hibernate Annotations. Refer to the *Hibernate Search Reference Guide* to learn to configure event listeners without using Hibernate EntityManager and Hibernate Annotations.

The following **JAR**s must be deployed alongside the configuration file:

- **`hibernate-search.jar`**
- **`hibernate-commons-annotations.jar`**
- **`lucene-core.jar`**



NOTE

If you deploy these in an **EAR**, remember to update **application.xml**.

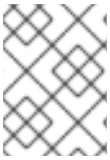
27.3. USAGE

Hibernate Search uses annotations to map entities to a Lucene index. Check the reference documentation for more information.

Hibernate Search is completely integrated with the API, and semantic of JPA and Hibernate. Switching from a HQL- or Criteria-based query requires little code. The application interacts primarily with the **FullTextSession** API, which is a subclass of Hibernate's **Session**.

When Hibernate Search is present, JBoss Seam injects a **FullTextSession**:

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{

  @In FullTextSession session;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    org.hibernate.Query query session.createFullTextQuery(luceneQuery,

Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
                    .setFirstResult(pageSize * currentPage)
                    .list();
  }
  [...]
}
```



NOTE

Here, **FullTextSession** extends **org.hibernate.Session** so that it can be used as a regular Hibernate Session.

A smoother integration is proposed if the JPA is used:

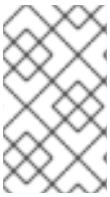
```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{
  @In FullTextEntityManager em;
  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    javax.persistence.Query query = em.createFullTextQuery(luceneQuery,

Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
```

```
                    .setFirstResult(pageSize * currentPage)
                    .getResultList();
  }
  [...]
}
```

Here, a **FulltextEntityManager** is injected where Hibernate Search is present. **FullTextEntityManager** extends **EntityManager** with search specific methods, the same way **FullTextSession** extends **Session**.

When an EJB 3.0 Session or Message Driven Bean injection is used (that is, where injection uses the **@PersistenceContext** annotation), the **EntityManager** interface cannot be replaced by using the **FullTextEntityManager** interface in the declaration statement. However, the implementation injected will be a **FullTextEntityManager** implementation, which allows downcasting.

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{

  @PersistenceContext EntityManager em;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    FullTextEntityManager ftEm = (FullTextEntityManager) em;
    javax.persistence.Query query =
      ftEm.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
                    .setFirstResult(pageSize * currentPage)
                    .getResultList();
  }
  [...]
}
```



NOTE

If you are accustomed to using Hibernate Search outside Seam, remember that you do not need to use **Search.createFullTextSession** when Hibernate Search is integrated with Seam.

For a working example of Hibernate Search, check the DVDStore or Blog examples in the JBoss Seam distribution.

CHAPTER 28. CONFIGURING SEAM AND PACKAGING SEAM APPLICATIONS

Configuration can be complex and tedious, but for the most part you will not need to write configuration data from scratch. Only a few lines of XML are required to integrate Seam with your JavaServer Faces (JSF) implementation and Servlet container, and for the most part you can either use seam-gen to start your application, or simply copy and paste from the example applications provided with Seam.

28.1. BASIC SEAM CONFIGURATION

First, the basic configuration required whenever Seam is used with JSF.

28.1.1. Integrating Seam with JSF and your servlet container

First, define a Faces Servlet.

```
<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>*.seam</url-pattern>
</servlet-mapping>
```

(You can adjust the URL pattern as you like.)

Seam also requires the following entry in your **web.xml** file:

```
<listener>
  <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
</listener>
```

This listener is responsible for bootstrapping Seam, and for destroying session and application contexts.

Some JSF implementations do not implement server-side state saving correctly, which interferes with Seam's conversation propagation. If you have problems with conversation propagation during form submissions, try switching to client-side state saving. To do so, add the following to **web.xml**:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

The JSF specification is unclear about the mutability of view state values. Since Seam uses the JSF view state to back its **PAGE** scope, this can be problematic. If you use server-side state saving with the JSF-RI (JSF Reference Implementation), and you want a page-scoped bean to retain its exact value for a given page view, you must specify the context parameter as follows:

```
<context-param>
  <param-name>com.sun.faces.serializeServerState</param-name>
```

```
  <param-value>true</param-value>
</context-param>
```

If this is not specified, a page-scoped component will contain the latest value of the page, and not the value of the "back" page, when the "back" button is used. (See this specification issue for details.) This setting is not enabled by default because serializing the JSF view with every request lowers overall performance.

28.1.2. Using Facelets

To use the recommended Facelets over JavaServer Pages (JSP), add the following lines to **`faces-config.xml`**:

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

Then, add the following lines to **`web.xml`**:

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

28.1.3. Seam Resource Servlet

The Seam Resource Servlet provides resources used by Seam Remoting, CAPTCHAs (see the Security chapter) and some JSF UI controls. Configuring the Seam Resource Servlet requires the following entry in **`web.xml`**:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.servlet.SeamResourceServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

28.1.4. Seam Servlet filters

Seam does not require Servlet filters for basic operation, but there are several features that depend upon filter use. Seam lets you add and configure Servlet filters as you would configure other built-in Seam components. To use this configuration method, you must first install a master filter in **`web.xml`**:

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>
<filter-mapping>
```

```
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

To ensure that it is run first, the Seam master filter *must* be the first filter specified in **web.xml**.

The Seam filters share a number of common attributes, which can be set in **components.xml**, along with any parameters discussed below:

- **url-pattern** — Specifies which requests are filtered. The default is all requests. **url-pattern** is a pattern which allows a wildcard suffix.
- **regex-url-pattern** — Specifies which requests are filtered. The default is all requests. **regex-url-pattern** is a true regular expression match for request path.
- **disabled** — Disables a built in filter.

These patterns are matched against the URI path of the request (see **HttpServletRequest.getURIPath()**), and the name of the Servlet context is removed before matching occurs.

Adding the master filter enables the following built-in filters:

28.1.4.1. Exception handling

This filter is required by most applications, and provides the exception mapping functionality in **pages.xml**. It also rolls back uncommitted transactions when uncaught exceptions occur. (The web container should do this automatically, but this does not occur reliably in some application servers.)

By default, the exception handling filter will process all requests, but you can adjust this behavior by adding a **<web:exception-filter>** entry to **components.xml**, like so:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:web="http://jboss.com/products/seam/web">
  <web:exception-filter url-pattern="*.seam"/>
</components>
```

28.1.4.2. Conversation propagation with redirects

This filter allows Seam to propagate the conversation context across browser redirects. It intercepts any browser redirects and adds a request parameter that specifies the Seam conversation identifier.

The redirect filter processes all requests by default, but this behavior can also be adjusted in **components.xml**:

```
<web:redirect-filter url-pattern="*.seam"/>
```

28.1.4.3. URL rewriting

This filter lets Seam apply URL rewriting for views, depending on its configuration in **pages.xml**. This filter is not active by default, but can be activated by adding the following configuration to **components.xml**:

```
<web:rewrite-filter view-mapping="*.seam"/>
```

The **view-mapping** parameter must match the Servlet mapping defined for the Faces Servlet in the **web.xml** file. If omitted, the rewrite filter assumes the pattern ***.seam**.

28.1.4.4. Multipart form submissions

This feature is required when you use the Seam *file upload* JSF control. It detects multipart form requests and processes them according the multipart/form-data specification (RFC-2388). Add the following to **components.xml** to override settings:

```
<web:multipart-filter create-temp-files="true"
     max-request-size="1000000" url-pattern="*.seam"/>
```

- **create-temp-files** — If **true**, uploaded files are written to a temporary file, rather than being held in memory. This can be important if you expect large file uploads. By default, this is set to **false**.
- **max-request-size** — If the size of a file upload request exceeds this value, the request will be aborted. The default setting is **0** (no size limit). (The size of a file upload is determined by reading the **Content-Length** header in the request.)

28.1.4.5. Character encoding

This filter sets the character encoding of submitted form data. It is not installed by default, and requires an entry in **components.xml** to enable it:

```
<web:character-encoding-filter encoding="UTF-16"
     override-client="true" url-pattern="*.seam"/>
```

- **encoding** — The type of encoding to use.
- **override-client** — If set to **true**, the request encoding will be set to that specified by **encoding**, regardless of whether the request specifies a particular encoding. If set to **false**, the request encoding will only be set if the client has not already specified the request encoding. By default, this is set to **false**.

28.1.4.6. RichFaces

If RichFaces is used in your project, Seam automatically installs the RichFaces AJAX filter before all other built-in filters, so there is no need to add it to **web.xml** manually.

The RichFaces Ajax filter is installed only if the RichFaces **JAR**s are present in your project.

To override the default settings, add the following entry to **components.xml**. The options are the same as those specified in the RichFaces Developer Guide:

```
<web:ajax4jsf-filter force-parser="true" enable-cache="true"
     log4j-init-file="custom-log4j.xml" url-pattern="*.seam"/>
```

- **force-parser** — forces all JSF pages to be validated by RichFaces's XML syntax checker. If **false**, only AJAX responses are validated and converted to well-formed XML. Setting **force-parser** to **false** improves performance, but can provide visual artifacts on AJAX updates.

- **`enable-cache`** — enables caching of framework-generated resources, such as JavaScript, CSS, images, etc. When developing custom JavaScript or CSS, setting this to **`true`** prevents the browser from caching the resource.
- **`log4j-init-file`** — is used to set up per-application logging. A path, relative to web application context, to the **`log4j.xml`** configuration file should be provided.

28.1.4.7. Identity Logging

This filter adds the authenticated username to the **`log4j`** mapped diagnostic context, so that it can be included in formatted log output by adding **`%X{username}`** to the pattern.

By default, the logging filter processes all requests. You can adjust this behavior by adding a **`<web:logging-filter>`** entry to **`components.xml`**, like so:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:web="http://jboss.com/products/seam/web">
  <web:logging-filter url-pattern="*.seam"/>
</components>
```

28.1.4.8. Context management for custom servlets

Requests that are sent directly to Servlets other than the JSF Servlet are not processed in the JSF life cycle, so Seam provides a Servlet filter that can be applied to any other Servlet requiring access to Seam components.

This filter lets custom Servlets interact with Seam contexts. It sets up Seam contexts at the beginning of each request, and removes them at the end of the request. This filter should *never* be applied to the JSF **`FacesServlet`** — Seam uses the phase listener to manage context in a JSF request.

This filter is not installed by default, and must be enabled in **`components.xml`**:

```
<web:context-filter url-pattern="/media/*"/>
```

The context filter expects the conversation ID of any conversation context to be defined in the **`conversationId`** request parameter. You are responsible for ensuring that this is included in the request.

You are also responsible for ensuring that any new conversation ID propagates back to the client. Seam exposes the conversation ID as a property of the built in component **`conversation`**.

28.1.4.9. Adding custom filters

Seam can install your filters for you. This allows you to specify your filter's placement in the chain — the Servlet specification does not provide a well-defined order when you specify your filters in **`web.xml`**. Add a **`@Filter`** annotation to your Seam component. (Your Seam component must implement **`javax.servlet.Filter`**.)

```
@Startup
@Scope(APPLICATION)
@Name("org.jboss.seam.web.multipartFilter")
```

```
@BypassInterceptors
@Filter(within="org.jboss.seam.web.ajax4jsfFilter")
public class MultipartFilter extends AbstractFilter {...}
```

Adding the **@Startup** annotation makes the component available during Seam start up. Bijection is not available here (**@BypassInterceptors**), and the filter should be further down the chain than the RichFaces filter (**@Filter(within="org.jboss.seam.web.ajax4jsfFilter")**).

28.1.5. Integrating Seam with your EJB container

EJB components in a Seam application are managed by both Seam and the EJB container. Seam resolves EJB component references, manages the lifetime of stateful session bean components, and participates in each method call via interceptors. To integrate Seam with your EJB container, you must first configure the interceptor chain.

Apply the **SeamInterceptor** to your Seam EJB components. This interceptor delegates to a set of built-in server-side interceptors that handle operations like bijection, conversation demarcation, and business process signals. The simplest way to do this across an entire application is to add the following interceptor configuration in **ejb-jar.xml**:

```
<interceptors>
  <interceptor>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor>
</interceptors>
<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

You must tell seam where to find session beans in JNDI. You could do this by specifying a **@JndiName** annotation on every session bean Seam component. A better approach is to specify a pattern with which Seam can calculate the JNDI name from the EJB name. However, the EJB3 specification does not define a standard method of mapping to global JNDI — this mapping is vendor-specific, and can also depend upon your naming conventions. We specify this option in **components.xml**:

For JBoss Enterprise Application Platform, the following pattern is correct:

```
<core:init jndi-name="earName/#{ejbName}/local" />
```

Here, **earName** is the name of the EAR in which the bean is deployed. Seam replaces **#{ejbName}** with the name of the EJB, and the final segment represents the type of interface (local or remote).

When outside the **EAR** context (for example, when using the JBoss Embeddable EJB3 container), the first segment is dropped, since there is no **EAR**, which leaves us with the following pattern:

```
<core:init jndi-name="#{ejbName}/local" />
```

This process looks complicated, but in reality comprises few steps.

First, we will talk about how the EJB component is transferred to JNDI. To avoid using XML, JBoss Enterprise Application Platform uses the aforementioned pattern (that is, **EAR** name/EJB name/interface type) to automatically assign an EJB component a global JNDI name. The EJB name will be the first non-empty value out of the following:

- the **<ejb-name>** element in **ejb-jar.xml**,
- the **name** attribute in the **@Stateless** or **@Stateful** annotation, or
- the simple name of the bean class.

For example, assume that you have the following EJB bean and interface defined:

```
package com.example.myapp;
import javax.ejb.Local;

@Local
public class Authenticator {
  boolean authenticate();
}


package com.example.myapp;
import javax.ejb.Stateless;

@Stateless
@Name("authenticator")
public class AuthenticatorBean implements Authenticator {
  public boolean authenticate() { ... }
}
```

Assuming that your EJB bean class is deployed in an **EAR** named **myapp**, the global JNDI name assigned on the JBoss Enterprise Application Platform will be **myapp/AuthenticatorBean/local**. You can refer to this EJB component as a Seam component with the name **authenticator**, and Seam will use the JNDI pattern (or the **@JndiName** annotation) to locate it in JNDI.

For other application servers, you must declare an EJB reference for your EJB so that it is assigned a JNDI name. This does require some XML, and means that you must establish your own JNDI naming convention so that you can use the Seam JNDI pattern. It may be useful to follow the JBoss convention.

You must define the EJB references in two locations when using Seam with a non-JBoss application server. If you look up the Seam EJB component with JSF (in a JSF view, or as a JSF action listener) or a Seam JavaBean component, then you must declare the EJB reference in **web.xml**. The EJB reference that would be required for our example is:\

```
<ejb-local-ref>
  <ejb-ref-name>myapp/AuthenticatorBean/local</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>org.example.vehicles.action.Authenticator</local>
</ejb-local-ref>
```

This reference covers most uses of the component in a Seam application. If you want to be able to inject a Seam EJB component into another Seam EJB component with the **@In** annotation, you must define this EJB reference in a second location: **ejb-jar.xml**. This is slightly more complicated.

When Seam looks for a Seam EJB component to satisfy an injection point defined with **@In**, the component will only be found if it is referenced in JNDI. JBoss automatically registers EJBs to the JNDI so that they are always available to the web and EJB containers. Other containers require you to define your EJBs explicitly.

Application servers that adhere to the EJB specification require that EJB references are always explicitly defined. These cannot be declared globally — you must specify each JNDI resource for an EJB component individually.

Assuming that you have an EJB with a resolved name of **RegisterAction**, the following Seam injection applies:

```
@In(create = true) Authenticator authenticator;
```

For this injection to work, you must also establish the link in **ejb-jar.xml**, like so:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>RegisterAction</ejb-name>
      <ejb-local-ref>
      <ejb-ref-name>myapp/AuthenticatorAction/local</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local>com.example.myapp.Authenticator</local>
      </ejb-local-ref>
    </session>
  </enterprise-beans>

  ...

</ejb-jar>
```

The component is referenced here just as it was in **web.xml**. Identifying it here brings the reference into the EJB context, where it can be used by the **RegisterAction** bean. You must add one reference for each injection (via **@In**) of one Seam EJB component into another Seam EJB component. You can see an example of this setup in the **jee5/booking** example.

It is possible to inject one EJB into another with the **@EJB** annotation, but this injects the EJB reference rather than the Seam EJB component instance. Because Seam's interceptor is invoked on any *method call* to an EJB component, and using **@EJB** only invokes Seam's server-side interceptor chain, some Seam features will not work with **@EJB** injection. (Seam's state management and Seam's client-side interceptor chain, which handles security and concurrency, are two affected features.) When a stateful session bean is injected using the **@EJB** annotation, it will not necessarily bind to the active session or conversation, either, so we recommend injecting with **@In**.

Some application servers (such as Glassfish) require you to specify JNDI names for all EJB components explicitly, sometimes more than once. You may also need to alter the JNDI pattern used in Seam, even if you follow the JBoss Enterprise Application Platform naming convention. For example, in Glassfish the global JNDI names are automatically prefixed with **java:comp/env**, so you must define the JNDI pattern as follows:

-

```
<core:init jndi-name="java:comp/env/earName/#{ejbName}/local" />
```

For transaction management, we recommend using a special built-in component that is fully aware of container transactions, and can correctly process transaction success events registered with the **Events** component. To tell Seam when container-managed transactions end, add the following line to your **components.xml** file:

```
<transaction:ejb-transaction/>
```

28.1.6. Remember

The final requirement for integration is that a **seam.properties**, **META-INF/seam.properties** or **META-INF/components.xml** file be placed in any archive in which your Seam components are deployed. For web archive (WAR) files, place a **seam.properties** file inside the **WEB-INF/classes** directory in which your components are deployed.

Seam scans any archive with **seam.properties** files for Seam components at start up. The **seam.properties** file can be empty, but it must be included so that the component is recognized by Seam. This is a workaround for Java Virtual Machine (JVM) limitations — without the **seam.properties** file, you would need to list every component explicitly in **components.xml**.

28.2. USING ALTERNATE JPA PROVIDERS

Seam comes packaged and configured with Hibernate as the default JPA provider. To use a different JPA provider, you must configure it with Seam.

NOTE

This is a workaround — future versions of Seam will not require configuration changes to use alternative JPA providers, unless you add a custom persistence provider implementation.

There are two ways to tell Seam about your JPA provider. The first is to update your application's **components.xml** so that the generic **PersistenceProvider** takes precedence over the Hibernate version. Simply add the following to the file:

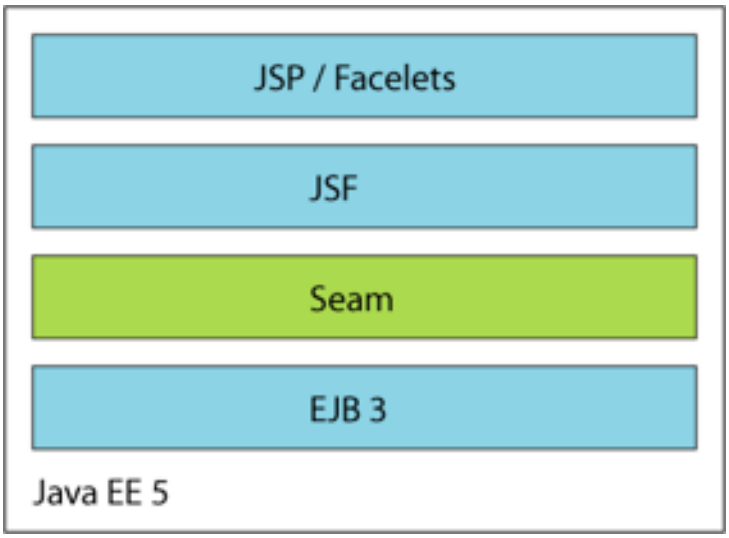
```
<component name="org.jboss.seam.persistence.persistenceProvider"
           class="org.jboss.seam.persistence.PersistenceProvider"
           scope="stateless">
</component>
```

To take advantage of any of your JPA provider's non-standard features, you must write your own implementation of the **PersistenceProvider**. (You can use **HibernatePersistenceProvider** as a starting point.) Tell Seam to use this **PersistenceProvider** like so:

```
<component name="org.jboss.seam.persistence.persistenceProvider"
           class="org.your.package.YourPersistenceProvider">
</component>
```

Now, update **persistence.xml** with the correct provider class, and any properties required by your provider. Remember to package any required **JAR** files with your application.

28.3. CONFIGURING SEAM IN JAVA EE 5



If you're running in a Java EE 5 environment, this is all the configuration required to start using Seam!

28.3.1. Packaging

Once packaged into an **EAR**, your archive will be structured similarly to the following:

```
my-application.ear/
    jboss-seam.jar
    lib/
        jboss-el.jar
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
            faces-config.xml
            lib/
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
```

```
                    Register.class
                    RegisterBean.class
          ...
```

Declare **jboss-seam.jar** as an EJB module in **META-INF/application.xml**. Add **jboss-el.jar** to the **EAR** classpath by placing it in the **EAR**'s **lib** directory.

To use jBPM or Drools, include the required **JAR**s in the **EAR**'s **lib** directory.

Tp use Facelets, as recommended, include **jsf-facelets.jar** in the **WEB-INF/lib** directory of the **WAR**.

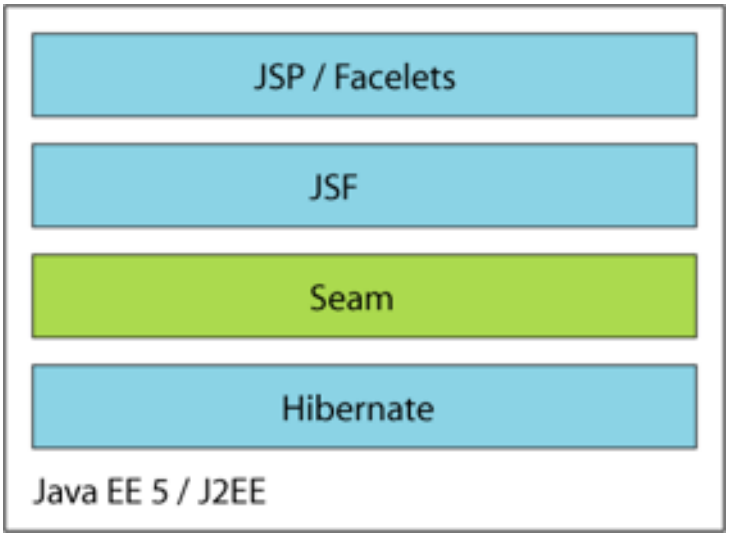
Most applications use the Seam tag library — to do so, include **jboss-seam-ui.jar** in the **WEB-INF/lib** directory of the WAR. To use the PDF or email tag libraries, you must also place **jboss-seam-pdf.jar** or **jboss-seam-mail.jar** in **WEB-INF/lib**.

To use the Seam debug page, include **jboss-seam-debug.jar** in the **WEB-INF/lib** directory of the **WAR**. Seam's debug page only works for applications using Facelets.)

Seam also ships with several example applications — these are deployable in any Java EE container with EJB3 support.

28.4. CONFIGURING SEAM IN J2EE

You can use Hibernate 3 or JPA instead of EJB3 persistence, and plain JavaBeans instead of session beans. You can still take advantage of Seam's declarative state management architecture, and it is easy to migrate to EJB3.



Unlike session beans, Seam JavaBean components do not provide declarative transaction demarcation. Most applications use Seam-managed transactions when using Hibernate with JavaBeans, but you can also manage your transactions manually with the JTA **UserTransaction**, or declaratively with Seam's **@Transactional** annotation.

The Seam distribution includes extra versions of the booking example application — one uses Hibernate3 and JavaBeans instead of EJB3, and the other uses JPA and JavaBeans. These example applications are ready to deploy into any J2EE application server.

28.4.1. Boostrapping Hibernate in Seam

Install the following built-in component to have Seam bootstrap a Hibernate **SessionFactory** from your **hibernate.cfg.xml** file:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

To make a Seam-managed Hibernate **Session** available via injection, configure a **managed session** as follows:

```
<persistence:managed-hibernate-session name="hibernateSession"
            session-factory="#{hibernateSessionFactory}"/>
```

28.4.2. Boostrapping JPA in Seam

Install the following built-in component to have Seam bootstrap a JPA **EntityManagerFactory** from your **persistence.xml** file:

```
<persistence:entity-manager-factory name="entityManagerFactory"/>
```

To make a Seam-managed JPA **EntityManager** available via injection, configure a managed persistence context as follows:

```
<persistence:managed-persistence-context name="entityManager"
            entity-manager-factory="#{entityManagerFactory}"/>
```

28.4.3. Packaging

Your application will have the following structure when packaged as a **WAR**:

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            jboss-el.jar
            jsf-facelets.jar
            hibernate3.jar
            hibernate-annotations.jar
            hibernate-validator.jar
            ...
            my-application.jar/
                META-INF/
                   MANIFEST.MF
                seam.properties
                hibernate.cfg.xml
                org/
                    jboss/
                        myapplication/
                            User.class
```

```
                            Login.class
                            Register.class
                            ...
    login.jsp
    register.jsp
    ...
```

Some additional configuration is required in order to deploy Hibernate in a non-EE environment, such as TestNG.

28.5. CONFIGURING SEAM IN JAVA SE, WITHOUT JBOSS EMBEDDED

To use Seam outside an EE environment, you must tell Seam how to manage transactions, since JTA will not be available. If you use JPA, you can tell Seam to use JPA resource-local transactions — that is, **`EntityTransaction`** — like so:

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

If you use Hibernate, you can tell Seam to use the Hibernate transaction API with the following:

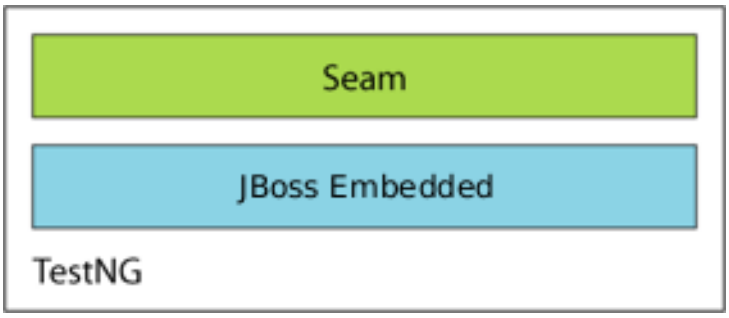
```
<transaction:hibernate-transaction session="#{session}"/>
```

You must also define a datasource.

28.6. CONFIGURING SEAM IN JAVA SE, WITH JBOSS EMBEDDED

JBoss Embedded lets you run EJB3 components outside the context of the Java EE 5 application server. This is particularly useful in testing.

The Seam booking example application includes a TestNG integration test suite that runs on Embedded JBoss via **`SeamTest`**.



28.6.1. Packaging

A **`WAR`**-based deployment on a Servlet engine will be structured as follows:

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
```

```
            jboss-seam.jar
            jboss-seam-ui.jar
            jboss-el.jar
            jsf-facelets.jar
            jsf-api.jar
            jsf-impl.jar
            ...
            my-application.jar/
                META-INF/
                    MANIFEST.MF
                    persistence.xml
                seam.properties
                org/
                    jboss/
                        myapplication/
                            User.class
                            Login.class
                            LoginBean.class
                            Register.class
                            RegisterBean.class
                            ...
    login.jsp
    register.jsp
    ...
```

28.7. CONFIGURING JBPM IN SEAM

Seam's jBPM integration is not installed by default. To enable jBPM, you must install a built-in component. You must also explicitly list your process and pageflow definitions. In **`components.xml`**:

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>createDocument.jpdl.xml</value>
    <value>editDocument.jpdl.xml</value>
    <value>approveDocument.jpdl.xml</value>
  </bpm:pageflow-definitions>
  <bpm:process-definitions>
    <value>documentLifecycle.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

If you only have pageflows, no further configuration is required. If you have business process definitions, you must provide a jBPM configuration, and a Hibernate configuration for jBPM. The Seam DVD Store demo includes example **`jbpm.cfg.xml`** and **`hibernate.cfg.xml`** files that will work with Seam:

```
<jbpm-configuration>
  <jbpm-context>
    <service name="persistence">
      <factory>
        <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
          <field name="isTransactionEnabled"><false/></field>
        </bean>
      </factory>
    </service>
    <service name="tx" factory="org.jbpm.tx.TxServiceFactory" />
```

```
    <service name="message"
     factory="org.jbpm.msg.db.DbMessageServiceFactory" />
    <service name="scheduler"
             factory="org.jbpm.scheduler.db.DbSchedulerServiceFactory" />
    <service name="logging"
             factory="org.jbpm.logging.db.DbLoggingServiceFactory" />
    <service name="authentication"
             factory="org.jbpm.security.authentication
                          .DefaultAuthenticationServiceFactory"/>
  </jbpm-context>
</jbpm-configuration>
```

Note that jBPM transaction control is disabled — JTA transactions should be controlled by either Seam or EJB3.

28.7.1. Packaging

There is no well-defined packaging format for jBPM configuration and process or pageflow definition files. While other standard packaging formats may be developed, the Seam examples are packaged into the root of the **EAR**, and follow this structure:

```
my-application.ear/
    jboss-seam.jar
    lib/
       jboss-el.jar
       jbpm-jpdl.jar
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
            faces-config.xml
            lib/
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
                    Register.class
                    RegisterBean.class
```

```
                    ...
    jbpm.cfg.xml
    hibernate.cfg.xml
    createDocument.jpdl.xml
    editDocument.jpdl.xml
    approveDocument.jpdl.xml
    documentLifecycle.jpdl.xml
```

28.8. CONFIGURING SFSB AND SESSION TIMEOUTS IN EAP

The timeout for stateful session beans must be longer than the timeout for HTTP sessions, or the stateful session bean may timeout before the user's HTTP Session ends. The EAP has a default session bean timeout of 30 minutes, which is configured in **`server/default/conf/standardjboss.xml`** — to change this, replace **`default`** with your own preferred configuration.

In the **`LRUStatefulContextCachePolicy`** cache configuration, modify the value of **`max-bean-life`** to change the default stateful session bean timeout:

```
<container-cache-conf>
  <cache-policy>
    org.jboss.ejb.plugins.LRUStatefulContextCachePolicy
  </cache-policy>
  <cache-policy-conf>
    <min-capacity>50</min-capacity>
    <max-capacity>1000000</max-capacity>
    <remover-period>1800</remover-period>

    <!-- SFSB timeout in seconds; 1800 seconds == 30 minutes -->
    <max-bean-life>1800</max-bean-life>

    <overager-period>300</overager-period>
    <max-bean-age>600</max-bean-age>
    <resizer-period>400</resizer-period>
    <max-cache-miss-period>60</max-cache-miss-period>
    <min-cache-miss-period>1</min-cache-miss-period>
    <cache-load-factor>0.75</cache-load-factor>
  </cache-policy-conf>
</container-cache-conf>
```

You can modify the default HTTP Session timeout in **`server/default/deployer/jbossweb.deployer/web.xml`** for JBoss Enterprise Application Platform 5.1. The following entry in the **`web.xml`** file controls the default session timeout for all web applications:

```
<session-config>
  <!-- HTTP Session timeout, in minutes -->
  <session-timeout>30</session-timeout>
</session-config>
```

To override this value for your own application, simply include a modified version of this entry in your application's own **`web.xml`**.

28.9. RUNNING SEAM IN A PORTLET

WARNING

Technology Preview features are not fully supported under Red Hat subscription level agreements (SLAs), may not be functionally complete, and are not intended for production use. However, these features provide early access to upcoming product innovations, enabling customers to test functionality and provide feedback during the development process. As Red Hat considers making future iterations of Technology Preview features generally available, we will provide commercially reasonable efforts to resolve any reported issues that customers experience when using these features.

You can use the JBoss Portlet Bridge to run your Seam application in a portlet. The bridge supports JSF within a portlet, and includes extensions for Seam and RichFaces. See http://labs.jboss.com/portletbridge for more information.

28.10. DEPLOYING CUSTOM RESOURCES

On start up, Seam scans all **JAR**s containing **/seam.properties**, **/META-INF/components.xml** or **/META-INF/seam.properties** for resources. For example, all classes annotated with **@Name** are registered on start as Seam components.

You can also use Seam to handle custom resources — that is, Seam can handle specific annotations. First, provide a list of annotation types to handle in the **/META-INF/seam-deployment.properties** files, like so:

```
# A colon-separated list of annotation types to handle
org.jboss.seam.deployment.annotationTypes=com.acme.Foo:com.acme.Bar
```

Then, collect all classes annotated with **@Foo** on application start up:

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {

   @In("#{deploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> fooClasses;

   @In("#{hotDeploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> hotFooClasses;

   @Create
   public void create() {
      for (Class clazz: fooClasses) {
         handleClass(clazz);
      }
      for (Class clazz: hotFooClasses) {
         handleClass(clazz);
      }
   }
```

```
    public void handleClass(Class clazz) {
        // ...
    }

}
```

You can also set Seam to handle any resource. For example, if you want to process files with the **`.foo.xml`** extension, you can write a custom deployment handler:

```
public class FooDeploymentHandler implements DeploymentHandler {
  private static DeploymentMetadata FOO_METADATA = new
DeploymentMetadata() {

    public String getFileNameSuffix() {
      return ".foo.xml";
    }
  };

  public String getName() {
    return "fooDeploymentHandler";
  }

  public DeploymentMetadata getMetadata() {
    return FOO_METADATA;
  }
}
```

This provides us with a list of all files with the **`.foo.xml`** suffix.

Next, register the deployment handler with Seam in **`/META-INF/seam-deployment.properties`**:

```
# For standard deployment
# org.jboss.seam.deployment.deploymentHandlers=
#    com.acme.FooDeploymentHandler

# For hot deployment
# org.jboss.seam.deployment.hotDeploymentHandlers=
#    com.acme.FooDeploymentHandler
```

You can register multiple deployment handlers with a comma-separated list.

Seam uses deployment handlers internally to install components and namespaces, so the **`handle()`** is called too early in Seam bootstrap to be useful. You can access the deployment handler easily during the start up of an application-scoped component:

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {
  @In("#{deploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myDeploymentHandler;
  @In("#
{hotDeploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myHotDeploymentHandler;
```

```
  @Create public void create() {
    for (FileDescriptor fd: myDeploymentHandler.getResources()) {
      handleFooXml(fd);
    }
    for (FileDescriptor f: myHotDeploymentHandler.getResources()) {
      handleFooXml(fd);
    }
  }

  public void handleFooXml(FileDescriptor fd) {
      // ...
  }
}
```

CHAPTER 29. SEAM ANNOTATIONS

Seam uses annotations to achieve a declarative style of programming. Most annotations are defined by the Enterprise JavaBean 3.0 (EJB3) specification, and the annotations used in data validation are defined by the Hibernate Validator package. However, Seam also defines its own set of annotations, which are described in this chapter.

All of these annotations are defined in the **`org.jboss.seam.annotations`** package.

29.1. ANNOTATIONS FOR COMPONENT DEFINITION

This group of annotations is used to define a Seam component. These annotations appear on the component class.

`@Name`

```
@Name("componentName")
```

Defines the Seam component name for a class. This annotation is required for all Seam components.

`@Scope`

```
@Scope(ScopeType.CONVERSATION)
```

Defines the default context of the component. The possible values are defined by the **`ScopeType`** enumeration: **`EVENT`**, **`PAGE`**, **`CONVERSATION`**, **`SESSION`**, **`BUSINESS_PROCESS`**, **`APPLICATION`**, or **`STATELESS`**.

When no scope is explicitly specified, the default varies with the component type. For stateless session beans, the default is **`STATELESS`**. For entity beans and stateful session beans, the default is **`CONVERSATION`**. For JavaBeans, the default is **`EVENT`**.

`@Role`

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

Allows a Seam component to be bound to multiple context variables. The **`@Name`** and **`@Scope`** annotations define a *default role*. Each **`@Role`** annotation defines an additional role.

- **`name`** — the context variable name.
- **`scope`** — the context variable scope. When no scope is explicitly specified, the default depends upon the component type, as above.

`@Roles`

```
@Roles({ @Role(name="user", scope=ScopeType.CONVERSATION),
@Role(name="currentUser", scope=ScopeType.SESSION) })
```

Allows you to specify multiple additional roles.

`@BypassInterceptors`

```
@BypassInterceptors
```

Disables all Seam interceptors on a particular component or component method.

@JndiName

```
@JndiName("my/jndi/name")
```

Specifies the JNDI name that Seam will use to look up the EJB component. If no JNDI name is explicitly specified, Seam will use the JNDI pattern specified by **org.jboss.seam.core.init.jndiPattern**.

@Conversational

```
@Conversational
```

Specifies that a conversation scope component is conversational, meaning that no method of the component may be called unless a long-running conversation is active.

@PerNestedConversation

```
@PerNestedConversation
```

Limits the scope of a conversation-scoped component to the parent conversation in which it was instantiated. The component instance will not be visible to nested child conversations, which will operate within their own instances.

WARNING

This is not a recommended application feature. It implies that a component will be visible only for a specific part of a request cycle.

@Startup

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

Specifies that an application-scoped component will start immediately at initialization time. This is used for built-in components that bootstrap critical infrastructure, such as JNDI, datasources, etc.

```
@Scope(SESSION) @Startup
```

Specifies that a session-scoped component will start immediately at session creation time.

- **depends** — specifies that the named components must be started first, if they are installed.

@Install

```
@Install(false)
```

Specifies that a component should not be installed by default. (If you do not specify this annotation, the component will be installed.)

```
@Install(dependencies="org.jboss.seam.bpm.jbpm")
```

Specifies that a component should only be installed if the components listed as dependencies are also installed.

```
@Install(genericDependencies=ManagedQueueSender.class)
```

Specifies that a component should only be installed if a component that is implemented by a certain class is installed. This is useful when a required dependency does not have a single well-known name.

```
@Install(classDependencies="org.hibernate.Session")
```

Specifies that a component should only be installed if the named class is included on the classpath.

```
@Install(precedence=BUILT_IN)
```

Specifies the precedence of the component. If multiple components with the same name exist, the one with the higher precedence will be installed. The defined precedence values are (in ascending order):

- **`BUILT_IN`** — precedence of all built-in Seam components.
- **`FRAMEWORK`** — precedence to use for components of frameworks which extend Seam.
- **`APPLICATION`** — precedence of application components (the default precedence).
- **`DEPLOYMENT`** — precedence to use for components which override application components in a particular deployment.
- **`MOCK`** — precedence for mock objects used in testing.

`@Synchronized`

```
@Synchronized(timeout=1000)
```

Specifies that a component is accessed concurrently by multiple clients, and that Seam should serialize requests. If a request is not able to obtain its lock on the component in the given timeout period, an exception will be raised.

`@ReadOnly`

```
@ReadOnly
```

Specifies that a JavaBean component or component method does not require state replication at the end of the invocation.

`@AutoCreate`

```
@AutoCreate
```

Specifies that a component will be automatically created, even if the client does not specify **`create=true`**.

29.2. ANNOTATIONS FOR BIJECTION

The next two annotations control bijection. These attributes occur on component instance variables or property accessor methods.

`@In`

```
@In
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. If the context variable is null, an exception will be thrown.

```
@In(required=false)
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. The context variable may be null.

```
@In(create=true)
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. If the context variable is null, an instance of the component is instantiated by Seam.

```
@In(value="contextVariableName")
```

Specifies the name of the context variable explicitly, instead of using the annotated instance variable name.

```
@In(value="#{customer.addresses['shipping']}")
```

Specifies that a component attribute is to be injected by evaluating a JSF EL expression at the beginning of each component invocation.

- **`value`** — specifies the name of the context variable. Defaults to the name of the component attribute. Alternatively, specifies a JSF EL expression, surrounded by **`#{...}`**.
- **`create`** — specifies that Seam should instantiate the component with the same name as the context variable, if the context variable is undefined (null) in all contexts. Defaults to **`false`**.
- **`required`** — specifies that Seam should throw an exception if the context variable is undefined in all contexts.

`@Out`

```
@Out
```

Specifies that a component attribute that is a Seam component is to be outjected to its context variable at the end of the invocation. If the attribute is null, an exception is thrown.

```
@Out(required=false)
```

Specifies that a component attribute that is a Seam component is to be outjected to its context variable at the end of the invocation. The attribute can be null.

```
@Out(scope=ScopeType.SESSION)
```

Specifies that a component attribute that is *not* a Seam component type is to be outjected to a specific scope at the end of the invocation.

Alternatively, if no scope is explicitly specified, the scope of the component with the **`@Out`** attribute issued (or the **`EVENT`** scope if the component is stateless).

```
@Out(value="contextVariableName")
```

Specifies the name of the context variable explicitly, instead of using the annotated instance variable name.

- **`value`** — specifies the name of the context variable. Default to the name of the component attribute.
- **`required`** — specifies that Seam should throw an exception if the component attribute is null during outjection.

These annotations commonly occur together, as in the following example:

```
@In(create=true)
@Out private User currentUser;
```

The next annotation supports the *manager component* pattern, where a Seam component manages the life cycle of an instance of some other class that is to be injected. It appears on a component getter method.

`@Unwrap`

```
@Unwrap
```

Specifies that the object returned by the annotated getter method will be injected instead of the component.

The next annotation supports the *factory component* pattern, in which a Seam component is responsible for initializing the value of a context variable. This is especially useful for initializing any state required to render a response to a non-Faces request. It appears on a component method.

`@Factory`

```
@Factory("processInstance")
public void createProcessInstance() { ... }
```

Specifies that the component method be used to initialize the value of the named context variable, when the context variable has no value. This style is used with methods that return **`void`**.

```
@Factory("processInstance", scope=CONVERSATION)
public ProcessInstance createProcessInstance() { ... }
```

Specifies that the value returned by the method should be used to initialize the value of the named context variable, if the context variable has no value. This style is used with methods that return a value. If no scope is explicitly specified, the scope of the component with the **`@Factory`** method is used (unless the component is stateless, in which case the **`EVENT`** context is used).

- **`value`** — specifies the name of the context variable. If the method is a getter method, this defaults to the JavaBeans property name.
- **`scope`** — specifies the scope to which Seam should bind the returned value. Only meaningful for factory methods that return a value.
- **`autoCreate`** — specifies that this factory method should be automatically called whenever the variable is asked for, even if **`@In`** does not specify **`create=true`**.

The following annotation lets you inject a **`Log`**:

`@Logger`

```
@Logger("categoryName")
```

Specifies that a component field is to be injected with an instance of **`org.jboss.seam.log.Log`**. For entity beans, the field must be declared as **`static`**.

- **`value`** — specifies the name of the log category. Defaults to the name of the component class.

The final annotation lets you inject a request parameter value:

`@RequestParameter`

```
@RequestParameter("parameterName")
```

Specifies that a component attribute is to be injected with the value of a request parameter. Basic type conversions are performed automatically.

- **`value`** — specifies the name of the request parameter. Defaults to the name of the component attribute.

29.3. ANNOTATIONS FOR COMPONENT LIFE CYCLE METHODS

These annotations allow a component to react to its own life cycle events. They occur on methods of the component. Only one of these annotations may be used in any one component class.

`@Create`

```
@Create
```

Specifies that the method should be called when an instance of the component is instantiated by Seam. Create methods are only supported for JavaBeans and stateful session beans.

@Destroy

```
@Destroy
```

Specifies that the method should be called when the context ends and its context variables are destroyed. Destroy methods are only supported for JavaBeans and stateful session beans.

Destroy methods should be used only for cleanup. Seam catches, logs and swallows any exception that propagates out of a destroy method.

@Observer

```
@Observer("somethingChanged")
```

Specifies that the method should be called when a component-driven event of the specified type occurs.

```
@Observer(value="somethingChanged",create=false)
```

Specifies that the method should be called when an event of the specified type occurs, but that an instance should not be created if it does not already exist. If an instance does not exist and create is set to **false**, the event will not be observed. The default value is **true**.

29.4. ANNOTATIONS FOR CONTEXT DEMARCATION

These annotations provide declarative conversation demarcation. They appear on Seam component methods, usually action listener methods.

Every web request is associated with a conversation context. Most of these conversations end when the request is complete. To span a conversation across multiple requests, you must "promote" the conversation to a *long-running conversation* by calling a method marked with **@Begin**.

@Begin

```
@Begin
```

Specifies that a long-running conversation begins when this method returns a non-null outcome without exception.

```
@Begin(join=true)
```

Specifies that, if a long-running conversation is already in progress, the conversation context is propagated.

```
@Begin(nested=true)
```

Specifies that, if a long-running conversation is already in progress, a new *nested* conversation context should begin. The nested conversation will end when the next **@End** is encountered, and the outer conversation will resume. Multiple nested conversations can exist concurrently in the same outer conversation.

```
@Begin(pageflow="process definition name")
```

Specifies a jBPM process definition name that defines the pageflow for this conversation.

```
@Begin(flushMode=FlushModeType.MANUAL)
```

Specifies the flush mode of any Seam-managed persistence contexts. **`flushMode=FlushModeType.MANUAL`** supports the use of *atomic conversations*, where all write operations are queued in the conversation context until an explicit call to **`flush()`** (which usually occurs at the end of the conversation) is made.

- **`join`** — determines the behavior when a long-running conversation is already in progress. If **`true`**, the context is propagated. If **`false`**, an exception is thrown. Defaults to **`false`**. This setting is ignored when **`nested=true`** is specified.
- **`nested`** — specifies that a nested conversation should be started if a long-running conversation is already in progress.
- **`flushMode`** — sets the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.
- **`pageflow`** — the name of a jBPM process definition deployed via **`org.jboss.seam.bpm.jbpm.pageflowDefinitions`**.

`@End`

```
@End
```

Specifies that a long-running conversation ends when this method returns a non-null outcome without exception.

- **`beforeRedirect`** — by default, the conversation will not actually be destroyed until after any redirect has occurred. Setting **`beforeRedirect=true`** specifies that the conversation should be destroyed at the end of the current request, and that the redirect will be processed in a new temporary conversation context.
- **`root`** — by default, ending a nested conversation simply pops the conversation stack and resumes the outer conversation. Setting **`root=true`** specifies that the root conversation should be destroyed, which destroys the entire conversation stack. If the conversation is not nested, the current conversation is destroyed.

`@StartTask`

```
@StartTask
```

Starts a jBPM task. Specifies that a long-running conversation begins when this method returns a non-null outcome without exception. This conversation is associated with the jBPM task specified in the named request parameter. Within the context of this conversation, a business process context is

also defined, for the business process instance of the task instance.

- The jBPM **`TaskInstance`** is available in the **`taskInstance`** request context variable. The jBPM **`ProcessInstance`** is available in the **`processInstance`** request context variable. These objects can be injected with **`@In`**.
- **`taskIdParameter`** — the name of a request parameter which holds the task ID. Default to **`"taskId"`**, which is also the default used by the Seam **`taskList`** JSF component.
- **`flushMode`** — sets the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.

`@BeginTask`

```
@BeginTask
```

Resumes work on an incomplete jBPM task. Specifies that a long-running conversation begins when this method returns a non-null outcome without exception. This conversation is associated with the jBPM task specified in the named request parameter. Within the context of this conversation, a business process context is also defined, for the business process instance of the task instance.

- The jBPM **`org.jbpm.taskmgmt.exe.TaskInstance`** is available in the **`taskInstance`** request context variable. The jBPM **`org.jbpm.graph.exe.ProcessInstance`** is available in the **`processInstance`** request context variable.
- **`taskIdParameter`** — the name of a request parameter which holds the ID of the task. Defaults to **`"taskId"`**, which is also the default used by the Seam **`taskList`** JSF component.
- **`flushMode`** — sets the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.

`@EndTask`

```
@EndTask
```

Ends a jBPM task. Specifies that a long-running conversation ends when this method returns a non-null outcome, and that the current task is complete. Triggers a jBPM transition. The actual transition triggered will be the default transition unless the application has called **`Transition.setName()`** on the built-in component named **`transition`**.

```
@EndTask(transition="transitionName")
```

Triggers the specified jBPM transition.

- **`transition`** — the name of the jBPM transition to be triggered when ending the task. Defaults to the default transition.
- **`beforeRedirect`** — by default, the conversation will not actually be destroyed until after any redirect has occurred. Setting **`beforeRedirect=true`** specifies that the conversation should be destroyed at the end of the current request, and that the redirect will be processed in a new temporary conversation context.

`@CreateProcess`

```
@CreateProcess(definition="process definition name")
```

Creates a new jBPM process instance when the method returns a non-null outcome without exception. The **`ProcessInstance`** object will be available in a context variable named **`processInstance`**.

- **`definition`** — the name of the jBPM process definition deployed via **`org.jboss.seam.bpm.jbpm.processDefinitions`**.

`@ResumeProcess`

```
@ResumeProcess(processIdParameter="processId")
```

Re-enters the scope of an existing jBPM process instance when the method returns a non-null outcome without exception. The **`ProcessInstance`** object will be available in a context variable named **`processInstance`**.

- **`processIdParameter`** — the name of the request parameter that holds the process ID. Defaults to **`"processId"`**.

`@Transition`

```
@Transition("cancel")
```

Marks a method as signalling a transition in the current jBPM process instance whenever the method returns a non-null result.

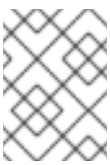
29.5. ANNOTATIONS FOR USE WITH SEAM JAVABEAN COMPONENTS IN A J2EE ENVIRONMENT

Seam provides an annotation that lets you force a rollback of the JTA transaction for certain action listener outcomes.

`@Transactional`

```
@Transactional
```

Specifies that a JavaBean component should have similar transactional behavior to the default behavior of a session bean component. That is, method invocations should take place in a transaction, and if no transaction exists when the method is called, a transaction will be started just for that method. This annotation can be applied at either class or method level.



NOTE

This annotation should not be used on EJB3 components — use **`@TransactionAttribute`** instead.

`@ApplicationException`

```
@ApplicationException
```

Applied to an exception to denote that it is an application exception and should be reported to the client directly — that is, unwrapped. Operates identically to **`javax.ejb.ApplicationException`** when used in a pre-Java EE 5 environment.

NOTE

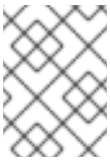
This annotation should not be used on EJB3 components — use **`@javax.ejb.ApplicationException`** instead.

- **`rollback`** — by default **`false`**, if **`true`** this exception sets the transaction to rollback only.
- **`end`** — by default **`false`**, if **`true`**, this exception ends the current long-running conversation.

`@Interceptors`

```
@Interceptors({DVDInterceptor, CDInterceptor})
```

Declares an ordered list of interceptors for a class or method. Operates identically to **`javax.interceptors.Interceptors`** when used in a pre-Java EE 5 environment. Note that this may only be used as a meta-annotation.



NOTE

This annotation should not be used on EJB3 components — use **`@javax.interceptor.Interceptors`** instead.

These annotations are used primarily for JavaBean Seam components. If you use EJB3 components, you should use the standard Java EE 5 annotations.

29.6. ANNOTATIONS FOR EXCEPTIONS

These annotations let you specify how Seam handles any exceptions propagating from a Seam component.

`@Redirect`

```
@Redirect(viewId="error.jsp")
```

Specifies that the annotated exception causes a browser redirect to a specified view ID.

- **`viewId`** — specifies the JSF view ID to redirect to. You can use EL here.
- **`message`** — a message to be displayed. Defaults to the exception message.
- **`end`** — specifies that the long-running conversation should end. Defaults to **`false`**.

`@HttpError`

```
@HttpError(errorCode=404)
```

Specifies that the annotated exception causes a HTTP error to be sent.

- **`errorCode`** — the HTTP error code. Defaults to **`500`**.
- **`message`** — a message to be sent with the HTTP error. Defaults to the exception message.
- **`end`** — specifies that the long-running conversation should end. Defaults to **`false`**.

29.7. ANNOTATIONS FOR SEAM REMOTING

Seam Remoting requires that the local interface of a session bean be annotated with the following annotation:

`@WebRemote`

```
@WebRemote(exclude="path.to.exclude")
```

Indicates that the annotated method may be called from client-side JavaScript. The **`exclude`** property is optional, and allows objects to be excluded from the result's object graph. (See the Chapter 24, *Remoting* chapter for more details.)

29.8. ANNOTATIONS FOR SEAM INTERCEPTORS

The following annotations appear on Seam interceptor classes.

Please refer to the documentation for the EJB3 specification for information about the annotations required to define EJB interceptors.

`@Interceptor`

```
@Interceptor(stateless=true)
```

Specifies that this interceptor is stateless and Seam may optimize replication.

```
@Interceptor(type=CLIENT)
```

Specifies that this interceptor is a "client-side" interceptor, called prior to the EJB container.

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

Specifies that this interceptor is positioned higher in the stack than the given interceptors.

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

Specifies that this interceptor is positioned deeper in the stack than the given interceptors.

29.9. ANNOTATIONS FOR ASYNCHRONICITY

The following annotations are used to declare an asynchronous method, as in the following example:

```
@Asynchronous public void scheduleAlert(Alert alert,
                                        @Expiration Date date) {
```

```
  ...
}
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert,
                                          @Expiration Date date,
                                          @IntervalDuration long
interval) {
  ...
}
```

@Asynchronous

```
@Asynchronous
```

Specifies that the method call is processed asynchronously.

@Duration

```
@Duration
```

Specifies the parameter of the asynchronous call that relates to the duration before the call is processed (or first processed, for recurring calls).

@Expiration

```
@Expiration
```

Specifies the parameter of the asynchronous call that relates to the date and time at which the call is processed (or first processed, for recurring calls).

@IntervalDuration

```
@IntervalDuration
```

Specifies that an asynchronous method call recurs. The associated parameter defines the duration of the interval between recurrences.

29.10. ANNOTATIONS FOR USE WITH JSF

The following annotations make it easier to work with JSF.

@Converter

Allows a Seam component to act as a JSF converter. The annotated class must be a Seam component, and must implement **`javax.faces.convert.Converter`**.

- **`id`** — the JSF converter ID. Defaults to the component name.
- **`forClass`** — if specified, registers this component as the default converter for a type.

@Validator

Allows a Seam component to act as a JSF validator. The annotated class must be a Seam component, and must implement **`javax.faces.validator.Validator`**.

- **`id`** — the JSF validator ID. Defaults to the component name.

29.10.1. Annotations for use with `dataTable`

The following annotations make it easy to implement clickable lists backed by a stateful session bean. They appear on attributes.

`@DataModel`

```
@DataModel("variableName")
```

Outjects a property of type **`List`**, **`Map`**, **`Set`** or **`Object[]`** as a JSF **`DataModel`** into the scope of the owning component (or the **`EVENT`** scope, if the owning component is **`STATELESS`**). In the case of **`Map`**, each row of the **`DataModel`** is a **`Map.Entry`**.

- **`value`** — name of the conversation context variable. Default to the attribute name.
- **`scope`** — if **`scope=ScopeType.PAGE`** is explicitly specified, the **`DataModel`** will be kept in the **`PAGE`** context.

`@DataModelSelection`

```
@DataModelSelection
```

Injects the selected value from the JSF **`DataModel`**. (This is the element of the underlying collection, or the map value.) If only one **`@DataModel`** attribute is defined for a component, the selected value from that **`DataModel`** will be injected. Otherwise, the component name of each **`@DataModel`** must be specified in the value attribute for each **`@DataModelSelection`**.

If **`PAGE`** scope is specified on the associated **`@DataModel`**, then the associated DataModel will be injected in addition to the DataModel Selection. In this case, if the property annotated with **`@DataModel`** is a getter method, then a setter method for the property must also be part of the Business API of the containing Seam Component.

- **`value`** — name of the conversation context variable. Not needed if there is exactly one **`@DataModel`** in the component.

`@DataModelSelectionIndex`

```
@DataModelSelectionIndex
```

Exposes the selection index of the JSF **`DataModel`** as an attribute of the component. (This is the row number of the underlying collection, or the map key.) If only one **`@DataModel`** attribute is defined for a component, the selected value from that **`DataModel`** will be injected. Otherwise, the component name of each **`@DataModel`** must be specified in the value attribute for each **`@DataModelSelectionIndex`**.

- **`value`** — name of the conversation context variable. This is not required if there is exactly one **`@DataModel`** in the component.

29.11. META-ANNOTATIONS FOR DATABINDING

These meta-annotations make it possible to implement similar functionality to **@DataModel** and **@DataModelSelection** for other datastructures apart from lists.

@DataBinderClass

```
@DataBinderClass(DataModelBinder.class)
```

Specifies that an annotation is a databinding annotation.

@DataSelectorClass

```
@DataSelectorClass(DataModelSelector.class)
```

Specifies that an annotation is a dataselection annotation.

29.12. ANNOTATIONS FOR PACKAGING

This annotation provides a mechanism for declaring information about a set of components that are packaged together. It can be applied to any Java package.

@Namespace

```
@Namespace(value="http://jboss.com/products/seam/example/seampay")
```

Specifies that components in the current package are associated with the given namespace. The declared namespace can be used as an XML namespace in a **components.xml** file to simplify application configuration.

```
@Namespace(value="http://jboss.com/products/seam/core",
                   prefix="org.jboss.seam.core")
```

Specifies a namespace to associate with a given package. Additionally, it specifies a component name prefix to be applied to component names specified in the XML file. For example, an XML element named **init** that is associated with this namespace would be understood to actually refer to a component named **org.jboss.seam.core.init**.

29.13. ANNOTATIONS FOR INTEGRATING WITH THE SERVLET CONTAINER

These annotations allow you to integrate your Seam components with the Servlet container.

@Filter

When used to annotate a Seam component implementing **javax.servlet.Filter**, designates that component as a servlet filter to be executed by Seam's master filter.

- ```
  @Filter(around={"seamComponent", "otherSeamComponent"})
  ```

  Specifies that this filter is positioned higher in the stack than the given filters.
  ```

- ```
  @Filter(within={"seamComponent", "otherSeamComponent"})
  ```

  Specifies that this filter is positioned deeper in the stack than the given filters.
  ```

CHAPTER 30. BUILT-IN SEAM COMPONENTS

This chapter describes Seam's built-in components, and their configuration properties. The built-in components are created automatically, even if they are not listed in your **components.xml** file. However, if you need to override default properties or specify more than one component of a certain type, you can do so in **components.xml**.

You can replace any of the built-in components with your own implementation by using **@Name** to name your own class after the appropriate built-in component.

30.1. CONTEXT INJECTION COMPONENTS

The first set of built-in components support the injection of various contextual objects. For example, the following component instance variable would have the Seam session context object injected:

```
@In private Context sessionContext;
```

org.jboss.seam.core.contexts

Component that provides access to Seam Context objects such as **org.jboss.seam.core.contexts.sessionContext['user']**.

org.jboss.seam.faces.facesContext

Manager component for the **FacesContext** context object. (This is not a true Seam context.)

All of these components are always installed.

30.2. JSF-RELATED COMPONENTS

The following set of components are provided to supplement JSF.

org.jboss.seam.faces.dateConverter

Provides a default JSF converter for properties of type **java.util.Date**.

This converter is automatically registered with JSF, so developers need not specify a DateTimeConverter on an input field or page parameter. By default, it assumes the type to be a date (as opposed to a time or date plus time), and uses the short input style adjusted to the user's **Locale**. For **Locale.US**, the input pattern is **mm/dd/yy**. However, to comply with Y2K, the year is changed from two digits to four — **mm/dd/yyyy**.

You can override the input pattern globally by reconfiguring your component. Consult the JavaServer Faces documentation for this class to see examples.

org.jboss.seam.faces.facesMessages

Allows Faces success messages to propagate across a browser redirect.

- **add(FacesMessage facesMessage)** — adds a Faces message, which will be displayed during the next render response phase that occurs in the current conversation.
- **add(String messageTemplate)** — adds a Faces message, rendered from the given message template, which may contain EL expressions.

- **`add(Severity severity, String messageTemplate)`** — adds a Faces message, rendered from the given message template, which may contain EL expressions.
- **`addFromResourceBundle(String key)`** — adds a Faces message, rendered from a message template defined in the Seam resource bundle which may contain EL expressions.
- **`addFromResourceBundle(Severity severity, String key)`** — adds a Faces message, rendered from a message template defined in the Seam resource bundle, which may contain EL expressions.
- **`clear()`** — clears all messages.

`org.jboss.seam.faces.redirect`

A convenient API for performing redirects with parameters. This is particularly useful for bookmarkable search results screens.

- **`redirect.viewId`** — the JSF view ID to redirect to.
- **`redirect.conversationPropagationEnabled`** — determines whether the conversation will propagate across the redirect.
- **`redirect.parameters`** — a map of request parameter name to value, to be passed in the redirect request.
- **`execute()`** — performs the redirect immediately.
- **`captureCurrentRequest()`** — stores the view ID and request parameters of the current GET request (in the conversation context) for later use by calling **`execute()`**.

`org.jboss.seam.faces.httpError`

A convenient API for sending HTTP errors.

`org.jboss.seam.ui.renderStampStore`

A component which maintains a collection of render stamps. A render stamp indicates whether a rendered form has been submitted. This is particularly useful in conjunction with JSF's client-side state saving method, because the form's status (posted or unposted) is controlled by the server rather than the client.

Client-side state saving is often used to unbind this check from the session. To do so, you will need an implementation that can store render stamps within the application (valid while the application runs), or the database (valid across server restarts).

- **`maxSize`** — The maximum number of stamps to keep in the store. The default is **`100`**.

The JSF components are installed when the class **`javax.faces.context.FacesContext`** is available on the classpath.

30.3. UTILITY COMPONENTS

The following components provide various functions that are useful across a broad range of applications.

`org.jboss.seam.core.events`

An API for raising events that can be observed via **`@Observer`** methods, or method bindings in **`components.xml`**.

- **`raiseEvent(String type)`** — raises an event of a particular type and distributes it to all observers.
- **`raiseAsynchronousEvent(String type)`** — raises an event to be processed asynchronously by the EJB3 timer service.
- **`raiseTimedEvent(String type, ....)`** — schedules an event to be processed asynchronously by the EJB3 timer service.
- **`addListener(String type, String methodBinding)`** — adds an observer for a particular event type.

`org.jboss.seam.core.interpolator`

An API for interpolating the values of JSF EL expressions in Strings.

- **`interpolate(String template)`** — scans the template for JSF EL expressions of the form **`#{...}`** and replaces them with their evaluated values.

`org.jboss.seam.core.expressions`

An API for creating value and method bindings.

- **`createValueBinding(String expression)`** — creates a value binding object.
- **`createMethodBinding(String expression)`** — creates a method binding object.

`org.jboss.seam.core.pojoCache`

Manager component for a JBoss Cache **`PojoCache`** instance.

- **`pojoCache.cfgResourceName`** — the name of the configuration file. Defaults to **`treecache.xml`**.

All of these components are always installed.

30.4. COMPONENTS FOR INTERNATIONALIZATION AND THEMES

These components make it easy to build internationalized user interfaces using Seam.

`org.jboss.seam.core.locale`

The Seam locale.

`org.jboss.seam.international.timezone`

The Seam timezone. The timezone is session-scoped.

`org.jboss.seam.core.resourceBundle`

The Seam resource bundle. The resource bundle is stateless. The Seam resource bundle performs a depth-first search for keys in a list of Java resource bundles.

`org.jboss.seam.core.resourceLoader`

The resource loader provides access to application resources and resource bundles.

- **`resourceLoader.bundleNames`** — the names of the Java resource bundles to search when the Seam resource bundle is used. Default to **`messages`**.

`org.jboss.seam.international.localeSelector`

Supports selection of the locale either at configuration time, or by the user at runtime.

- **`select()`** — selects the specified locale.
- **`localeSelector.locale`** — the actual **`java.util.Locale`**.
- **`localeSelector.localeString`** — the string representation of the locale.
- **`localeSelector.language`** — the language for the specified locale.
- **`localeSelector.country`** — the country for the specified locale.
- **`localeSelector.variant`** — the variant for the specified locale.
- **`localeSelector.supportedLocales`** — a list of **`SelectItem`**s representing the supported locales listed in **`jsf-config.xml`**.
- **`localeSelector.cookieEnabled`** — specifies that the locale selection should be persisted via a cookie.

`org.jboss.seam.international.timezoneSelector`

Supports selection of the timezone either at configuration time, or by the user at runtime.

- **`select()`** — selects the specified locale.
- **`timezoneSelector.timezone`** — the actual **`java.util.TimeZone`**.
- **`timezoneSelector.timeZoneId`** — the string representation of the timezone.
- **`timezoneSelector.cookieEnabled`** — specifies that the timezone selection should be persisted via a cookie.

`org.jboss.seam.international.messages`

A map containing internationalized messages rendered from message templates defined in the Seam resource bundle.

`org.jboss.seam.theme.themeSelector`

Supports selection of the theme either at configuration time, or by the user at runtime.

- **`select()`** — select the specified theme.
- **`theme.availableThemes`** — the list of defined themes.
- **`themeSelector.theme`** — the selected theme.
- **`themeSelector.themes`** — a list of **`SelectItem`**s representing the defined themes.

- **`themeSelector.cookieEnabled`** — specifies that the theme selection should be persisted via a cookie.

`org.jboss.seam.theme.theme`

A map containing theme entries.

All of these components are always installed.

30.5. COMPONENTS FOR CONTROLLING CONVERSATIONS

The following components allow you to control conversations through either the application or the user interface.

`org.jboss.seam.core.conversation`

An API for controlling the current Seam conversation's attributes from within the application.

- **`getId()`** — returns the current conversation ID.
- **`isNested()`** — specifies whether the current conversation is a nested conversation.
- **`isLongRunning()`** — specifies whether the current conversation is a long-running conversation.
- **`getId()`** — returns the current conversation ID.
- **`getParentId()`** — returns the conversation ID of the parent conversation.
- **`getRootId()`** — returns the conversation ID of the root conversation.
- **`setTimeout(int timeout)`** — sets the timeout for the current conversation.
- **`setViewId(String outcome)`** — sets the view ID to use when switching back to the current conversation from the conversation switcher, conversation list, or breadcrumbs.
- **`setDescription(String description)`** — sets the description of the current conversation to be displayed in the conversation switcher, conversation list, or breadcrumbs.
- **`redirect()`** — redirects to the last well-defined view ID for this conversation. This is useful after log in challenges.
- **`leave()`** — exits the scope of this conversation, without actually ending the conversation.
- **`begin()`** — begins a long-running conversation (equivalent to **`@Begin`**).
- **`beginPageflow(String pageflowName)`** — begin a long-running conversation with a pageflow (equivalent to **`@Begin(pageflow="...")`**).
- **`end()`** — ends a long-running conversation (equivalent to **`@End`**).
- **`pop()`** — pops the conversation stack, and returns to the parent conversation.
- **`root()`** — returns to the root conversation of the conversation stack.

- **`changeFlushMode(FlushModeType flushMode)`** — changes the flush mode of the conversation.

`org.jboss.seam.core.conversationList`

A manager component for the conversation list.

`org.jboss.seam.core.conversationStack`

A manager component for the conversation stack (breadcrumbs).

`org.jboss.seam.faces.switcher`

The conversation switcher.

All of these components are always installed.

30.6. JBPM-RELATED COMPONENTS

The following components are used with jBPM.

`org.jboss.seam.pageflow.pageflow`

An API for controlling Seam pageflows.

- **`isInProcess()`** — returns **`true`** if there is currently a pageflow in process.
- **`getProcessInstance()`** — returns jBPM **`ProcessInstance`** for the current pageflow.
- **`begin(String pageflowName)`** — begins a pageflow in the context of the current conversation.
- **`reposition(String nodeName)`** — repositions the current pageflow to a particular node.

`org.jboss.seam.bpm.actor`

An API that controls the attributes of the jBPM actor associated with the current session, from within the application.

- **`setId(String actorId)`** — sets the jBPM actor ID of the current user.
- **`getGroupActorIds()`** — returns a **`Set`** to which jBPM actor IDs for the current users groups may be added.

`org.jboss.seam.bpm.transition`

An API that controls the current task's jBPM transition from within the application.

- **`setName(String transitionName)`** — sets the jBPM transition name to be used when the current task is ended via **`@EndTask`**.

`org.jboss.seam.bpm.businessProcess`

An API for programmatic control of the association between the conversation and business process.

- **`businessProcess.taskId`** — the ID of the task associated with the current conversation.

- **`businessProcess.processId`** — the ID of the process associated with the current conversation.
- **`businessProcess.hasCurrentTask()`** — specifies whether a task instance is associated with the current conversation.
- **`businessProcess.hasCurrentProcess()`** — specifies whether a process instance is associated with the current conversation.
- **`createProcess(String name)`** — creates an instance of the named process definition and associates it with the current conversation.
- **`startTask()`** — starts the task associated with the current conversation.
- **`endTask(String transitionName)`** — ends the task associated with the current conversation.
- **`resumeTask(Long id)`** — associates the task with the specified ID with the current conversation.
- **`resumeProcess(Long id)`** — associates the process with the specified ID with the current conversation.
- **`transition(String transitionName)`** — triggers the transition.

`org.jboss.seam.bpm.taskInstance`

A manager component for the jBPM **`TaskInstance`**.

`org.jboss.seam.bpm.processInstance`

A manager component for the jBPM **`ProcessInstance`**.

`org.jboss.seam.bpm.jbpmContext`

A manager component for an event-scoped **`JbpmContext`**.

`org.jboss.seam.bpm.taskInstanceList`

A manager component for the jBPM task list.

`org.jboss.seam.bpm.pooledTaskInstanceList`

A manager component for the jBPM pooled task list.

`org.jboss.seam.bpm.taskInstanceListForType`

A manager component for the jBPM task lists.

`org.jboss.seam.bpm.pooledTask`

An action handler for pooled task assignment.

`org.jboss.seam.bpm.processInstanceFinder`

A manager component for the process instance task list.

`org.jboss.seam.bpm.processInstanceList`

The process instance task list.

All of these components are installed whenever the component **org.jboss.seam.bpm.jbpm** is installed.

30.7. SECURITY-RELATED COMPONENTS

These components relate to web-tier security.

org.jboss.seam.web.userPrincipal

A manager component for the current user **Principal**.

org.jboss.seam.web.isUserInRole

Allows JSF pages to choose to render a control, depending upon the roles available to the current principal, for example: **<h:commandButton value="edit" rendered="#{isUserInRole['admin']}"/>**.

30.8. JMS-RELATED COMPONENTS

These components are for use with managed **TopicPublisher**s and **QueueSender**s (see below).

org.jboss.seam.jms.queueSession

A manager component for a JMS **QueueSession**.

org.jboss.seam.jms.topicSession

A manager component for a JMS **TopicSession**.

30.9. MAIL-RELATED COMPONENTS

These components are for use with Seam's Email support.

org.jboss.seam.mail.mailSession

A manager component for a JavaMail **Session**. The session can be either looked up in the JNDI context (by setting the **sessionJndiName** property), or created from the configuration options. In this case, the **host** is mandatory.

- **org.jboss.seam.mail.mailSession.host** — the hostname of the SMTP server to use.
- **org.jboss.seam.mail.mailSession.port** — the port of the SMTP server to use.
- **org.jboss.seam.mail.mailSession.username** — the username to use to connect to the SMTP server.
- **org.jboss.seam.mail.mailSession.password** — the password to use to connect to the SMTP server.
- **org.jboss.seam.mail.mailSession.debug** — enables JavaMail debugging (very verbose).

- **`org.jboss.seam.mail.mailSession.ssl`** — enables SSL connection to SMTP (will default to port 465).

  `org.jboss.seam.mail.mailSession.tls` — enables TLS support in the mail session. Defaults to **`true`**.

- **`org.jboss.seam.mail.mailSession.sessionJndiName`** — name under which a javax.mail.Session is bound to JNDI. If this is supplied, all other properties will be ignored.

30.10. INFRASTRUCTURAL COMPONENTS

These components provide critical platform infrastructure. You can install a component that is not installed by default by setting **`install="true"`** on the component in **`components.xml`**.

`org.jboss.seam.core.init`

This component contains initialization settings for Seam. Always installed.

- **`org.jboss.seam.core.init.jndiPattern`** — the JNDI pattern used for looking up session beans.
- **`org.jboss.seam.core.init.debug`** — enables Seam debug mode. During production, this should be set to **`false`**; you may see errors if the system is placed under any load while debug is enabled.
- **`org.jboss.seam.core.init.clientSideConversations`** — when **`true`**, saves conversation context variables in the client rather than the **`HttpSession`**.

`org.jboss.seam.core.manager`

An internal component for Seam page and conversation context management. Always installed.

- **`org.jboss.seam.core.manager.conversationTimeout`** — the conversation context timeout in milliseconds.
- **`org.jboss.seam.core.manager.concurrentRequestTimeout`** — the maximum wait time for a thread attempting to gain a lock on the long-running conversation context.
- **`org.jboss.seam.core.manager.conversationIdParameter`** — the request parameter used to propagate the conversation ID. The default is **`conversationId`**.
- **`org.jboss.seam.core.manager.conversationIsLongRunningParameter`** — the request parameter used to propagate that the conversation is long-running. The default is **`conversationIsLongRunning`**.
- **`org.jboss.seam.core.manager.defaultFlushMode`** — sets the default flush mode on any Seam-managed Persistence Context. This defaults to **`AUTO`**.

`org.jboss.seam.navigation.pages`

An internal component for Seam workspace management. Always installed.

- **`org.jboss.seam.navigation.pages.noConversationViewId`** — specifies the view ID to redirect to, globally, when a conversation entry is not found on the server side.

- **`org.jboss.seam.navigation.pages.loginViewId`** — specifies the view ID to redirect to, globally, when an unauthenticated user attempts to access a protected view.
- **`org.jboss.seam.navigation.pages.httpPort`** — specifies the port to use, globally, when the HTTP scheme is requested.
- **`org.jboss.seam.navigation.pages.httpsPort`** — specifies the port to use, globally, when the HTTPS scheme is requested.
- **`org.jboss.seam.navigation.pages.resources`** — specifies a list of resources to search for **`pages.xml`** style resources. The default is **`WEB-INF/pages.xml`**.

`org.jboss.seam.bpm.jbpm`

This component bootstraps a **`JbpmConfiguration`**. Install it as the **`org.jboss.seam.bpm.Jbpm`** class.

- **`org.jboss.seam.bpm.jbpm.processDefinitions`** — specifies a list of jPDL file resource names to use for orchestrating business processes.
- **`org.jboss.seam.bpm.jbpm.pageflowDefinitions`** — specifies a list of jPDL file resource names to use for orchestrating conversation page flows.

`org.jboss.seam.core.conversationEntries`

An internal session-scoped component that records active long-running conversations between requests.

`org.jboss.seam.faces.facesPage`

An internal page-scoped component that records the conversation context associated with a page.

`org.jboss.seam.persistence.persistenceContexts`

An internal component that records the persistence contexts used in the current conversation.

`org.jboss.seam.jms.queueConnection`

Manages a JMS **`QueueConnection`**. This is installed whenever managed **`QueueSender`** is installed.

- **`org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName`** — specifies the JNDI name of a JMS **`QueueConnectionFactory`**. The default is **`UIL2ConnectionFactory`**.

`org.jboss.seam.jms.topicConnection`

Manages a JMS **`TopicConnection`**. This is installed whenever managed **`TopicPublisher`** is installed.

- **`org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName`** — specifies the JNDI name of a JMS **`TopicConnectionFactory`**. The default is **`UIL2ConnectionFactory`**.

`org.jboss.seam.persistence.persistenceProvider`

An abstraction layer for non-standardized features of the JPA provider.

`org.jboss.seam.core.validators`

Caches instances of Hibernate Validator **`ClassValidator`**.

`org.jboss.seam.faces.validation`

Lets the application determine whether validation succeeded.

`org.jboss.seam.debug.introspector`

Provides support for the Seam Debug Page.

`org.jboss.seam.debug.contexts`

Provides support for the Seam Debug Page.

`org.jboss.seam.exception.exceptions`

An internal component for exception handling.

`org.jboss.seam.transaction.transaction`

An API for controlling transactions and abstracting the underlying transaction management implementation behind a JTA-compatible interface.

`org.jboss.seam.faces.safeActions`

Determines that an action expression in an incoming URL is safe by checking that the action expression exists in the view.

30.11. MISCELLANEOUS COMPONENTS

Additional, uncategorized components.

`org.jboss.seam.async.dispatcher`

Dispatches stateless session beans for asynchronous methods.

`org.jboss.seam.core.image`

Used for image manipulation and interrogation.

`org.jboss.seam.core.pojoCache`

A manager component for a PojoCache instance.

`org.jboss.seam.core.uiComponent`

Manages a map of UIComponents keyed by component ID.

30.12. SPECIAL COMPONENTS

Certain Seam component classes can be installed multiple times under names specified in the Seam configuration. For example, the following lines in **`components.xml`** install and configure two Seam components:

```
<component name="bookingDatabase"
           class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">
    java:/comp/emf/bookingPersistence
  </property>
</component>

<component name="userDatabase"
           class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">
    java:/comp/emf/userPersistence
  </property>
</component>
```

The Seam component names are **bookingDatabase** and **userDatabase**.

***<entityManager>*, org.jboss.seam.persistence.ManagedPersistenceContext**

A manager component for a conversation-scoped, managed **EntityManager** with an extended persistence context.

- *<entityManager>*.entityManagerFactory — a value binding expression that evaluates to an instance of **EntityManagerFactory**.

  <entityManager>.persistenceUnitJndiName — the JNDI name of the entity manager factory. By default, this is **java:/<managedPersistenceContext>** .

***<entityManagerFactory>*, org.jboss.seam.persistence.EntityManagerFactory**

Manages a JPA **EntityManagerFactory**. This is most useful when using JPA outside of an environment with EJB3 support.

- **entityManagerFactory.persistenceUnitName** — the name of the persistence unit.

See the API JavaDoc for further configuration properties.

***<session>*, org.jboss.seam.persistence.ManagedSession**

A manager component for a conversation-scoped, managed Hibernate **Session**.

- *<session>*.sessionFactory — a value binding expression that evaluates to an instance of **SessionFactory**.

  <session>.sessionFactoryJndiName — the JNDI name of the session factory. By default, this is **java:/<managedSession>**.

***<sessionFactory>*, org.jboss.seam.persistence.HibernateSessionFactory**

Manages a Hibernate **SessionFactory**.

- **<sessionFactory>.cfgResourceName** — specifies the path to the configuration file. By default, this is **hibernate.cfg.xml**.

See the API JavaDoc for further configuration properties.

***<managedQueueSender>*, org.jboss.seam.jms.ManagedQueueSender**

A manager component for an event scoped managed JMS **`QueueSender`**.

- *<managedQueueSender>*.queueJndiName — the JNDI name of the JMS queue.

***<managedTopicPublisher>*, `org.jboss.seam.jms.ManagedTopicPublisher`**

A manager component for an event-scoped, managed JMS **`TopicPublisher`**.

- *<managedTopicPublisher>*.topicJndiName — the JNDI name of the JMS topic.

***<managedWorkingMemory>*, `org.jboss.seam.drools.ManagedWorkingMemory`**

A manager component for a conversation-scoped, managed Drools **`WorkingMemory`**.

- *<managedWorkingMemory>*.ruleBase — a value expression that evaluates to an instance of **`RuleBase`**.

***<ruleBase>*, `org.jboss.seam.drools.RuleBase`**

A manager component for an application-scoped Drools **`RuleBase`**. Note that this does not support dynamic installation of new rules, so it is not appropriate for use in production.

- *<ruleBase>*.ruleFiles — a list of files containing Drools rules.

  <ruleBase>.dslFile — a Drools DSL definition.

CHAPTER 31. SEAM JSF CONTROLS

Seam includes a number of JavaServer Faces (JSF) controls to complement built-in controls, and controls from other third-party libraries. We recommend JBoss RichFaces and Apache MyFaces Trinidad tag libraries for use with Seam. We do not recommend the use of the Tomahawk tag library.

31.1. TAGS

To use these tags, define the **s** namespace in your page as follows (Facelets only):

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib">
```

The user interface example demonstrates the use of a number of these tags.

31.1.1. Navigation Controls

31.1.1.1. <s:button>

Description

A button that supports invoking an action with control over conversation propagation. *This button does not submit the form.*

Attributes

- **value** — the button label.
- **action** — a method binding that specifies the action listener.
- **view** — specifies the JSF view ID to link to.
- **fragment** — specifies the fragment identifier to link to.
- **disabled** — specifies whether the link is disabled.
- **propagation** — determines the conversation propagation style: **begin**, **join**, **nest**, **none** or **end**.
- **pageflow** — specifies a pageflow definition to begin. (Effective only when **propagation="begin"** or **propagation="join"** is used.)

Usage

```
<s:button id="cancel" value="Cancel" action="#{hotelBooking.cancel}"/>
```

You can specify both **view** and **action** on **<s:link />**. In this case, the action will be called once the redirect to the specified view has occurred.

The use of action listeners (including the default JSF action listener) is not supported with **<s:button />**.

31.1.1.2. <s:conversationId>

Description

Adds the conversation ID to a JSF link or button, for example:

`<h:commandLink />`, **`<s:button />`**.

Attributes

None.

31.1.1.3. `<s:taskId>`

Description

Adds the task ID to an output link (or similar JSF control) when the task is available via **`#{task}`**.

Attributes

None.

31.1.1.4. `<s:link>`

Description

A link that supports invoking an action with control over conversation propagation. *This does not submit the form.*

The use of action listeners (including the default JSF action listener) is not supported with **`<s:link />`**.

Attributes

- **`value`** — specifies the link label.
- **`action`** — a method binding that specifies the action listener.
- **`view`** — specifies the JSF view ID to link to.
- **`fragment`** — specifies the fragment identifier to link to.
- **`disabled`** — specifies whether the link is disabled.
- **`propagation`** — determines the conversation propagation style: **`begin`**, **`join`**, **`nest`**, **`none`** or **`end`**.
- **`pageflow`** — specifies a pageflow definition to begin. (Effective only when using **`propagation="begin"`** or **`propagation="join"`**.)

Usage

```
<s:link id="register" view="/register.xhtml" value="Register New User"/>
```

You can specify both **`view`** and **`action`** on **`<s:link />`**. In this case, the action will be called once the redirect to the specified view has occurred.

31.1.1.5. `<s:conversationPropagation>`

Description

Customizes the conversation propagation for a command link or button (or similar JSF control). *Facelets only.*

Attributes

- **`type`** — determines the conversation propagation style: **`begin`**, **`join`**, **`nest`**, **`none`** or **`end`**.
- **`pageflow`** — specifies a pageflow definition to begin. (Effective only useful when using **`propagation="begin"`** or **`propagation="join"`**.)

Usage

```
<h:commandButton value="Apply" action="#{personHome.update}">
  <s:conversationPropagation type="join" />
</h:commandButton>
```

31.1.1.6. `<s:defaultAction>`

Description

Specifies the default action to run when the form is submitted using the enter key.

Currently you this can only be nested inside buttons, such as **`<h:commandButton />`**, **`<a:commandButton />`** or **`<tr:commandButton />`**).

You must specify an ID on the action source, and only one default action can be specified per form.

Attributes

None.

Usage

```
<h:commandButton id="foo" value="Foo" action="#{manager.foo}">
  <s:defaultAction />
</h:commandButton>
```

31.1.2. Converters and Validators

31.1.2.1. `<s:convertDateTime>`

Description

Perform date or time conversions in the Seam timezone.

Attributes

None.

Usage

```
<h:outputText value="#{item.orderDate}">
  <s:convertDateTime type="both" dateStyle="full"/>
</h:outputText>
```

31.1.2.2. <s:convertEntity>

Description

Assigns an entity converter to the current component. This is useful for radio button and dropdown controls.

The converter works with any managed entity - either simple or composite. If the converter cannot find the items declared in the JSF controls upon form submission, a validation error will occur.

Attributes

None.

Configuration

You must use *Seam managed transactions* (see Section 10.2, “Seam managed transactions”) with **<s:convertEntity />**.

Your *Managed Persistence Context* must be named **entityManager** — if it is not, you can change its named in **components.xml**:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:jpa-entity-loader entity-manager="#{em}" />
```

If you are using a *Managed Hibernate Session*, you must also set this in **components.xml**:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:hibernate-entity-loader />
```

Your *Managed Hibernate Session* must be named **session** — if it is not, you can change its named in **components.xml**:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:hibernate-entity-loader session="#{hibernateSession}" />
```

To use multiple entity managers with the entity converter, create a copy of the entity converter for each entity manager in **components.xml**. The entity converter delegates to the entity loader to perform persistence operations like so:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:entity-converter name="standardEntityConverter"
    entity-loader="#{standardEntityLoader}" />
<ui:jpa-entity-loader name="standardEntityLoader"
    entity-manager="#{standardEntityManager}" />
<ui:entity-converter name="restrictedEntityConverter"
```

```
    entity-loader="#{restrictedEntityLoader}" />
<ui:jpa-entity-loader name="restrictedEntityLoader"
    entity-manager="#{restrictedEntityManager}" />
```

```
<h:selectOneMenu value="#{person.continent}">
  <s:selectItems value="#{continents.resultList}"
     var="continent" label="#{continent.name}" />
  <f:converter converterId="standardEntityConverter" />
</h:selectOneMenu>
```

Usage

```
<h:selectOneMenu value="#{person.continent}" required="true">
  <s:selectItems value="#{continents.resultList}" var="continent"
     label="#{continent.name}" noSelectionLabel="Please Select..."/>
  <s:convertEntity />
</h:selectOneMenu>
```

31.1.2.3. `<s:convertEnum>`

Description

Assigns an enum converter to the current component. This is primarily useful for radio button and dropdown controls.

Attributes

None.

Usage

```
<h:selectOneMenu value="#{person.honorific}">
  <s:selectItems value="#{honorifics}" var="honorific"
     label="#{honorific.label}" noSelectionLabel="Please select" />
  <s:convertEnum />
</h:selectOneMenu>
```

31.1.2.4. `<s:convertAtomicBoolean>`

Description

`javax.faces.convert.Converter` for **`java.util.concurrent.atomic.AtomicBoolean`**.

Attributes

None.

Usage

```
<h:outputText value="#{item.valid}">
  <s:convertAtomicBoolean />
</h:outputText>
```

31.1.2.5. `<s:convertAtomicInteger>`

Description

`javax.faces.convert.Converter` for **`java.util.concurrent.atomic.AtomicInteger`**.

Attributes

None.

Usage

```
<h:outputText value="#{item.id}">
  <s:convertAtomicInteger />
</h:outputText>
```

31.1.2.6. `<s:convertAtomicLong>`

Description

`javax.faces.convert.Converter` for **`java.util.concurrent.atomic.AtomicLong`**.

Attributes

None.

Usage

```
<h:outputText value="#{item.id}">
  <s:convertAtomicLong />
</h:outputText>
```

31.1.2.7. `<s:validateEquality>`

Description

Validates that an input control's parent's value is equal, or not equal, to the referenced control's value.

Attributes

- **`for`** — The ID of a control to validate against.
- **`message`** — Message to show on failure.
- **`required`** — False will disable a check that a value at all is inputted in fields.
- **`messageId`** — Message ID to show on failure.
- **`operator`** — The operator to use when comparing values. Valid operators are:
  - **`equal`** — validates that **`value.equals(forValue)`**.
  - **`not_equal`** — validates that **`!value.equals(forValue)`**
  - **`greater`** — validates that ((Comparable)value).compareTo(forValue) > 0

- **`greater_or_equal`** — validates that ((Comparable)value).compareTo(forValue) >= 0
- **`less`** — validates that ((Comparable)value).compareTo(forValue) < 0
- **`less_or_equal`** — validates that ((Comparable)value).compareTo(forValue) <= 0

Usage

```
<h:inputText id="name" value="#{bean.name}"/>
<h:inputText id="nameVerification" >
  <s:validateEquality for="name" />
</h:inputText>
```

31.1.2.8. `<s:validate>`

Description

A non-visual control that validates a JSF input field against the bound property with the Hibernate Validator.

Attributes

None.

Usage

```
<h:inputText id="userName" required="true" value="#{customer.userName}">
  <s:validate />
</h:inputText>
<h:message for="userName" styleClass="error" />
```

31.1.2.9. `<s:validateAll>`

Description

A non-visual control that validates all child JSF input fields against their bound properties with the Hibernate Validator.

Attributes

None.

Usage

```
<s:validateAll>
  <div class="entry">
    <h:outputLabel for="username">Username:</h:outputLabel>
    <h:inputText id="username" value="#{user.username}" required="true"/>
    <h:message for="username" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="password">Password:</h:outputLabel>
    <h:inputSecret id="password" value="#{user.password}"
     required="true"/>
```

```
    <h:message for="password" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="verify">Verify Password:</h:outputLabel>
    <h:inputSecret id="verify" value="#{register.verify}"
     required="true"/>
    <h:message for="verify" styleClass="error" />
  </div>
</s:validateAll>
```

31.1.3. Formatting

31.1.3.1. `<s:decorate>`

Description

"Decorates" a JSF input field when validation fails or when **`required="true"`** is set.

Attributes

- **`template`** — the Facelets template used to decorate the component.
- **`enclose`** — if **`true`**, the template used to decorate the input field is enclosed by the element specified with the "element" attribute. (By default, this is a **`div`** element.)
- **`element`** — the element enclosing the template that decorates the input field. By default, the template is enclosed with a **`div`** element.

`#{invalid}` and **`#{required}`** are available inside **`s:decorate`**. **`#{required}`** evaluates to **`true`** if the input component being decorated is set to **`required`**. **`#{invalid}`** evaluates to **`true`** if a validation error occurs.

Usage

```
<s:decorate template="edit.xhtml">
  <ui:define name="label">Country:</ui:define>
  <h:inputText value="#{location.country}" required="true"/>
</s:decorate>
```

```
<ui:composition 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:s="http://jboss.com/products/seam/taglib">
  <div>
    <s:label styleClass="#{invalid?'error':''}">
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}">*</s:span>
    </s:label>
    <span class="#{invalid?'error':''}">
      <s:validateAll>
        <ui:insert/>
      </s:validateAll>
    </span>
```

```
    <s:message styleClass="error"/>
  </div>
</ui:composition>
```

31.1.3.2. `<s:div>`

Description

Renders a HTML **`<div>`**.

Attributes

None.

Usage

```
<s:div rendered="#{selectedMember == null}">
  Sorry, but this member does not exist.
</s:div>
```

31.1.3.3. `<s:span>`

Description

Renders a HTML **`<span>`**.

Attributes

- **`title`** — Title for a span.

Usage

```
<s:span styleClass="required" rendered="#{required}" title="Small
tooltip">
  *
</s:span>
```

31.1.3.4. `<s:fragment>`

Description

A non-rendering component useful for enabling/disabling rendering of it's children.

Attributes

None.

Usage

```
<s:fragment rendered="#{auction.highBidder ne null}">
  Current bid:
</s:fragment>
```

31.1.3.5. `<s:label>`

Description

"Decorates" a JSF input field with the label. The label is placed inside the HTML **`<label>`** tag, and is associated with the nearest JSF input component. It is often used with **`<s:decorate>`**.

Attributes

- **`style`** — The control's style.
- **`styleClass`** — The control's style class.

Usage

```
<s:label styleClass="label"> Country: </s:label>
<h:inputText value="#{location.country}" required="true"/>
```

31.1.3.6. `<s:message>`

Description

"Decorates" a JSF input field with the validation error message.

Attributes

None.

Usage

```
<f:facet name="afterInvalidField">
  <s:span>
    Error:
    <s:message/>
  </s:span>
</f:facet>
```

31.1.4. Seam Text

31.1.4.1. `<s:validateFormattedText>`

Description

Checks that the submitted value is valid Seam Text.

Attributes

None.

31.1.4.2. `<s:formattedText>`

Description

Outputs *Seam Text*, a rich text markup useful for blogs, wikis and other applications that might use rich text. See the Seam Text chapter for full usage.

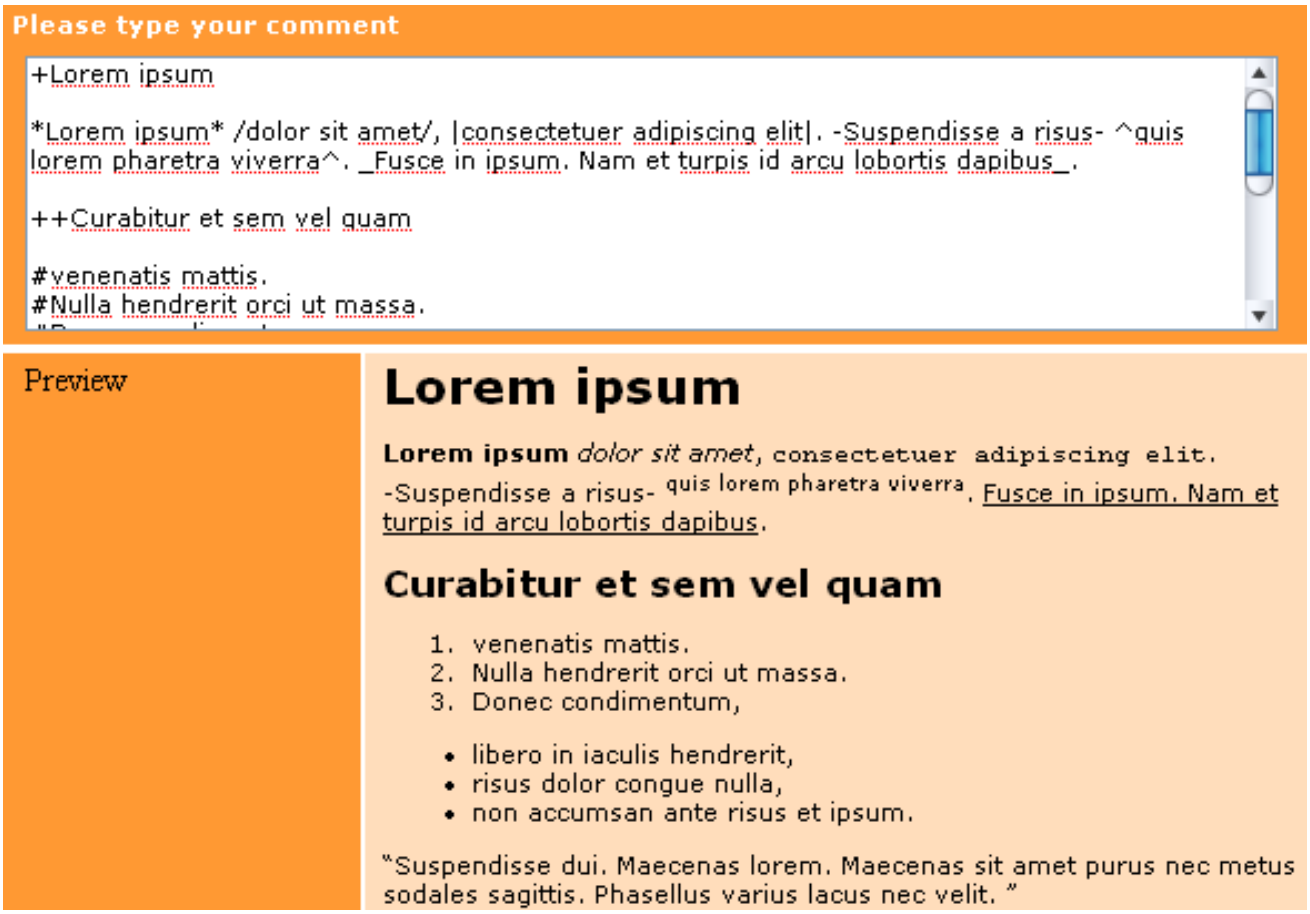
Attributes

- **`value`** — an EL expression specifying the rich text markup to render.

Usage

```
<s:formattedText value="#{blog.text}"/>
```

Example



31.1.5. Form support

31.1.5.1. `<s:token>`

Description

Produces a random token to insert into a hidden form field in order to secure JSF form posts against Cross-Site Request Forgery (XSRF) attacks. The browser must have cookies enabled to submit forms that include this component.

Attributes

- **`requireSession`** — indicates whether the session ID should be included in the form signature to bind the token to the session. The default value is **`false`**, but this should only be used if Facelets is in "build before restore" mode. ("Build before restore" is the default mode in JSF 2.0.)

- **`enableCookieNotice`** — indicates that a JavaScript check should be inserted into the page to verify that cookies are enabled in the browser. If cookies are not enabled, present a notice to the user that form posts will not work. The default value is **`false`**.
- **`allowMultiplePosts`** — indicates whether the same form is allowed to submit multiple times with the same signature (where the view has not changed). This is often required when the form is performing AJAX calls without rerendering itself or the UIToken component. It is better to rerender the UIToken component upon any AJAX call where the UIToken component would be processed. The default value is **`false`**.

Usage

```
<h:form>
  <s:token enableCookieNotice="true" requireSession="false"/>
  ...
</h:form>
```

31.1.5.2. `<s:enumItem>`

Description

Creates a **`SelectItem`** from an enum value.

Attributes

- **`enumValue`** — the string representation of the enum value.
- **`label`** — the label to be used when rendering the **`SelectItem`**.

Usage

```
<h:selectOneRadio id="radioList"
                  layout="lineDirection"
 value="#{newPayment.paymentFrequency}">
   <s:convertEnum />
   <s:enumItem enumValue="ONCE"         label="Only Once" />
   <s:enumItem enumValue="EVERY_MINUTE" label="Every Minute" />
   <s:enumItem enumValue="HOURLY"       label="Every Hour" />
   <s:enumItem enumValue="DAILY"        label="Every Day" />
   <s:enumItem enumValue="WEEKLY"       label="Every Week" />
</h:selectOneRadio>
```

31.1.5.3. `<s:selectItems>`

Description

Creates a **`List<SelectItem>`** from a List, Set, DataModel or Array.

Attributes

- **`value`** — an EL expression specifying the data that backs the **`ListSelectItem>`**
- **`var`** — defines the name of the local variable that holds the current object during iteration.

- **`label`** — the label to be used when rendering the **`SelectItem`**. Can reference the **`var`** variable.
- **`itemValue`** — specifies the value to return to the server if this option is selected. This is an optional attribute. If included, **`var`** is the default object used. Can reference the **`var`** variable.
- **`disabled`** — if this is set to **`true`**, the **`SelectItem`** will be rendered disabled. Can reference the **`var`** variable.
- **`noSelectionLabel`** — specifies the (optional) label to place at the top of list. If **`required="true"`** is also specified then selecting this value will cause a validation error.
- **`hideNoSelectionLabel`** — if true, the **`noSelectionLabel`** will be hidden when a value is selected.

Usage

```
<h:selectOneMenu value="#{person.age}" converter="ageConverter">
  <s:selectItems value="#{ages}" var="age" label="#{age}" />
</h:selectOneMenu>
```

31.1.5.4. `<s:fileUpload>`

Description

Renders a file upload control. This control must be used within a form with an encoding type of **`multipart/form-data`**:

```
<h:form enctype="multipart/form-data">
```

For multipart requests, the Seam Multipart servlet filter must also be configured in **`web.xml`**:

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Configuration

The following configuration options for multipart requests can be configured in **`components.xml`**:

- **`createTempFiles`** — if this option is set to **`true`**, uploaded files are streamed to a temporary file rather than being held in memory.
- **`maxRequestSize`** — the maximum size of a file upload request, in bytes.

Here's an example:

```
<component class="org.jboss.seam.web.MultipartFilter">
  <property name="createTempFiles">true</property>
```

```
  <property name="maxRequestSize">1000000</property>
</component>
```

Attributes

- **`data`** — specifies the value binding that receives the binary file data. The receiving field must be declared as either a **`byte[]`** or **`InputStream`**.
- **`contentType`** — an optional attribute specifying the value binding that receives the file's content type.
- **`fileName`** — an optional attribute specifying the value binding that receives the filename.
- **`fileSize`** — an optional attribute specifying the value binding that receives the file size.
- **`accept`** — a comma-separated list of acceptable content types, for example, **`"images/png,images/jpg"`**, **`"images/*"`**. The types listed may not be supported by the browser.
- **`style`** — The control's style.
- **`styleClass`** — The control's style class.

Usage

```
<s:fileUpload id="picture"
              data="#{register.picture}" accept="image/png"
              contentType="#{register.pictureContentType}" />
```

31.1.6. Other

31.1.6.1. `<s:cache>`

Description

Caches the rendered page fragment using JBoss Cache. Note that **`<s:cache>`** actually uses the instance of JBoss Cache managed by the built-in **`pojoCache`** component.

Attributes

- **`key`** — the key to cache rendered content, often a value expression. For example, if we were caching a page fragment that displays a document, we might use **`key="Document-#{document.id}"`**.
- **`enabled`** — a value expression that determines whether the cache should be used.
- **`region`** — specifies the JBoss Cache node to use. Different nodes can have different expiry policies.

Usage

```
<s:cache key="entry-#{blogEntry.id}" region="pageFragments">
  <div class="blogEntry">
    <h3>#{blogEntry.title}</h3>
```

```
    <div>
      <s:formattedText value="#{blogEntry.body}"/>
    </div>
    <p>
      [Posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timezone="#{blog.timeZone}"
         locale="#{blog.locale}" type="both"/>
      </h:outputText>]
    </p>
  </div>
</s:cache>
```

31.1.6.2. `<s:resource>`

Description

A tag that acts as a file download provider. It must be alone in the JSF page. To use this control, you must configure **`web.xml`** as follows:

Configuration

```
<servlet>
  <servlet-name>Document Store Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.document.DocumentStoreServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Document Store Servlet</servlet-name>
  <url-pattern>/seam/docstore/*</url-pattern>
</servlet-mapping>
```

Attributes

- **`data`** — specifies data that should be downloaded. May be a java.util.File, an InputStream or a byte array.
- **`fileName`** — the filename of the file to be served.
- **`contentType`** — the content type of the file to be downloaded.
- **`disposition`** — the disposition to use. The default disposition is **`inline`**.

Usage

The tag is used as follows:

```
<s:resource xmlns="http://www.w3.org/1999/xhtml"
   xmlns:s="http://jboss.com/products/seam/taglib"
   data="#{resources.data}" contentType="#{resources.contentType}"
   fileName="#{resources.fileName}" />
```

Here, the bean named **`resources`** is some backing bean that, given some request parameters, serves a specific file — see **`s:download`**.

31.1.6.3. `<s:download>`

Description

Builds a RESTful link to a **`<s:resource>`**. Nested **`f:param`** build up the url.

- **`src`** — Resource file serving files.

Attributes

```
<s:download src="/resources.xhtml">
  <f:param name="fileId" value="#{someBean.downloadableFileId}"/>
</s:download>
```

This produces a link of a similar form to the following: **`http://localhost/resources.seam?fileId=1`**

31.1.6.4. `<s:graphicImage>`

Description

An extended **`<h:graphicImage>`** that allows the image to be created in a Seam Component. It is possible to transform the image further.

All **`<h:graphicImage>`** attributes are supported, in addition to:

Attributes

- **`value`** — specifies the image to display. Can be a path **`String`** (loaded from the classpath), a **`byte[]`**, a **`java.io.File`**, a **`java.io.InputStream`** or a **`java.net.URL`**. Currently supported image formats are **`image/bmp`**, **`image/png`**, **`image/jpeg`** and **`image/gif`**.
- **`fileName`** — specifies the filename of the image. This name should be unique. If left unspecified, a unique filename will be generated for the image.

Transformations

To transform the image, nest a tag specifying which transformation to apply. Seam currently supports the following transformation tags:

`<s:transformImageSize>`

- **`width`** — specifies the new width of the image.
- **`height`** — specifies the new height of the image.
- **`maintainRatio`** — if **`true`**, and one of either **`width`** or **`height`** is specified, the image will be resized to maintain the **`height`**:**`width`** aspect ratio.
- **`factor`** — scales the image by the specified factor.

`<s:transformImageBlur>`

- **`radius`** — performs a convolution blur with the specified radius.

<s:transformImageType>

- **contentType** — alters the image type to either **image/jpeg** or **image/png**.

You can also create your own image transformation. Create a **UIComponent** that implements **org.jboss.seam.ui.graphicImage.ImageTransform**. Inside the **applyTransform()** method, use **image.getBufferedImage()** to get the original image and **image.setBufferedImage()** to set your transformed image. Transforms are applied in the order specified in the view.

Usage

```
<s:graphicImage rendered="#{auction.image ne null}"
   value="#{auction.image.data}">
  <s:transformImageSize width="200" maintainRatio="true"/>
</s:graphicImage>
```

31.1.6.5. <s:remote>

Description

Generates the Javascript stubs required to use Seam Remoting.

Attributes

- **include** — a comma-separated list of the component names (or fully qualified class names) for which to generate Seam Remoting Javascript stubs. See Chapter 24, *Remoting* for more details.

Usage

```
<s:remote include="customerAction,accountAction,com.acme.MyBean"/>
```

31.2. ANNOTATIONS

Seam also provides annotations to let you use Seam components as JSF converters and validators:

@Converter

```
@Name("itemConverter")
@BypassInterceptors
@Converter
public class ItemConverter implements Converter {
  @Transactional
  public Object getAsObject(FacesContext context, UIComponent cmp,
                            String value) {
    EntityManager entityManager =
      (EntityManager) Component.getInstance("entityManager");
    entityManager.joinTransaction(); // Do the conversion
  }
  public String getAsString(FacesContext context, UIComponent cmp,
                            Object value) {
    // Do the conversion
  }
}
```

```
<h:inputText value="#{shop.item}" converter="itemConverter" />
```

Registers the Seam component as a JSF converter. Here, the converter accesses the JPA EntityManager inside a JTA transaction when converting the value back to its object representation.

@Validator

```
@Name("itemValidator")
@BypassInterceptors
@org.jboss.seam.annotations.faces.Validator
public class ItemValidator implements javax.faces.validator.Validator {
  public void validate(FacesContext context, UIComponent cmp,
                       Object value) throws ValidatorException {
    ItemController ItemController =
      (ItemController) Component.getInstance("itemController");
    boolean valid = itemController.validate(value);
    if (!valid) {
      throw ValidatorException("Invalid value " + value);
    }
  }
}
```

```
<h:inputText value="#{shop.item}" validator="itemValidator" />
```

Registers the Seam component as a JSF validator. Here, the validator injects another Seam component; the injected component is used to validate the value.

CHAPTER 32. JBOSS EL

Seam uses JBoss EL to provide an extension to the standard Unified Expression Language (EL). This provides several enhancements to the expressiveness and power of EL expressions.

32.1. PARAMETERIZED EXPRESSIONS

Standard EL does not allow methods to be used with user-defined parameters, but JBoss EL removes this restriction. For example:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel)}"
                 value="Book Hotel"/>
```

```
@Name("hotelBooking")
public class HotelBooking {
  public String bookHotel(Hotel hotel) {
    // Book the hotel
  }
}
```

32.1.1. Usage

As in method calls from Java, parameters are surrounded by parentheses, and separated by commas:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}"
                 value="Book Hotel"/>
```

Here, the parameters **`hotel`** and **`user`** will be evaluated as value expressions and passed to the **`bookHotel()`** method of the component.

Any value expression can be used as a parameter:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel.id,
                           user.username)}"
                 value="Book Hotel"/>
```

When the page is rendered, the parameter names —**`hotel.id`** and **`user.username`** —are stored, and evaluated as value expressions when the page is submitted. Objects cannot be passed as parameters.

Parameters must be available both when the page is rendered and when it is submitted. If the arguments cannot be resolved at page submission time, the action method will be called with **`null`** arguments.

You can also pass literal strings using single quotes:

```
<h:commandLink action="#{printer.println('Hello world!')}"
               value="Hello"/>
```

Unified EL also supports value expressions, which are used to bind a field to a backing bean. Value expressions use JavaBean naming conventions and expect a getter/setter pair. JSF often expects a value expression where only retrieval (get) is required (for example, in the **`rendered`** attribute), but many objects do not have appropriately named property accessors, or do not require parameters.

JBoss EL removes this restriction by allowing values to be retrieved using the method syntax. For example:

```
<h:outputText value="#{person.name}"
              rendered="#{person.name.length() > 5}" />
```

You can access the size of a collection in a similar manner:

```
#{searchResults.size()}
```

In general, any expression of the form **#{obj.property}** would be identical to the expression **#{obj.getProperty()}**.

Parameters are also allowed. The following example calls the **productsByColorMethod** with a literal string argument:

```
#{controller.productsByColor('blue')}
```

32.1.2. Limitations and Hints

JBoss EL does have several limitations:

- *Incompatibility with JSP 2.1* —JBoss EL cannot currently be used with JSP 2.1, because the compiler rejects expressions that include parameters. You will need Facelets if you want to use this extension with JSF 1.2. The extension works correctly with JSP 2.0.
- *Use inside iterative components* —Components like **<c:forEach />** and **<ui:repeat />** iterate over a list or array, exposing each item in the list to nested components. This is effective if you are selecting a row with a **<h:commandButton />** or **<h:commandLink />** like so:

  ```
  @Factory("items")
  public List<Item> getItems() {
    return entityManager.createQuery("select ...").getResultList();
  }
  ```

  ```
  <h:dataTable value="#{items}" var="item">
    <h:column>
      <h:commandLink value="Select #{item.name}"
         action="#{itemSelector.select(item})" />
    </h:column>
  </h:dataTable>
  ```

  However, if you want to use **<s:link />** or **<s:button />** you must expose the items as a **DataModel**, and use a **<dataTable />** (or equivalent from a component set like **<rich:dataTable />**). Neither **<s:link />** or **<s:button />** submit the form, so they do not produce a bookmarkable link. An additional parameter is required to recreate the item when the action method is called. This parameter can only be added when a data table backed by a **DataModel** is used.

- *Calling a **MethodExpression** from Java code* —Normally, when a **MethodExpression** is created, the parameter types are passed in by JSF. However, in a method binding, JSF assumes that there are no parameters to pass. With this extension, there is no way to know the parameter types prior to expression evaluation. This has two minor consequences:

- When you invoke a **`MethodExpression`** in Java code, parameters you pass may be ignored. Parameters defined in the expression will take precedence.
- Ordinarily, it is safe to call **`methodExpression.getMethodInfo().getParamTypes()`** at any time. For an expression with parameters, you must first invoke the **`MethodExpression`** before calling **`getParamTypes()`**.

Both of these cases are exceedingly rare and only apply when you want to invoke the **`MethodExpression`** by hand in Java code.

32.2. PROJECTION

JBoss EL supports a limited projection syntax. A projection expression maps a sub-expression across a multi-valued (list, set, etc...) expression. For instance, the expression:

```
#{company.departments}
```

might return a list of departments. If you only need a list of department names, you must iterate over the list to retrieve the values. JBoss EL allows this with a projection expression:

```
#{company.departments.{d|d.name}}
```

The sub-expression is enclosed in braces. In this example, the expression **`d.name`** is evaluated for each department, using **`d`** as an alias to the department object. The result of this expression will be a list of String values.

Any valid expression can be used in an expression, so —assuming you would use department names of all lengths in a company —it would also be valid to write the following:

```
#{company.departments.{d|d.size()}}
```

Projections can be nested. The following expression returns the last names of every employee in every department:

```
#{company.departments.{d|d.employees.{emp|emp.lastName}}}
```

Nested projections can be slightly tricky, however. The following expression appears to return a list of all employees in all departments:

```
#{company.departments.{d|d.employees}}
```

However, it actually returns a list containing a list of the employees for each individual department. To combine the values, it is necessary to use a slightly longer expression:

```
#{company.departments.{d|d.employees.{e|e}}}
```

This syntax cannot be parsed by either Facelets or JSP, so it cannot be used in XHTML or JSP files. Future versions of JBoss EL may accommodate the projection syntax more easily.

CHAPTER 33. CLUSTERING AND EJB PASSIVATION

Web clustering and EJB passivation share a common solution in Seam, so they are addressed together. This chapter focuses on the programming model and how it is affected by the use of clustering and EJB passivation. You will learn how Seam passivates component and entity instances, how passivation relates to clustering, and how to enable passivation. You will also learn how to deploy a Seam application into a cluster and verify that HTTP session replication is working correctly.

First, we will take you through some background information on clustering and show you an example deployment of a Seam application to an EAP cluster.

33.1. CLUSTERING

Clustering, also known as *web clustering*, lets an application run on two or more parallel servers (*nodes*), while providing a client with a uniform view of the application. Load is distributed across servers such that if one or more servers fail, the application can still be accessed through any surviving nodes. This is crucial for scalable enterprise applications, since performance and availability can be improved just by adding nodes. However, this also provokes the question of what happens to state held on a server that fails.

So far, you know that Seam provides state management by including additional scopes and governing the life cycles of stateful (scoped) components. But Seam's state management goes beyond creating, storing, and destroying instances. Seam tracks changes to JavaBean components and stores those changes at strategic points during a request so that changes can be restored when the request shifts to a secondary node in the cluster. Monitoring and replicating stateful EJB components is already handled by the EJB server; Seam state management provides the same feature for stateful JavaBeans.

In addition to monitoring JavaBean components, Seam ensures that managed entity instances (that is, JPA and Hibernate entities) are not attached during replication. Seam keeps a record of entities to be loaded, and loads them automatically on the secondary node. *You must be using a Seam-managed persistence context to use this feature.* More information is provided later in this chapter.

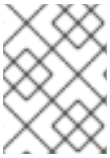
Next we will take you through how to program for clustering.

33.1.1. Programming for clustering

Any session- or conversation-scoped mutable JavaBean component to be used in a clustered environment must implement the **`org.jboss.seam.core.Mutable`** interface from the Seam API. As part of the contract, the component must maintain a **`dirtyflag`** event, which indicates whether the user has made changes to the form that must be saved. This event is reported and reset by the **`clearDirty()`** method, which is called to determine whether the component must be replicated. This lets you avoid using the Servlet API to add and remove the session attribute for every change to an object.

You must also make sure that all session- and conversation-scoped JavaBean components are *serializable*. All fields of a stateful component (EJB or JavaBean) must be serializable, unless marked *transient* or set to null in a **`@PrePassivate`** method. You can restore the value of a transient or nullified field in a **`@PostActivate`** method.

One area that can be problematic is in using **`List.subList`** to create a list, because the list created is not serializable. A similar situation can occur for a number of methods that create objects automatically. If you encounter a **`java.io.NotSerializableException`**, you can place a breakpoint on this exception and run the application server in debug mode to find the problem method.



NOTE

Clustering does not work with components that are hot-deployed. Further, you should avoid using hot-deployable components in non-development environments.

33.1.2. Deploying a Seam application to a EAP cluster with session replication

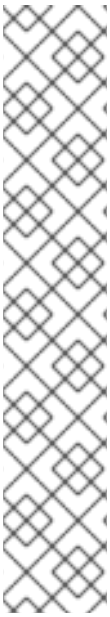
The following procedure has been validated against a **seam-gen** application and the Seam Booking example.

This section assumes that the IP addresses of the master and slave servers are **192.168.1.2** and **192.168.1.3**, respectively. The **mod_jk** load balancer was not use intentionally to make it easier to validate that both nodes are responding to requests and interchanging sessions.

The following log messages were generated out of the deployment of a WAR application, **vehicles.war**, and its corresponding datasource, **vehiclesDatasource**. The Booking example fully supports this process, and you can find instructions about deploying it to a cluster in the **examples/booking/readme.txt** file.

These instructions use the farm deployment method, but you can also deploy the application normally and let the two servers negotiate a master/slave relationship based on start order.

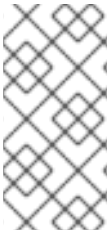
All timestamps in this tutorial have been replaced with zeroes to reduce noise.



NOTE

If your nodes are on different machines that run Red Hat Enterprise Linux or Fedora, they may not acknowledge each other automatically. EAP clustering relies on the UDP (User Datagram Protocol) multi-casting provided by jGroups. The SELinux configuration that ships with Red Hat Enterprise Linux and Fedora blocks these packets by default. To allow the packets, modify the **iptables** rules (as root). The following commands apply to an IP address that matches **192.168.1.x**:

```
  /sbin/iptables -I RH-Firewall-1-INPUT 5 -p udp -d 224.0.0.0/4
-j ACCEPT
  /sbin/iptables -I RH-Firewall-1-INPUT 9 -p udp -s
192.168.1.0/24 -j ACCEPT
  /sbin/iptables -I RH-Firewall-1-INPUT 10 -p tcp -s
192.168.1.0/24 -j ACCEPT
  /etc/init.d/iptables save
```



NOTE

If you are deploying an application with stateful session beans and HTTP Session replication to a EAP cluster, your stateful session bean classes must be annotated with **@Clustered** (from the JBoss EJB 3.0 annotation API) or marked as **clustered** in the **jboss.xml** descriptor. For details, see the Booking example.

33.1.3. Tutorial

1. Create two instances of JBoss Enterprise Application Platform. (To do so, just extract the **zip** twice.)

Deploy the JDBC driver to **server/all/lib/** on both instances if you are not using HSQLDB.

2. Add **<distributable/>** as the first child element in **WEB-INF/web.xml**.

3. Set the **distributable** property on **org.jboss.seam.core.init** to **true** to enable the **ManagedEntityInterceptor** (that is, **<core:init distributable="true">** in **WEB-INF/components.xml**).

4. Ensure that you have two IP addresses available (two computers, two network cards, or two IP addresses bound to the same interface). We assume that these two IP addresses are **192.168.1.2** and **192.168.1.3**.

5. Start the master JBoss Enterprise Application Platform instance on the first IP:

```
./bin/run.sh -c all -b 192.168.1.2
```

The log should report one cluster member and zero other members.

6. Verify that the **server/all/farm** directory in the slave JBoss Enterprise Application Platform instance is empty.

7. Start the slave JBoss Enterprise Application Platform instance on the second IP:

```
./bin/run.sh -c all -b 192.168.1.3
```

The log should report two cluster members and one other member. It should also show the state being retrieved from the master instance.

8. Deploy the **-ds.xml** to the **server/all/farm** of the master instance.

In the log of the master instance you should see acknowledgement of this deployment. You should see a corresponding message acknowledging deployment to the slave instance.

9. Deploy the application to the server/all/farm directory. You should see acknowledgement of deployment in the log of the master instance after normal application start messages have finished. The slave instance log should show a corresponding message acknowledging deployment. (You may need to wait up to three minutes for the deployed archive to be transferred.)

Your application is now running in a cluster with HTTP Session replication. The next step is to validate that the clustering is working correctly.

33.1.4. Validating the distributable services of an application running in a EAP cluster

Your application now starts successfully on two different JBoss Enterprise Application Platform servers, but it is important to validate that the two instances are exchanging HTTP Sessions correctly, so that the application continues to operate with the slave instance if the master instance is stopped.

First, browse to the application on the master instance to start the first HTTP Session. On the same instance, open the JBoss Enterprise Application Platform JMX Console and navigate to the following managed bean:

- *Category:* jboss.cache

- *Entry:* config=standard-session-cache,service=Cache
- *Method:* printDetails()

Invoke the **`printDetails()`** method. This will present you with a tree of active HTTP Sessions. Verify that the session used by your browser corresponds to one of the sessions on the tree.

Next, switch to the slave instance and invoke the same method in the JMX Console. You should see an identical tree under this application's context path.

That these trees are identical proves that both servers claim to have identical sessions. Next, we must test that the data is serializing and unserializing correctly.

Sign in via the URL of the master instance. Then, construct a URL for the second instance by placing the **`;jsessionid=XXXX`** immediately after the Servlet path and changing the IP address. (You should see that the session has carried over to the other instance.)

Now, kill the master instance and check that you can continue to use the application from the slave instance. Then, remove the deployed applications from the **`server/all/farm`** directory and restart the instance.

Change the IP in the URL back to that of the master instance, and browse to the new URL — you should see that the original session ID is still being used.

You can watch objects passivate and activate by creating a session- or conversation-scoped Seam component and implementing the appropriate life-cycle methods. You can use methods from the **`HttpSessionActivationListener`** interface (which is automatically registered on all non-EJB components):

```
public void sessionWillPassivate(HttpSessionEvent e);
public void sessionDidActivate(HttpSessionEvent e);
```

Alternatively, you can mark two public void methods (without arguments) with **`@PrePassivate`** and **`@PostActivate`** respectively. Remember that passivation will occur at the end of every request, while activation will occur when a node is called.

In order to make replication transparent, Seam automatically keeps track of objects that have been changed. All that you need to do is maintain a **`dirtyflag`** on your session- or conversation-scoped component, and Seam will handle JPA entity instances for you.

33.2. EJB PASSIVATION AND THE MANAGEDENTITYINTERCEPTOR

The **`ManagedEntityInterceptor`** (MEI) is an optional interceptor in Seam. When enabled, it is applied to conversation-scoped components. To enable the MEI, set **`distributable`** to **`true`** on the **`org.jboss.seam.init.core`** component. You can also add or update the following component declaration in your **`components.xml`** file:

```
<core:init distributable="true"/>
```

This does not enable HTTP Session replication, but it does let Seam handle the passivation of either EJB components or components in the HTTP Session.

The MEI ensures that, throughout the life of a conversation with at least one extended persistence context, any entity instances loaded by the persistence context remain managed — that is, they are not prematurely detached by a passivation event. This ensures the integrity of the extended persistence

context, and therefore the integrity of its guarantees.

There are two situations that threaten integrity: the passivation of a stateful session bean that hosts an extended persistence context, and the passivation of the HTTP Session.

33.2.1. The friction between passivation and persistence

The *persistence context* is used to store entity instances (objects) that the persistence manager has loaded from the database. There is only ever one object per unique database record in a persistence context. It is often referred to as the *first-level cache* because it allows an application to avoid a call to the database when a record has been loaded into the persistence context.

Objects in the persistence context can be modified, and once modified they are considered *dirty*. Changes are tracked by the persistence manager, which then migrates these changes to the database when necessary. The persistence context, therefore, maintains a set of pending changes to the database.

Database-oriented applications capture transactional information that must be transferred into the database immediately. This information cannot always be captured in one screen, and the user may need to decide whether to accept or reject the pending changes.

These aspects of transactions have not necessarily been apparent from the user's perspective. The extended persistence context extends the user's understanding of transactions. It can hold changes for as long as the application requires, and then push these pending changes to the database via built-in persistence manager capabilities (**`EntityManager#flush()`**).

The persistence manager is linked to an entity instance via an *object reference*. Entity instances can be serialized, but the persistence manager cannot. Serialization can occur when either a stateful session bean or the HTTP Session is passivated. For the application to continue its activity, the relationship between the persistence manager and its entity instances must be maintained. MEI provides this support.

33.2.2. Case #1: Surviving EJB passivation

Conversations were initially designed for stateful session beans because the EJB3 specification defines stateful session beans as the hosts of the extended persistence context. Seam introduces the *Seam-managed persistence context*, which works around a number of limitations in the specification. Both contexts can be used with stateful session beans.

For a stateful session bean to remain active, a client must hold a reference to the stateful session bean. Seam's conversation context is an ideal location for this reference, which means that the stateful session bean remains active for the duration of the conversation context. Further, **`EntityManager`**s that are injected into the stateful session bean with the **`@PersistenceContext(EXTENDED)`** annotation will be bound to the stateful session bean and remain active for the bean's lifetime. **`EntityManager`**s injected with the **`@In`** annotation are maintained by Seam and stored directly in the conversation context, so they remain active for the duration of the *conversation*, independent of the stateful session bean.

The Java EE container can also passivate a stateful session bean, but this method can be problematic. Rather than making the container responsible for this process, after each invocation of the stateful session bean, Seam transfers the reference to the entity instance from the stateful session bean to the current conversation, and therefore into the HTTP Session. This nullifies the associated fields on the stateful session bean. Seam then restores these references at the beginning of the subsequent invocation. Because Seam already stores the persistence manager in the conversation, stateful session bean passivation and activation has no adverse effect on the application.

IMPORTANT

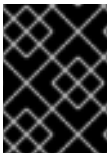
If your application uses stateful session beans that hold references to extended persistence contexts, and those beans can passivate, then you *must* use the MEI, regardless of whether you use a single instance or a cluster.

You can disable passivation on stateful session beans. See the Ejb3DisableSfsbPassivation page on the JBoss Wiki for details.

33.2.3. Case #2: Surviving HTTP session replication

The HTTP Session is used when passivating stateful session beans, but in clustered environments that have enabled session replication, the HTTP Session can also be passivated. Since the persistence manager cannot be serialized, passivating the HTTP Session would normally involve destroying the persistence manager.

When an entity instance is first placed into a conversation, Seam embeds the instance in a wrapper that contains information about reassociating the instance with the persistence manager post-serialization. When the application changes nodes (when a server fails, for instance), Seam's MEI reconstructs the persistence context. The persistence context is reconstructed from the database, so pending changes to the instance are lost. However, Seam's optimistic locking ensures that, where files have changed, only the most recent changes are accepted where multiple versions of an instance occur.



IMPORTANT

If your application is deployed in a cluster with HTTP Session replication, you *must* use the MEI.

CHAPTER 34. PERFORMANCE TUNING

This chapter contains tips for getting the best performance from your Seam application.

34.1. BYPASSING INTERCEPTORS

For repetitive value bindings such as those found in JavaServer Faces (JSF) dataTables, or in iterative controls such as **`ui:repeat`**, the full interceptor stack is invoked upon each invocation of the referenced Seam component. This can substantially decrease performance, particularly if the component is accessed many times. You can improve performance by disabling the interceptor stack for the invoked Seam component —annotate the component class with **`@BypassInterceptors`**.

WARNING

Before you disable the interceptors, note that any component marked with **`@BypassInterceptors`** cannot use features such as bijection, annotated security restrictions, or synchronization. However, you can usually compensate for the loss of these features —for example, instead of injecting a component with **`@In`**, you can use **`Component.getInstance()`** instead.

The following code listing demonstrates a Seam component with its interceptors disabled:

```
@Name("foo")
@Scope(EVENT)
@BypassInterceptors
public class Foo {
  public String getRowActions() {
    // Role-based security check performed inline instead of using
    // @Restrict or other security annotation
    Identity.instance().checkRole("user");

    // Inline code to lookup component instead of using @In
    Bar bar = (Bar) Component.getInstance("bar");

    String actions;
    // some code here that does something
    return actions;
  }
}
```

CHAPTER 35. TESTING SEAM APPLICATIONS

Most Seam applications will require at least two kinds of automated tests: *unit tests*, which test a particular Seam component in isolation, and scripted *integration tests*, which exercise all Java layers of the application (that is, everything except the view pages).

Both test types are easily written.

35.1. UNIT TESTING SEAM COMPONENTS

All Seam components are POJOs (Plain Old Java Objects), which simplifies unit testing. Since Seam also emphasizes the use of bijection for component interaction and contextual object access, testing Seam components outside their normal runtime environments is very easy.

The following Seam Component which creates a statement of account for a customer:

```
@Stateless
@Scope(EVENT)
@Name("statementOfAccount")
public class StatementOfAccount {

  @In(create=true) EntityManager entityManager

  private double statementTotal;

  @In
  private Customer customer;

  @Create
  public void create() {
    List<Invoice> invoices = entityManager
      .createQuery("select invoice from Invoice invoice where " +
                   "invoice.customer = :customer")
      .setParameter("customer", customer)
      .getResultList();
    statementTotal = calculateTotal(invoices);
  }

  public double calculateTotal(List<Invoice> invoices) {
    double total = 0.0;
      for (Invoice invoice: invoices) {
        double += invoice.getTotal();
      }
      return total;
     }
   // getter and setter for statementTotal
}
```

We can test the **`calculateTotal`** method, which tests the component's business logic, as follows:

```
public class StatementOfAccountTest {
  @Test
  public testCalculateTotal {
    List<Invoice> invoices =
      generateTestInvoices(); // A test data generator
```

```
    double statementTotal =
      new StatementOfAccount().calculateTotal(invoices);
    assert statementTotal = 123.45;
  }
}
```

Note that we are not testing data retrieval or persistence, or any of the functions provided by Seam. Here, we are testing only the logic of our POJOs. Seam components do not usually depend directly upon container infrastructure, so most unit tests are just as easy.

If you do want to test the entire application, read the section following.

35.2. INTEGRATION TESTING SEAM COMPONENTS

Integration testing is more complicated. We cannot eliminate the container infrastructure, but neither do we want to deploy our application to an application server to run automated tests. Therefore, our testing environment must replicate enough container infrastructure that we can exercise the entire application, without impacting performance too heavily.

Seam lets you use the JBoss Embedded container to test your components — see the configuration chapter for details. This means you can write tests to exercise your application fully within a minimized container:

```
public class RegisterTest extends SeamTest {

  @Test
  public void testRegisterComponent() throws Exception {

    new ComponentTest() {

      protected void testComponents() throws Exception {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
        assert invokeMethod("#{register.register}").equals("success");
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }

    }.run();

  }

  ...

}
```

35.2.1. Using mocks in integration tests

You may need to replace Seam components requiring resources that are unavailable in the integration test environment. For example, suppose that you use the following Seam component as a facade to some payment processing system:

```
@Name("paymentProcessor")
```

```
public class PaymentProcessor {
  public boolean processPayment(Payment payment) { .... }
}
```

For integration tests, we can make a mock component like so:

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
  public boolean processPayment(Payment payment) {
    return true;
  }
}
```

The **MOCK** precedence is higher than the default precedence of application components, so Seam will install the mock implementation whenever it is in the classpath. When deployed into production, the mock implementation is absent, so the real component will be installed.

35.3. INTEGRATION TESTING SEAM APPLICATION USER INTERACTIONS

It is more difficult to emulate user interactions, and to place assertions appropriately. Some test frameworks let us test the whole application by reproducing user interactions with the web browser. These are useful, but not appropriate during development.

SeamTest lets you write *scripted* tests in a simulated JSF environment. A scripted test reproduces the interaction between the view and the Seam components, so you play the role of the JSF implementation during testing. You can test everything but the view with this approach.

Consider a JSP view for the component we unit tested above:

```
<html>
  <head>
    <title>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <table border="0">
          <tr>
            <td>Username</td>
            <td><h:inputText value="#{user.username}"/></td>
          </tr>
          <tr>
            <td>Real Name</td>
            <td><h:inputText value="#{user.name}"/></td>
          </tr>
          <tr>
            <td>Password</td>
            <td><h:inputSecret value="#{user.password}"/></td>
          </tr>
        </table>
        <h:messages/>
        <h:commandButton type="submit" value="Register"
```

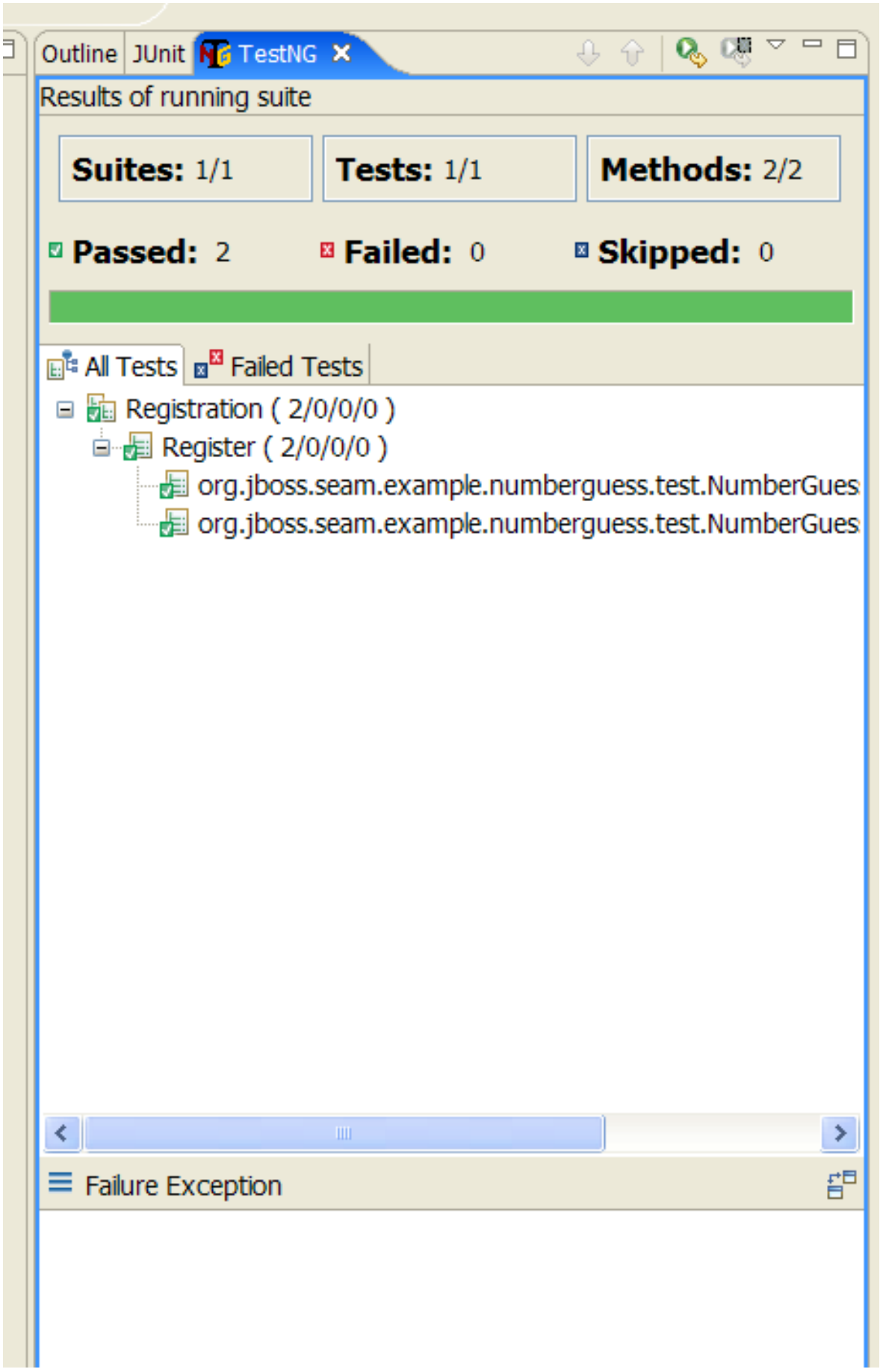
```
                          action="#{register.register}"/>
      </h:form>
    </f:view>
  </body>
</html>
```

We want to test the registration functionality of our application (that is, what happens when a user clicks the Register button). We will reproduce the JSF request life cycle in an automated TestNG test:

```
public class RegisterTest extends SeamTesFt {

  @Test
  public void testRegister() throws Exception {

    new FacesRequest() {

      @Override
      protected void processValidations() throws Exception {
        validateValue("#{user.username}", "1ovthafew");
        validateValue("#{user.name}", "Gavin King");
        validateValue("#{user.password}", "secret");
        assert !isValidationFailure();
      }

      @Override
      protected void updateModelValues() throws Exception {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
      }

      @Override
      protected void invokeApplication() {
        assert invokeMethod("#{register.register}").equals("success");
      }

      @Override
      protected void renderResponse() {
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }
    }.run();
  }
  ...
}
```

Here, we extend **`SeamTest`** to provide a Seam environment for our components, and our test script is written as an anonymous class that extends **`SeamTest.FacesRequest`**, which provides an emulated JSF request life cycle. (There is also a **`SeamTest.NonFacesRequest`** for testing GET requests.) Our code includes methods named for various JSF phases, to emulate the calls that JSF would make to our components. We have then included various assertions.

The Seam example applications include integration tests demonstrating more complex cases. You can run these tests with Ant, or with the TestNG plug-in for Eclipse:



35.3.1. Configuration

If you created your project with **seam-gen**, you can start writing tests immediately. Otherwise, you must first set up a testing environment in a build tool such as Ant, Maven, or Eclipse.

You will require at *least* the following dependencies:

Table 35.1.

Group ID	Artifact ID	Location in Seam
`org.jboss.seam.embedded`	`hibernate-all`	`lib/test/hibernate-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-all`	`lib/test/jboss-embedded-all.jar`
`org.jboss.seam.embedded`	`thirdparty-all`	`lib/test/thirdparty-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-api`	`lib/jboss-embedded-api.jar`
`org.jboss.seam`	`jboss-seam`	`lib/jboss-seam.jar`
`org.jboss.el`	`jboss-el`	`lib/jboss-el.jar`
`javax.faces`	`jsf-api`	`lib/jsf-api.jar`
`javax.el`	`el-api`	`lib/el-api.jar`
`javax.activation`	`javax.activation`	`lib/activation.jar`

Do not put the compile-time JBoss Enterprise Application Platform dependencies from `lib/` (such as `jboss-system.jar`) on the classpath, as this will prevent Embedded JBoss from booting. Add dependencies such as Drools and jBPM as you require them.

You must include the `bootstrap/` directory on the classpath, since it contains the configuration for Embedded JBoss.

You must also include your built project, tests, and the `jar` for your test framework on the classpath, as well as configuration files for JPA and Seam. Seam asks Embedded JBoss to deploy any resource (JAR or directory) with `seam.properties` in its root. If the structure of the directory containing your built project does not resemble that of a deployable archive, you must include `seam.properties` in each resource.

By default, a generated project uses the `java:/DefaultDS` (a built in HSQL datasource in Embedded JBoss) for testing. To use another datasource, place the `foo-ds.xml` into `bootstrap/deploy` directory.

35.3.2. Using SeamTest with another test framework

Seam provides TestNG support out of the box, but you can also use other test frameworks such as JUnit. To do so, you must provide an implementation of `AbstractSeamTest` that does the following:

- Calls `super.begin()` before every test method.

- Calls **`super.end()`** after every test method.
- Calls **`super.setupClass()`** to set up the integration test environment. This should be called prior to any test methods.
- Calls **`super.cleanupClass()`** to clean up the integration test environment.
- Calls **`super.startSeam()`** to start Seam when integration testing begins.
- Calls **`super.stopSeam()`** to cleanly shut down Seam at the end of integration testing.

35.3.3. Integration Testing with Mock Data

To insert or clean data in your database before each test, you can use Seam's integration with DBUnit. To do this, extend DBUnitSeamTest rather than **`SeamTest`**.

You must provide a dataset for DBUnit.

DBUnit supports two formats for dataset files, flat and XML. Seam's DBUnitSeamTest assumes that you use the flat format, so make sure that your dataset is in this format.

```
<dataset>

  <ARTIST
    id="1"
    dtype="Band"
    name="Pink Floyd" />

  <DISC
    id="1"
    name="Dark Side of the Moon"
    artist_id="1" />

</dataset>
```

In your test class, configure your dataset by overriding **`prepareDBUnitOperations()`** as follows:

```
protected void prepareDBUnitOperations() {
  beforeTestOperations.add(
    new DataSetOperation("my/datasets/BaseData.xml")
  );
}
```

`DataSetOperation` defaults to **`DatabaseOperation.CLEAN_INSERT`** if no other operation is specified as a constructor argument. The previous example cleans all tables defined **`BaseData.xml`**, then inserts all rows declared in **`BaseData.xml`** before each **`@Test`** method is invoked.

If you require extra cleanup after a test method executes, add operations to the **`afterTestOperations`** list.

You need to tell DBUnit about your datasource by setting a TestNG test parameter named **`datasourceJndiName`**:

```
<parameter name="datasourceJndiName" value="java:/seamdiscsDatasource"/>
```

DBUnitSeamTest supports both MySQL and HSQL. You must tell it which database is being used, otherwise it defaults to HSQL:

```
  <parameter name="database" value="MYSQL" />
```

It also allows you to insert binary data into the test data set. (Note that this is *untested on Windows*.) Tell DBUnitSeamTest where to find these resources on your classpath:

```
<parameter name="binaryDir" value="images/" />
```

You do not have to configure any of these parameters if you use HSQL and have no binary imports. However, unless you specify **`datasourceJndiName`** in your test configuration, you will have to call **`setDatabaseJndiName()`** before your test runs. If you are not using HSQL or MySQL, you need to override some methods. See the Javadoc of **`DBUnitSeamTest`** for more details.

35.3.4. Integration Testing Seam Mail

WARNING

This feature is still under development.

It is very easy to integration test your Seam Mail:

```
public class MailTest extends SeamTest {

  @Test
  public void testSimpleMessage() throws Exception {

    new FacesRequest() {

      @Override
      protected void updateModelValues() throws Exception {
        setValue("#{person.firstname}", "Pete");
        setValue("#{person.lastname}", "Muir");
        setValue("#{person.address}", "test@example.com");
      }

      @Override
      protected void invokeApplication() throws Exception {
        MimeMessage renderedMessage =
          getRenderedMailMessage("/simple.xhtml");
        assert renderedMessage.getAllRecipients().length == 1;
        InternetAddress to =
          (InternetAddress) renderedMessage.getAllRecipients()[0];
        assert to.getAddress().equals("test@example.com");
      }
```

```
      }.run();
   }
}
```

Create a new **FacesRequest** as normal. Inside the **invokeApplication** hook, we render the message using **getRenderedMailMessage(viewId);**, which passes the **viewId** of he message to be rendered. The method returns the rendered message on which you can perform tests. You can also use any standard JSF life cycle method.

There is no support for rendering standard JSF components, so you cannot easily test the contents of the mail message.

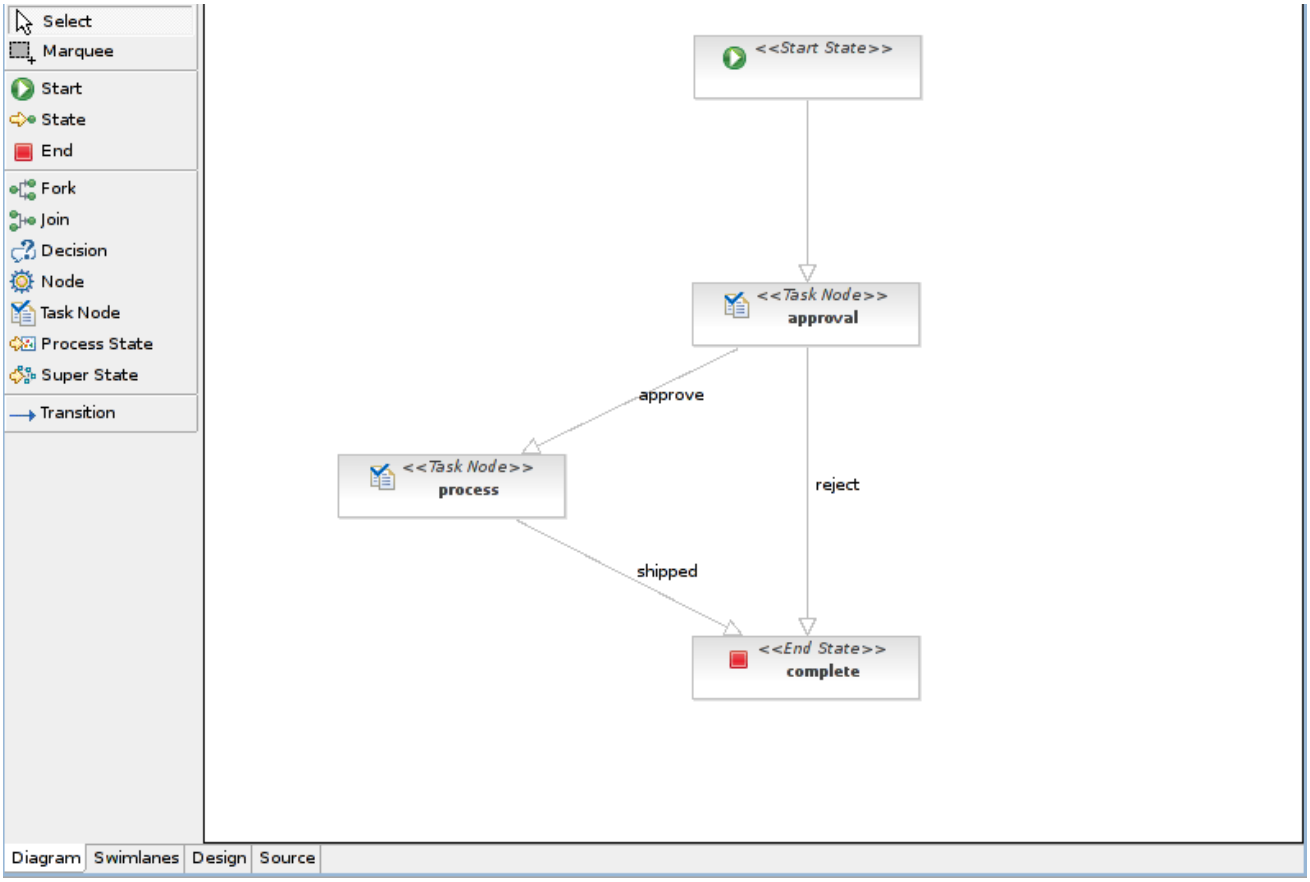
CHAPTER 36. SEAM TOOLS

36.1. JBPM DESIGNER AND VIEWER

The jBPM designer and viewer is included in JBoss Eclipse IDE, and lets you design and view business processes and pageflows aesthetically. Detailed usage information is located in the JBoss Developer Studio Seam Developer Tools Reference Guide available on http://docs.redhat.com/docs/en-US/index.html.

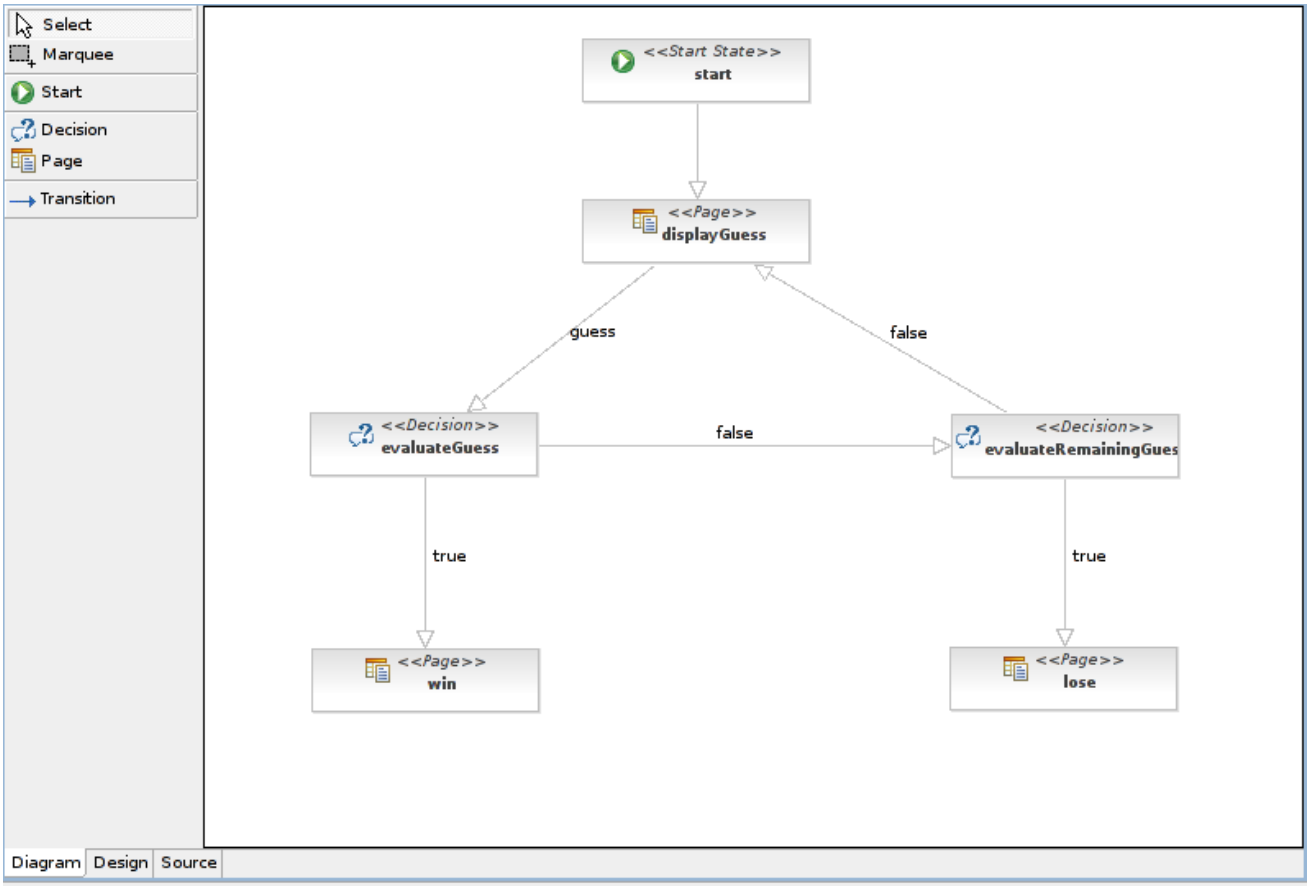
36.1.1. Business process designer

This tool lets you design your own business process graphically.



36.1.2. Pageflow viewer

This tool lets you build graphical representations of pageflows so that complex designs can be shared and compared easily.


Select
Marquee
Start
Decision
Page
Transition
<<Start State>>
start
<<Page>>
displayGuess
guess
false
<<Decision>>
evaluateGuess
false
<<Decision>>
evaluateRemainingGues
true
true
<<Page>>
win
<<Page>>
lose
Diagram
Design
Source

CHAPTER 37. DEPENDENCIES

37.1. JAVA DEVELOPMENT KIT DEPENDENCIES

Seam does not work with JDK™ (Java Development Kit) 1.4, and requires JDK 6 or higher to support annotations and other features. Seam has been thoroughly tested with other JDKs. There are no known issues that are specific to Seam.

37.1.1. Sun's JDK 6 Considerations

The version of JAXB distributed with early versions of JDK 6 was incompatible with Seam, and had to be overridden. The upgrade to JAXB 2.1 (released in JDK 6 Update 4) resolved this issue. When building, testing, or executing, be sure to use this version or higher.

Seam uses Embedded JBoss in its unit and integration testing. When using Embedded JBoss with JDK 6, you must set the following JVM argument: **`-Dsun.lang.ClassLoader.allowArraySyntax=true`**

Seam's internal build system sets this by default when it executes Seam's test suite, but you must set this value manually when using Embedded JBoss.

37.2. PROJECT DEPENDENCIES

This section lists both compilation and runtime dependencies for Seam. For **EAR**-type dependencies, include the library in the **`/lib`** directory of your application's **EAR** file. For **WAR**-type dependencies, include the library in the **`/WEB-INF/lib`** directory of your application's **WAR** file. The scope of each dependency is either *all*, *runtime* or *provided* (by EAP 4.2 or 5.0).

This documentation does not include up-to-date version information and complete dependency information —this information is provided in the **`/dependency-report.txt`** generated from the Maven POM stored in **`/build`**. You can generate this file by running **`ant dependencyReport`**.

37.2.1. Core

Table 37.1.

Name	Scope	Type	Notes
`jboss-seam.jar`	all	**EAR**	The core Seam library. Always required.
`jboss-seam-debug.jar`	runtime	**WAR**	Include during development when enabling Seam's debug feature.
`jboss-seam-ioc.jar`	runtime	**WAR**	Required when using Seam with Spring.
`jboss-seam-pdf.jar`	runtime	**WAR**	Required when using Seam's PDF features.

Name	Scope	Type	Notes
jboss-seam-excel.jar	runtime	**WAR**	Required when using Seam's Microsoft® Excel® features.
jboss-seam-remoting.jar	runtime	**WAR**	Required when using Seam Remoting.
jboss-seam-ui.jar	runtime	**WAR**	Required to use the Seam JavaServer Faces (JSF) controls.
jsf-api.jar	provided		JSF API.
jsf-impl.jar	provided		JSF Reference Implementation.
jsf-facelets.jar	runtime	**WAR**	Facelets.
urlrewritefilter.jar	runtime	**WAR**	URL Rewrite library.
quartz.jar	runtime	**EAR**	Required when using Quartz with Seam's asynchronous features.

37.2.2. RichFaces

Table 37.2. RichFaces dependencies

Name	Scope	Type	Notes
richfaces-api.jar	all	**EAR**	Required to use RichFaces. Provides API classes that can be used from your application, for example, to create a tree.
richfaces-impl.jar	runtime	**WAR**	Required to use RichFaces.
richfaces-ui.jar	runtime	**WAR**	Required to use RichFaces. Provides all the UI components.

37.2.3. Seam Mail

Table 37.3. Seam Mail Dependencies

Name	Scope	Type	Notes
activation.jar	runtime	**EAR**	Required for attachment support.
mail.jar	runtime	**EAR**	Required for outgoing mail support.

Name	Scope	Type	Notes
mail-ra.jar	compile only		Required for incoming mail support.
jboss-seam-mail.jar	runtime	**WAR**	Seam Mail.

37.2.4. Seam PDF

Table 37.4. Seam PDF Dependencies

Name	Type	Scope	Notes
itext.jar	runtime	**WAR**	PDF Library
jfreechart.jar	runtime	**WAR**	Charting library.
jcommon.jar	runtime	**WAR**	Required by JFreeChart.
jboss-seam-pdf.jar	runtime	**WAR**	Seam PDF core library.

37.2.5. Seam Microsoft®Excel®

Table 37.5. Seam Microsoft®Excel® Dependencies

Name	Type	Scope	Notes
jxl.jar	runtime	**WAR**	JExcelAPI library.
jboss-seam-excel.jar	runtime	**WAR**	Seam Microsoft® Excel® core library.

37.2.6. JBoss Rules

The JBoss Rules (Drools) libraries can be found in the **drools/lib** directory in Seam.

Table 37.6. JBoss Rules Dependencies

Name	Scope	Type	Notes
antlr-runtime.jar	runtime	**EAR**	ANTLR Runtime Library.
core.jar	runtime	**EAR**	Eclipse JDT.
drools-api.jar	runtime	**EAR**	

Name	Scope	Type	Notes
`drools-compiler.jar`	runtime	**EAR**	
`drools-core.jar`	runtime	**EAR**	
`drools-decisiontables.jar`	runtime	**EAR**	
`drools-templates.jar`	runtime	**EAR**	
`janino.jar`	runtime	**EAR**	
`mvel2.jar`	runtime	**EAR**	

37.2.7. JBPM

Table 37.7. JBPM dependencies

Name	Scope	Type	Notes
`jbpm-jpdl.jar`	runtime	**EAR**	

37.2.8. GWT

These libraries are required to use the Google Web Toolkit (GWT) with your Seam application.

Table 37.8. GWT dependencies

Name	Scope	Type	Notes
`gwt-servlet.jar`	runtime	**WAR**	The GWT Servlet libraries.

37.2.9. Spring

These libraries are required to use the Spring Framework with your Seam application.

Table 37.9. Spring Framework dependencies

Name	Scope	Type	Notes
`spring.jar`	runtime	**EAR**	The Spring Framework library.

37.2.10. Groovy

These libraries are required to use Groovy with your Seam application.

Table 37.10. Groovy dependencies

Name	Scope	Type	Notes
`groovy-all.jar`	runtime	**EAR**	The Groovy libraries.

APPENDIX A. REVISION HISTORY

Revision 5.2.0-100.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.2.0-100 **Wed 23 Jan 2013** **Russell Dickenson**

Incorporated changes for JBoss Enterprise Application Platform 5.2.0 GA. For information about documentation changes to this guide, refer to *Release Notes 5.2.0*.

Revision 5.1.2-101 **Fri 27 January 2012** **Russell Dickenson**

Bug fixes for Bugzilla: https://bugzilla.redhat.com/show_bug.cgi?id=755367

Revision 5.1.2-100 **Thu 8 December 2011** **Russell Dickenson**

Incorporated changes for JBoss Enterprise Application Platform 5.1.2 GA. For information about documentation changes to this guide, refer to *Release Notes 5.1.2*.