Seam - Contextual Components

A Framework for Enterprise Java

2.3.1.Final

by Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Christian Bauer, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, Denis Forveille, Marek Novotny, and Jozef Hartinger

edited by Samson Kittoli

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), Mark Newton, Steve Ebersole, Michael Courcy (French Translation), Nicola Benaglia (Italian Translation), Stefano Travelli (Italian Translation), Francesco Milesi (Italian Translation), and Japan JBoss User Group (Japanese Translation)

Introduction to JBoss Seam xv
1. Contribute to Seam xix
1. Seam Tutorial 1
1.1. Using the Seam examples 1
1.1.1. Running the examples on JBoss AS 1
1.1.2. Running the example tests 2
1.2. Your first Seam application: the registration example 2
1.2.1. Understanding the code 3
1.2.2. How it works 14
1.3. Clickable lists in Seam: the messages example 15
1.3.1. Understanding the code 15
1.3.2. How it works 21
1.4. Seam and jBPM: the todo list example 21
1.4.1. Understanding the code 22
1.4.2. How it works 29
1.5. Seam pageflow: the numberguess example 30
1.5.1. Understanding the code 30
1.5.2. How it works 37
1.6. A complete Seam application: the Hotel Booking example 38
1.6.1. Introduction 38
1.6.2. Overview of the booking example 40
1.6.3. Understanding Seam conversations 40
1.6.4. The Seam Debug Page 49
1.7. Nested conversations: extending the Hotel Booking example 50
1.7.1. Introduction 50
1.7.2. Understanding Nested Conversations 52
1.8. A complete application featuring Seam and jBPM: the DVD Store example 58
1.9. Bookmarkable URLs with the Blog example 60
1.9.1. Using "pull"-style MVC 61
1.9.2. Bookmarkable search results page 63
1.9.3. Using "push"-style MVC in a RESTful application 66
2. Getting started with Seam, using seam-gen 71
2.1. Before you start 71
2.2. Setting up a new project 72
2.3. Creating a new action 75
2.4. Creating a form with an action 76
2.5. Generating an application from an existing database 77
2.6. Generating an application from existing JPA/EJB3 entities 78
2.7. Deploying the application as an EAR 78
2.8. Seam and incremental hot deployment 78
3. Getting started with Seam, using JBoss Tools 81
3.1. Before you start 81
4. Migration from 2.2 to 2.3 83
4.1. Migration of XML Schemas 83

4.1.1. Seam schema migration 83
4.1.2. Java EE 6 schema changes 85
4.2. Java EE 6 upgrade 86
4.2.1. Using Bean Validation standard instead of Hibernate Validator 86
4.2.2. Migration of JSF 1 to JSF 2 Facelets templates 86
4.2.3. Migration to JPA 2.0 87
4.2.4. Using compatible JNDI for resources 87
4.3. JBoss AS 7.1 deployment 87
4.3.1. Deployment changes 87
4.3.2. Datasource migration 88
4.4. Changes in testing framework 89
4.5. Dependency changes with using Maven 91
4.5.1. Seam Bill of Materials 91
5. The contextual component model 93
5.1. Seam contexts 93
5.1.1. Stateless context 93
5.1.2. Event context 94
5.1.3. Page context 94
5.1.4. Conversation context 94
5.1.5. Session context 95
5.1.6. Business process context 95
5.1.7. Application context 95
5.1.8. Context variables 95
5.1.9. Context search priority 96
5.1.10. Concurrency model 96
5.2. Seam components 97
5.2.1. Stateless session beans 98
5.2.2. Stateful session beans 98
5.2.3. Entity beans 99
5.2.4. JavaBeans 99
5.2.5. Message-driven beans 99
5.2.6. Interception 100
5.2.7. Component names 100
5.2.8. Defining the component scope 102
5.2.9. Components with multiple roles 103
5.2.10. Built-in components 103
5.3. Bijection 104
5.4. Lifecycle methods 107
5.5. Conditional installation 107
5.6. Logging 109
5.7. The Mutable interface and @ReadOnly 110
5.8. Factory and manager components 112
6. Configuring Seam components 115
6.1. Configuring components via property settings 115

6.2. Configuring components via components.xml 115
6.3. Fine-grained configuration files 119
6.4. Configurable property types 120
6.5. Using XML Namespaces 122
7. Events, interceptors and exception handling 127
7.1. Seam events 127
7.2. Page actions 128
7.3. Page parameters 129
7.3.1. Mapping request parameters to the model 129
7.4. Propagating request parameters 130
7.5. URL rewriting with page parameters 131
7.6. Conversion and Validation 132
7.7. Navigation 133
7.8. Fine-grained files for definition of navigation, page actions and parameters 137
7.9. Component-driven events 137
7.10. Contextual events 139
7.11. Seam interceptors 141
7.12. Managing exceptions 143
7.12.1. Exceptions and transactions 143
7.12.2. Enabling Seam exception handling 144
7.12.3. Using annotations for exception handling 144
7.12.4. Using XML for exception handling 145
7.12.5. Some common exceptions 147
8. Conversations and workspace management 149
8.1. Seam's conversation model 149
8.2. Nested conversations 152
8.3. Starting conversations with GET requests 153
8.4. Requiring a long-running conversation 154
8.5. Using <s:link> and <s:button> 155
8.6. Success messages 157
8.7. Natural conversation ids 158
8.8. Creating a natural conversation 158
8.9. Redirecting to a natural conversation 159
8.10. Workspace management 160
8.10.1. Workspace management and JSF navigation 160
8.10.2. Workspace management and jPDL pageflow 161
8.10.3. The conversation switcher 162
8.10.4. The conversation list 162
8.10.5. Breadcrumbs 163
8.11. Conversational components and JSF component bindings 164
8.12. Concurrent calls to conversational components 165
8.12.1. How should we design our conversational AJAX application? 166
8.12.2. Dealing with errors 167
9. Pageflows and business processes 169

9.1. Pageflow in Seam 169
9.1.1. The two navigation models 169
9.1.2. Seam and the back button 173
9.2. Using jPDL pageflows 174
9.2.1. Installing pageflows 174
9.2.2. Starting pageflows 175
9.2.3. Page nodes and transitions 176
9.2.4. Controlling the flow 177
9.2.5. Ending the flow 178
9.2.6. Pageflow composition 178
9.3. Business process management in Seam 178
9.4. Using jPDL business process definitions 180
9.4.1. Installing process definitions 180
9.4.2. Initializing actor ids 180
9.4.3. Initiating a business process 180
9.4.4. Task assignment 181
9.4.5. Task lists 181
9.4.6. Performing a task 182
10. Seam and Object/Relational Mapping 185
10.1. Introduction 185
10.2. Seam managed transactions 186
10.2.1. Disabling Seam-managed transactions 187
10.2.2. Configuring a Seam transaction manager 187
10.2.3. Transaction synchronization 188
10.3. Seam-managed persistence contexts 188
10.3.1. Using a Seam-managed persistence context with JPA 189
10.3.2. Using a Seam-managed Hibernate session 189
10.3.3. Seam-managed persistence contexts and atomic conversations 190
10.4. Using the JPA "delegate" 192
10.5. Using EL in EJB-QL/HQL 193
10.6. Using Hibernate filters 194
11. JSF form validation in Seam 195
12. Groovy integration 203
12.1. Groovy introduction 203
12.2. Writing Seam applications in Groovy 203
12.2.1. Writing Groovy components 203
12.2.2. seam-gen 205
12.3. Deployment 205
12.3.1. Deploying Groovy code 206
12.3.2. Native .groovy file deployment at development time 206
12.3.3. seam-gen 206
13. Writing your presentation layer using Apache Wicket 207
13.1. Adding Seam to your wicket application 207
13.1.1. Bijection 207

13.1.2. Orchestration 208
13.2. Setting up your project 209
13.2.1. Runtime instrumentation 209
13.2.2. Compile-time instrumentation 210
13.2.3. The @SeamWicketComponent annotation 212
13.2.4. Defining the Application 212
14. The Seam Application Framework 215
14.1. Introduction 215
14.2. Home objects 217
14.3. Query objects 222
14.4. Controller objects 225
15. Seam and JBoss Rules 227
15.1. Installing rules 227
15.2. Using rules from a Seam component 230
15.3. Using rules from a jBPM process definition 230
16. Security 233
16.1. Overview 233
16.2. Disabling Security 233
16.3. Authentication 234
16.3.1. Configuring an Authenticator component 234
16.3.2. Writing an authentication method 234
16.3.3. Writing a login form 237
16.3.4. Configuration Summary 238
16.3.5. Remember Me 238
16.3.6. Handling Security Exceptions 241
16.3.7. Login Redirection 242
16.3.8. HTTP Authentication 243
16.3.9. Advanced Authentication Features 244
16.4. Identity Management 244
16.4.1. Configuring IdentityManager 245
16.4.2. JpaIdentityStore 246
16.4.3. LdapIdentityStore 252
16.4.4. Writing your own IdentityStore 254
16.4.5. Authentication with Identity Management 254
16.4.6. Using IdentityManager 254
16.5. Error Messages 259
16.6. Authorization 260
16.6.1. Core concepts 260
16.6.2. Securing components 261
16.6.3. Security in the user interface 263
16.6.4. Securing pages 265
16.6.5. Securing Entities 265
16.6.6. Typesafe Permission Annotations 268
16.6.7. Typesafe Role Annotations 269

16.6.8. The Permission Authorization Model 270
16.6.9. RuleBasedPermissionResolver 273
16.6.10. PersistentPermissionResolver 278
16.7. Permission Management 287
16.7.1. PermissionManager 287
16.7.2. Permission checks for PermissionManager operations 288
16.8. SSL Security 289
16.8.1. Overriding the default ports 290
16.9. CAPTCHA 290
16.9.1. Configuring the CAPTCHA Servlet 290
16.9.2. Adding a CAPTCHA to a form 291
16.9.3. Customising the CAPTCHA algorithm 291
16.10. Security Events 292
16.11. Run As 292
16.12. Extending the Identity component 293
16.13. OpenID 294
16.13.1. Configuring OpenID 294
16.13.2. Presenting an OpenIdDLogin form 295
16.13.3. Logging in immediately 295
16.13.4. Deferring login 296
16.13.5. Logging out 296
17. Internationalization, localization and themes 297
17.1. Internationalizing your app 297
17.1.1. Application server configuration 297
17.1.2. Translated application strings 298
17.1.3. Other encoding settings 298
17.2. Locales 299
17.3. Labels 300
17.3.1. Defining labels 300
17.3.2. Displaying labels 301
17.3.3. Faces messages 302
17.4. Timezones 302
17.5. Themes 303
17.6. Persisting locale and theme preferences via cookies 304
18. Seam Text 305
18.1. Basic fomatting 305
18.2. Entering code and text with special characters 307
18.3. Links 308
18.4. Entering HTML 309
18.5. Using the SeamTextParser 309
19. iText PDF generation 311
19.1. Using PDF Support 311
19.1.1. Creating a document 311
19.1.2. Basic Text Elements 312

19.1.3. Headers and Footers 317
19.1.4. Chapters and Sections 318
19.1.5. Lists 320
19.1.6. Tables 321
19.1.7. Document Constants 324
19.2. Charting 324
19.3. Bar codes 333
19.4. Fill-in-forms 334
19.5. Rendering Swing/AWT components 335
19.6. Configuring iText 336
19.7. Further documentation 337
20. The Microsoft® Excel® spreadsheet application 339
20.1. The Microsoft® Excel® spreadsheet application support 339
20.2. Creating a simple workbook 340
20.3. Workbooks 341
20.4. Worksheets 343
20.5. Columns 347
20.6. Cells 348
20.6.1. Validation 349
20.6.2. Format masks 353
20.7. Formulas 353
20.8. Images 354
20.9. Hyperlinks 355
20.10. Headers and footers 356
20.11. Print areas and titles 358
20.12. Worksheet Commands 359
20.12.1. Grouping 359
20.12.2. Page breaks 360
20.12.3. Merging 361
20.13. Datatable exporter 361
20.14. Fonts and layout 362
20.14.1. Stylesheet links 363
20.14.2. Fonts 363
20.14.3. Borders 364
20.14.4. Background 365
20.14.5. Column settings 365
20.14.6. Cell settings 365
20.14.7. The datatable exporter 366
20.14.8. Layout examples 366
20.14.9. Limitations 366
20.15. Internationalization 366
20.16. Links and further documentation 367
21. RSS support 369
21.1. Installation 369

21.2. Generating feeds 369
21.3. Feeds 370
21.4. Entries 370
21.5. Links and further documentation 371
22. Email 373
22.1. Creating a message 373
22.1.1. Attachments 374
22.1.2. HTML/Text alternative part 376
22.1.3. Multiple recipients 376
22.1.4. Multiple messages 376
22.1.5. Templating 376
22.1.6. Internationalisation 377
22.1.7. Other Headers 378
22.2. Receiving emails 378
22.3. Configuration 379
22.3.1. mailSession 379
22.4. Tags 380
23. Asynchronicity and messaging 383
23.1. Messaging in Seam 383
23.1.1. Configuration 383
23.1.2. Sending messages 384
23.1.3. Receiving messages using a message-driven bean 385
23.1.4. Receiving messages in the client 386
23.2. Asynchronicity 386
23.2.1. Asynchronous methods 387
23.2.2. Asynchronous methods with the Quartz Dispatcher 390
23.2.3. Asynchronous events 393
23.2.4. Handling exceptions from asynchronous calls 393
24. Caching 395
24.1. Using Caching in Seam 396
24.2. Page fragment caching 398
25. Web Services 401
25.1. Configuration and Packaging 401
25.2. Conversational Web Services 401
25.2.1. A Recommended Strategy 402
25.3. An example web service 403
25.4. RESTful HTTP webservices with RESTEasy 405
25.4.1. RESTEasy configuration and request serving 405
25.4.2. Resources as Seam components 408
25.4.3. Securing resources 411
25.4.4. Mapping exceptions to HTTP responses 411
25.4.5. Exposing entities via RESTful API 412
25.4.6. Testing resources and providers 415
26. Remoting 417

26.1. Configuration 417
26.2. The "Seam" object 418
26.2.1. A Hello World example 418
26.2.2. Seam.Component 420
26.2.3. Seam.Remoting 422
26.3. Client Interfaces 422
26.4. The Context 423
26.4.1. Setting and reading the Conversation ID 423
26.4.2. Remote calls within the current conversation scope 423
26.5. Batch Requests 424
26.6. Working with Data types 424
26.6.1. Primitives / Basic Types 424
26.6.2. JavaBeans 425
26.6.3. Dates and Times 425
26.6.4. Enums 425
26.6.5. Collections 426
26.7. Debugging 427
26.8. Handling Exceptions 427
26.9. The Loading Message 427
26.9.1. Changing the message 428
26.9.2. Hiding the loading message 428
26.9.3. A Custom Loading Indicator 428
26.10. Controlling what data is returned 428
26.10.1. Constraining normal fields 429
26.10.2. Constraining Maps and Collections 429
26.10.3. Constraining objects of a specific type 430
26.10.4. Combining Constraints 430
26.11. Transactional Requests 430
26.12. JMS Messaging 431
26.12.1. Configuration 431
26.12.2. Subscribing to a JMS Topic 431
26.12.3. Unsubscribing from a Topic 432
26.12.4. Tuning the Polling Process 432
27. Seam and the Google Web Toolkit 433
27.1. Configuration 433
27.2. Preparing your component 433
27.3. Hooking up a GWT widget to the Seam component 434
27.4. GWT Ant Targets 436
27.5. GWT Maven plugin 437
28. Spring Framework integration 439
28.1. Injecting Seam components into Spring beans 439
28.2. Injecting Spring beans into Seam components 441
28.3. Making a Spring bean into a Seam component 441
28.4. Seam-scoped Spring beans 442

28.5. Using Spring PlatformTransactionManagement 443
28.6. Using a Seam Managed Persistence Context in Spring 444
28.7. Using a Seam Managed Hibernate Session in Spring 446
28.8. Spring Application Context as a Seam Component 446
28.9. Using a Spring TaskExecutor for @Asynchronous 447
29. Guice integration 449
29.1. Creating a hybrid Seam-Guice component 449
29.2. Configuring an injector 450
29.3. Using multiple injectors 451
30. Hibernate Search 453
30.1. Introduction 453
30.2. Configuration 453
30.3. Usage 454
31. Configuring Seam and packaging Seam applications 457
31.1. Basic Seam configuration 457
31.1.1. Integrating Seam with JSF and your servlet container 457
31.1.2. Seam Resource Servlet 459
31.1.3. Seam servlet filters 459
31.1.4. Integrating Seam with your EJB container 464
31.1.5. Don't forget! 468
31.2. Using Alternate JPA Providers 468
31.3. Configuring Seam in Java EE 6 469
31.3.1. Packaging 469
31.4. Configuring Seam without EJB 471
31.4.1. Boostrapping Hibernate in Seam 471
31.4.2. Boostrapping JPA in Seam 472
31.4.3. Packaging 472
31.5. Configuring Seam in Java SE 473
31.6. Configuring jBPM in Seam 473
31.6.1. Packaging 474
31.7. Deployment in JBoss AS 7 475
31.8. Configuring SFSB and Session Timeouts in JBoss AS 7 478
31.9. Running Seam in a Portlet 479
31.10. Deploying custom resources 479
32. Seam annotations 483
32.1. Annotations for component definition 483
32.2. Annotations for bijection 486
32.3. Annotations for component lifecycle methods 490
32.4. Annotations for context demarcation 491
32.5. Annotations for use with Seam JavaBean components in a J2EE environment ... 495
32.6. Annotations for exceptions 496
32.7. Annotations for Seam Remoting 496
32.8. Annotations for Seam interceptors 497
32.9. Annotations for asynchronicity 497

32.10. Annotations for use with JSF 498
32.10.1. Annotations for use with dataTable 499
32.11. Meta-annotations for databinding 500
32.12. Annotations for packaging 500
32.13. Annotations for integrating with the servlet container 501
33. Built-in Seam components 503
33.1. Context injection components 503
33.2. JSF-related components 503
33.3. Utility components 505
33.4. Components for internationalization and themes 506
33.5. Components for controlling conversations 507
33.6. jBPM-related components 508
33.7. Security-related components 510
33.8. JMS-related components 510
33.9. Mail-related components 510
33.10. Infrastructural components 511
33.11. Miscellaneous components 513
33.12. Special components 514
34. Seam JSF controls 517
34.1. Tags 517
34.1.1. Navigation Controls 517
34.1.2. Converters and Validators 520
34.1.3. Formatting 526
34.1.4. Seam Text 529
34.1.5. Form support 530
34.1.6. Other 533
34.2. Annotations 537
35. JBoss EL 539
35.1. Parameterized Expressions 539
35.1.1. Usage 539
35.1.2. Limitations and Hints 541
35.2. Projection 542
36. Clustering and EJB Passivation 545
36.1. Clustering 545
36.1.1. Programming for clustering 546
36.1.2. Deploying a Seam application to a JBoss AS cluster with session replication 546
36.2. EJB Passivation and the ManagedEntityInterceptor 549
36.2.1. The friction between passivation and persistence 550
36.2.2. Case #1: Surviving EJB passivation 551
36.2.3. Case #2: Surviving HTTP session replication 551
36.2.4. ManagedEntityInterceptor wrap-up 552
37. Performance Tuning 553
37.1. Bypassing Interceptors 553

38. Testing Seam applications 555
38.1. Unit testing Seam components 555
38.2. Integration testing Seam components 556
38.2.1. Configuration 558
38.2.2. Using JUnitSeamTest with Arquillian 559
38.2.3. Integration testing Seam application user interactions 561
39. Dependencies 569
39.1. JDK Dependencies 569
39.1.1. Oracle's JDK 6 Considerations 569
39.2. Project Dependencies 569
39.2.1. Core 569
39.2.2. RichFaces 570
39.2.3. Seam Mail 570
39.2.4. Seam PDF 571
39.2.5. Seam Microsoft Excel 571
39.2.6. Seam RSS support 571
39.2.7. Drools 572
39.2.8. JBPM 572
39.2.9. GWT 572
39.2.10. Spring 572
39.2.11. Groovy 573
39.3. Dependency Management using Maven 573

Introduction to JBoss Seam

Seam is an application framework for Enterprise Java. It is inspired by the following principles:

One kind of "stuff"

Seam defines a uniform component model for all business logic in your application. A Seam component may be stateful, with the state associated with any one of several well-defined contexts, including the long-running, persistent, *business process context* and the *conversation context*, which is preserved across multiple web requests in a user interaction.

There is no distinction between presentation tier components and business logic components in Seam. You can layer your application according to whatever architecture you devise, rather than being forced to shoehorn your application logic into an unnatural layering scheme forced upon you by whatever combination of stovepipe frameworks you're using today.

Unlike plain Java EE or Java EE components, Seam components may *simultaneously* access state associated with the web request and state held in transactional resources (without the need to propagate web request state manually via method parameters). You might object that the application layering imposed upon you by the old Java EE platform was a Good Thing. Well, nothing stops you creating an equivalent layered architecture using Seam — the difference is that *you* get to architect your own application and decide what the layers are and how they work together.

Integrate JSF with EJB 3.0

JSF and EJB 3 are two of the best new features of Java EE 5. EJB3 is a brand new component model for server side business and persistence logic. Meanwhile, JSF is a great component model for the presentation tier. Unfortunately, neither component model is able to solve all problems in computing by itself. Indeed, JSF and EJB3 work best used together. But the Java EE 5 specification provides no standard way to integrate the two component models. Fortunately, the creators of both models foresaw this situation and provided standard extension points to allow extension and integration with other frameworks.

Seam unifies the component models of JSF and EJB 3, eliminating glue code, and letting the developer think about the business problem.

It is possible to write Seam applications where "everything" is an EJB. This may come as a surprise if you're used to thinking of EJBs as coarse-grained, so-called "heavyweight" objects. However, version 3.0 has completely changed the nature of EJB from the point of view of the developer. An EJB is a fine-grained object — nothing more complex than an annotated JavaBean. Seam even encourages you to use session beans as JSF action listeners!

On the other hand, if you prefer not to adopt EJB 3.0 at this time, you don't have to. Virtually any Java class may be a Seam component, and Seam provides all the functionality that you expect from a "lightweight" container, and more, for any component, EJB or otherwise.

Integrated with Java EE6

While Seam 2.2 was targeted Java EE 5 mainly, you can use some Java EE 6 technologies also on Seam 2.3.x.

Seam 2 and some of its extensions/implementations were added into Java EE 6 as CDI technology. So this should be a current focus of majority users. But for previous Seam 2.2 users who doesn't want or can't use pure Java EE 6, we bring some new features from the Java EE 6 set like JSF 2, JPA 2 and Bean Validation integrations into Seam 2.3.x.

Note

Be warned - Seam 2.3 should work only on Java EE 6 certified server like JBoss AS 7 is.

Integrated AJAX

Seam supports the best open source JSF-based AJAX solutions: RichFaces and ICEfaces. These solutions let you add AJAX capability to your user interface without the need to write any JavaScript code.

Alternatively, Seam provides a built-in JavaScript remoting layer that lets you call components asynchronously from client-side JavaScript without the need for an intermediate action layer. You can even subscribe to server-side JMS topics and receive messages via AJAX push.

Neither of these approaches would work well, were it not for Seam's built-in concurrency and state management, which ensures that many concurrent fine-grained, asynchronous AJAX requests are handled safely and efficiently on the server side.

Business process as a first class construct

Optionally, Seam provides transparent business process management via jBPM. You won't believe how easy it is to implement complex workflows, collaboration and task management using jBPM and Seam.

Seam even allows you to define presentation tier pageflow using the same language (jPDL) that jBPM uses for business process definition.

JSF provides an incredibly rich event model for the presentation tier. Seam enhances this model by exposing jBPM's business process related events via exactly the same event handling mechanism, providing a uniform event model for Seam's uniform component model.

Declarative state management

We're all used to the concept of declarative transaction management and declarative security from the early days of EJB. EJB 3.0 even introduces declarative persistence context management. These are three examples of a broader problem of managing state that is associated with a particular *context*, while ensuring that all needed cleanup occurs when the context ends. Seam takes the concept of declarative state management much further and applies it to *application state*. Traditionally, Java EE applications implement state management manually, by getting and setting servlet session and request attributes. This approach to state management is the source of many bugs and memory leaks when applications fail to clean up session attributes, or when session data associated with different

workflows collides in a multi-window application. Seam has the potential to almost entirely eliminate this class of bugs.

Declarative application state management is made possible by the richness of the *context model* defined by Seam. Seam extends the context model defined by the servlet spec — request, session, application — with two new contexts — conversation and business process — that are more meaningful from the point of view of the business logic.

You'll be amazed at how many things become easier once you start using conversations. Have you ever suffered pain dealing with lazy association fetching in an ORM solution like Hibernate or JPA? Seam's conversation-scoped persistence contexts mean you'll rarely have to see a `LazyInitializationException`. Have you ever had problems with the refresh button? The back button? With duplicate form submission? With propagating messages across a post-then-redirect? Seam's conversation management solves these problems without you even needing to really think about them. They're all symptoms of the broken state management architecture that has been prevalent since the earliest days of the web.

Bijection

The notion of *Inversion of Control* or *dependency injection* exists in both JSF and EJB3, as well as in numerous so-called "lightweight containers". Most of these containers emphasize injection of components that implement *stateless services*. Even when injection of stateful components is supported (such as in JSF), it is virtually useless for handling application state because the scope of the stateful component cannot be defined with sufficient flexibility, and because components belonging to wider scopes may not be injected into components belonging to narrower scopes.

Bijection differs from IoC in that it is *dynamic*, *contextual*, and *bidirectional*. You can think of it as a mechanism for aliasing contextual variables (names in the various contexts bound to the current thread) to attributes of the component. Bijection allows auto-assembly of stateful components by the container. It even allows a component to safely and easily manipulate the value of a context variable, just by assigning it to an attribute of the component.

Workspace management and multi-window browsing

Seam applications let the user freely switch between multiple browser tabs, each associated with a different, safely isolated, conversation. Applications may even take advantage of *workspace management*, allowing the user to switch between conversations (workspaces) in a single browser tab. Seam provides not only correct multi-window operation, but also multi-window-like operation in a single window!

Prefer annotations to XML

Traditionally, the Java community has been in a state of deep confusion about precisely what kinds of meta-information counts as configuration. Java EE and popular "lightweight" containers have provided XML-based deployment descriptors both for things which are truly configurable between different deployments of the system, and for any other kinds or declaration which can not easily be expressed in Java. Java 5 annotations changed all this.

EJB 3.0 embraces annotations and "configuration by exception" as the easiest way to provide information to the container in a declarative form. Unfortunately, JSF is still heavily dependent

on verbose XML configuration files. Seam extends the annotations provided by EJB 3.0 with a set of annotations for declarative state management and declarative context demarcation. This lets you eliminate the noisy JSF managed bean declarations and reduce the required XML to just that information which truly belongs in XML (the JSF navigation rules).

Integration testing is easy

Seam components, being plain Java classes, are by nature unit testable. But for complex applications, unit testing alone is insufficient. Integration testing has traditionally been a messy and difficult task for Java web applications. Therefore, Seam provides for testability of Seam applications as a core feature of the framework. You can easily write JUnit or TestNG tests that reproduce a whole interaction with a user, exercising all components of the system apart from the view. You can run these tests directly inside your IDE, where Seam will automatically deploy EJB components using Arquillian.

The specs ain't perfect

We think the latest incarnation of Java EE is great. But we know it's never going to be perfect. Where there are holes in the specifications (for example, limitations in the JSF lifecycle for GET requests), Seam fixes them. And the authors of Seam are working with the JCP expert groups to make sure those fixes make their way back into the next revision of the standards.

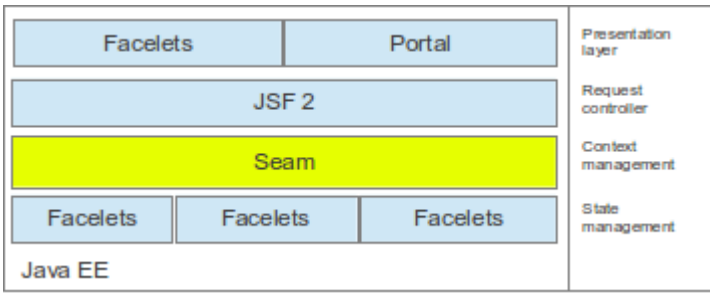
There's more to a web application than serving HTML pages

Today's web frameworks think too small. They let you get user input off a form and into your Java objects. And then they leave you hanging. A truly complete web application framework should address problems like persistence, concurrency, asynchronicity, state management, security, email, messaging, PDF and chart generation, workflow, wikitext rendering, webservices, caching and more. Once you scratch the surface of Seam, you'll be amazed at how many problems become simpler...

Seam integrates JPA and Hibernate for persistence, the EJB Timer Service and Quartz for lightweight asychronicity, jBPM for workflow, JBoss Rules for business rules, Meldware Mail for email, Hibernate Search and Lucene for full text search, JMS for messaging and JBoss Cache for page fragment caching. Seam layers an innovative rule-based security framework over JAAS and JBoss Rules. There's even JSF tag libraries for rendering PDF, outgoing email, charts and wikitext. Seam components may be called synchronously as a Web Service, asynchronously from client-side JavaScript or Google Web Toolkit or, of course, directly from JSF.

Get started now!

Seam should work in any Java EE application server, and even works in Tomcat. If your environment supports EJB 3.0 or 3.1, great! If it doesn't, no problem, you can use Seam's built-in transaction management with JPA or Hibernate for persistence.



It turns out that the combination of Seam, JSF and EJB is *the* simplest way to write a complex web application in Java. You won't believe how little code is required!

1. Contribute to Seam

Visit *SeamFramework.org* [http://www.seamframework.org/Community/Contribute] to find out how to contribute to Seam!

Seam Tutorial

1.1. Using the Seam examples

Seam provides a number of example applications demonstrating how to use the various features of Seam. 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 registration example, which will be the first example we look at, is in the `examples/registration` directory.

Each example has the very similar directory structure which is based on *Maven project structure defaults* [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]:

- The `<example>-ear` directory contains enterprise application submodule files such as aggregator for web application files, EJB project.
- The `<example>-web` directory contains web application submodule view-related files such as web page templates, images and stylesheets.
- The `<example>-ejb` directory contains Enterprise Java Beans components.
- The `<example>-tests` directory contains integration and functional tests.

- The `<example>-web/src/main/webapp` directory contains view-related files such as web page templates, images and stylesheets.
- The `<example>-[ear|ejb]/src/main/resources` directory contains deployment descriptors and other configuration files.
- The `<example>-ejb/src/main/java` directory contains the application source code.

The example applications run on JBoss AS 7.1.1 with no additional configuration. The following sections will explain the procedure. Note that all the examples are built and run from the Maven `pom.xml`, so you'll need at least version 3.x of Maven installed before you get started. At the time of writing this text recent version of Maven was 3.0.4.

1.1.1. Running the examples on JBoss AS

The examples are configured for use on JBoss AS 7.1. You'll need to set `JBOSS_HOME`, in your environment, to the location of your JBoss AS installation.

Once you've set the location of JBoss AS and started the application server, you can build any example by typing `mvn install` in the example root directory. Any example is deployed by

changing directory to *-ear or *-web directory in case of existence only *-web submodule. Type in that submodule `mvn jboss-as:deploy`. Any example that is packaged as an EAR 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 ommitted. For instance, if JBoss AS is running on port 8080, the URL for the registration example is *`http://localhost:8080/seam-registration/`* [http://localhost:8080/seam-registration/], whereas the URL for the seamspace example is *`http://localhost:8080/seam-space/`* [http://localhost:8080/seam-space/].

If, on the other hand, the example gets packaged as a WAR, then it deploys to a URL like `/jboss-seam-example`. Several of the examples can only be deployed as a WAR. Those examples are groovybooking, hibernate, jpa, and spring.

1.1.2. Running the example tests

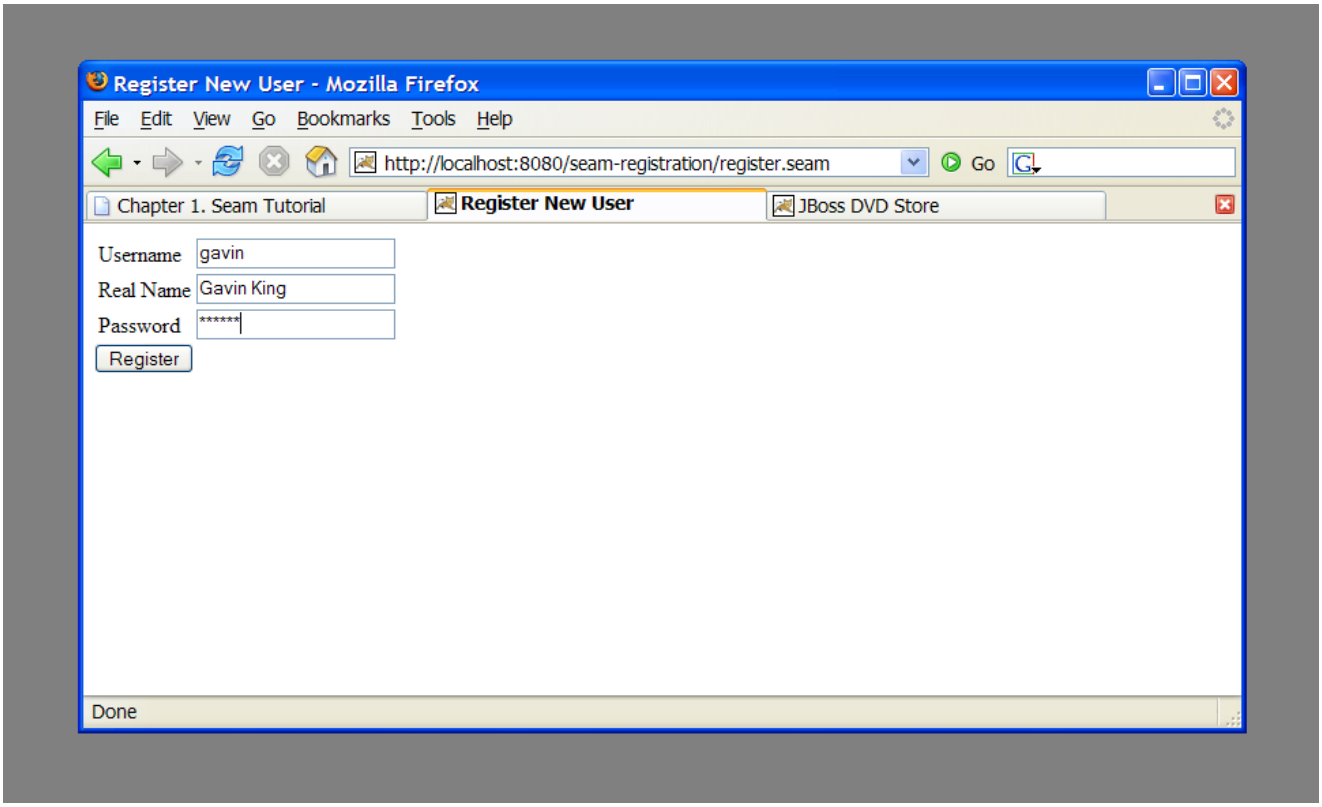
Most of the examples come with a suite of Arquillian JUnit integration tests. The easiest way to run the tests is to run `mvn verify -Darquillian=jbossas-managed-7`. It is also possible to run the tests inside your IDE using the JUnit plugin. Consult the readme.txt in the examples directory of the Seam 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 his username, real name and password in the database. The example isn't intended to show off all of the cool functionality of Seam. However, it demonstrates the use of an EJB3 session bean as a JSF action listener, and basic configuration of Seam.

We'll go slowly, since we realize you might not yet be familiar with EJB 3.0.

The start page displays a very basic form with three input fields. Try filling them in and then submitting the form. This 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. Let's take a look at the code, starting from the "bottom".

1.2.1.1. The entity bean: `User.java`

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

Example 1.1. User.java

```
@Entity                                                    (1)
@Name("user")                                              (2)
@Scope(SESSION)                                            (3)
@Table(name="users")                                       (4)
public class User implements Serializable
{
  private static final long serialVersionUID = 1881413500711441951L;

  private String username;                                 (5)
  private String password;
```

```
  private String name;

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

  public User() {}                                                    (6)

  @NotNull @Size(min=5, max=15)                                       (7)
  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;
  }

  @Id @NotNull @Size(min=5, max=15)                                   (8)
  public String getUsername()
  {
    return username;
  }

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

```
}
```

1. The JPA 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 JPA 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 both required by both the JPA specification and by Seam.
7. The `@NotNull` and `@Size` annotations are part of the Bean Validation annotations specification (JSR-303). Seam integrates Bean Validation through Hibernate Validator, which is the reference implementation, and lets you use it for data validation (even if you are not using Hibernate for persistence).
8. The JPA 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.

We'll see below that the properties of our `User` class are bound directly to JSF components and are populated by JSF during the update model values phase. We don't need any tedious glue code to copy data back and forth between the JSF pages and the entity bean domain model.

However, entity beans shouldn't do transaction management or database access. So we can't use this component as a JSF action listener. For that we need a session bean.

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

Most Seam application use session beans as JSF action listeners (you can use JavaBeans instead if you like).

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

This is the only really interesting code in the example!

Example 1.2. RegisterAction.java

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

  @PersistenceContext
  private EntityManager em;                                   (3)

  @Logger
  private Log log;                                            (4)

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

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

}
```

(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 JPA entity manager.

4. The Seam `@Logger` annotation is used to inject the component's `Log` instance.
5. The action listener method uses the standard JPA `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 JPQL. Under the covers, this results in an ordinary JPA `setParameter()` call on the standard JPA `Query` object. Nice, huh?
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 application 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 if not explicitly specified. For stateless session beans, the default scope is the stateless context, which is the only sensible value.

Our session bean action listener performs the business and persistence logic for our mini-application. In more complex applications, we might need require a separate service layer. This is easy to achieve with Seam, but it's overkill for most web applications. Seam does not force you into any particular strategy for application layering, allowing your application to be as simple, or as complex, as you want.

Note that in this simple application, we've actually made it far more complex than it needs to be. If we had used the Seam application framework controllers, we would have eliminated all of our application code. However, then we wouldn't have had much of an application to explain.

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

Naturally, our session bean needs a local interface.

Example 1.3. Register.java

```
@Local
public interface Register
```

```
{
   public String register();
}
```

That's the end of the Java code. Now we'll look at the view.

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

The view pages for a Seam application could be implemented using any technology that supports JSF. In this example we use Facelets, because we think it's better than JSF.

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.org/schema/seam/taglib"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">

  <h:head>
     <title>Register New User</title>
  </h:head>
  <h:body>
 <h:head>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>
  </h:body>

</html>
```

The only thing here that is specific to Seam is the `<s:validateAll>` tag. This JSF component tells JSF to validate all the contained input fields against the Bean Validation 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">

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

</html>
```

This is a simple Facelets page using some inline EL. There's nothing specific to Seam here.

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

Since this is the first Seam app we've seen, we'll take a look at the deployment descriptors. Before we get into them, it is worth noting that Seam strongly values minimal configuration. These configuration files will be created for you when you create a Seam application. You'll never need to touch most of these files. We're presenting them now only to help you understand what all the pieces in the example are doing.

If you've used many Java frameworks before, you'll be used to having to declare all your component classes in some kind of XML file that gradually grows more and more unmanageable as your project matures. You'll be relieved to know that Seam does not require that application components be accompanied by XML. Most Seam applications require a very small amount of XML that does not grow very much as the project gets bigger.

Nevertheless, it is often useful to be able to provide for *some* external configuration of *some* components (particularly the components built in to Seam). You have a couple of options here, but the most flexible option is to provide this configuration in a file called `components.xml`, located in the `WEB-INF` directory. We'll use the `components.xml` file to tell Seam how to find our EJB components in JNDI:

Example 1.6. components.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.org/schema/seam/components"
   xmlns:core="http://jboss.org/schema/seam/core"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://jboss.org/schema/seam/core
      http://jboss.org/schema/seam/core-2.3.xsd
      http://jboss.org/schema/seam/components
      http://jboss.org/schema/seam/components-2.3.xsd">

   <core:init jndi-pattern="${jndiPattern}"/>

</components>
```

This code configures a property named `jndiPattern` of a built-in Seam component named `org.jboss.seam.core.init`. The funny `@` symbols are there because our Maven build puts the correct JNDI pattern in when we deploy the application, which it reads from the components.properties file. You learn more about how this process works in *Section 6.2, "Configuring components via components.xml"*.

Note

Eclipse M2e Web tools plugin can't use the `@` for token property filtering. Fortunately there works the other way which is in Maven filtering defined - `${property}`.

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

The presentation layer for our mini-application will be deployed in a WAR. So we'll need a web deployment descriptor.

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_3_0.xsd"
   version="3.0">
```

```
    <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>
```

This `web.xml` file configures Seam and JSF. The configuration you see here is pretty much identical in all Seam applications.

1.2.1.7. The JSF configuration: `faces-config.xml`

Most Seam applications use JSF views as the presentation layer. So usually we'll need `faces-config.xml`. In our case, we are going to use Facelets for defining 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_2_1.xsd"
    version="2.1">
```

```
</faces-config>
```

Note that we don't need any JSF managed bean declarations and neither FaceletViewHandler definition as Facelets are default view technology in JSF 2! Our managed beans are annotated Seam components. So basically we don't need `faces-config.xml` at all, but here is the `faces-config.xml` as the template for advanced JSF configurations.

In fact, once you have all the basic descriptors set up, the *only* XML you need to write as you add new functionality to a Seam application is orchestration: navigation rules or jBPM process definitions. Seam's stand is that *process flow* and *configuration data* are the only things that truly belong in XML.

In this simple example, we don't even need a navigation rule, since we decided to embed the view id 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.

```
<?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 JPA persistence deployment descriptor: `persistence.xml`

The `persistence.xml` file tells the JPA persistence provider where to find the datasource, and contains some vendor-specific settings. In this case, enables automatic schema export at startup time.

```
<?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_2_0.xsd"
   version="2.0">

   <persistence-unit name="userDatabase">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</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 need a deployment descriptor there, too.

Note

This file can be generated by Maven EAR plugin and registration application has got this set up in registration-ear/pom.xml.

Just for clarity, the following is the result of that generation:

Example 1.9. 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_6.xsd"
   version="6">
   <display-name>registration-ear</display-name>
   <module>
      <web>
         <web-uri>registration-web.war</web-uri>
         <context-root>/seam-registration</context-root>
      </web>
   </module>
   <module>
      <ejb>registration-ejb.jar</ejb>
   </module>
   <module>
      <ejb>jboss-seam.jar</ejb>
   </module>
</application>
```

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

We've now seen *all* the files in the entire application!

1.2.2. How it works

When the form is submitted, JSF asks Seam to resolve the variable named `user`. Since there is no value already 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 Bean 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.

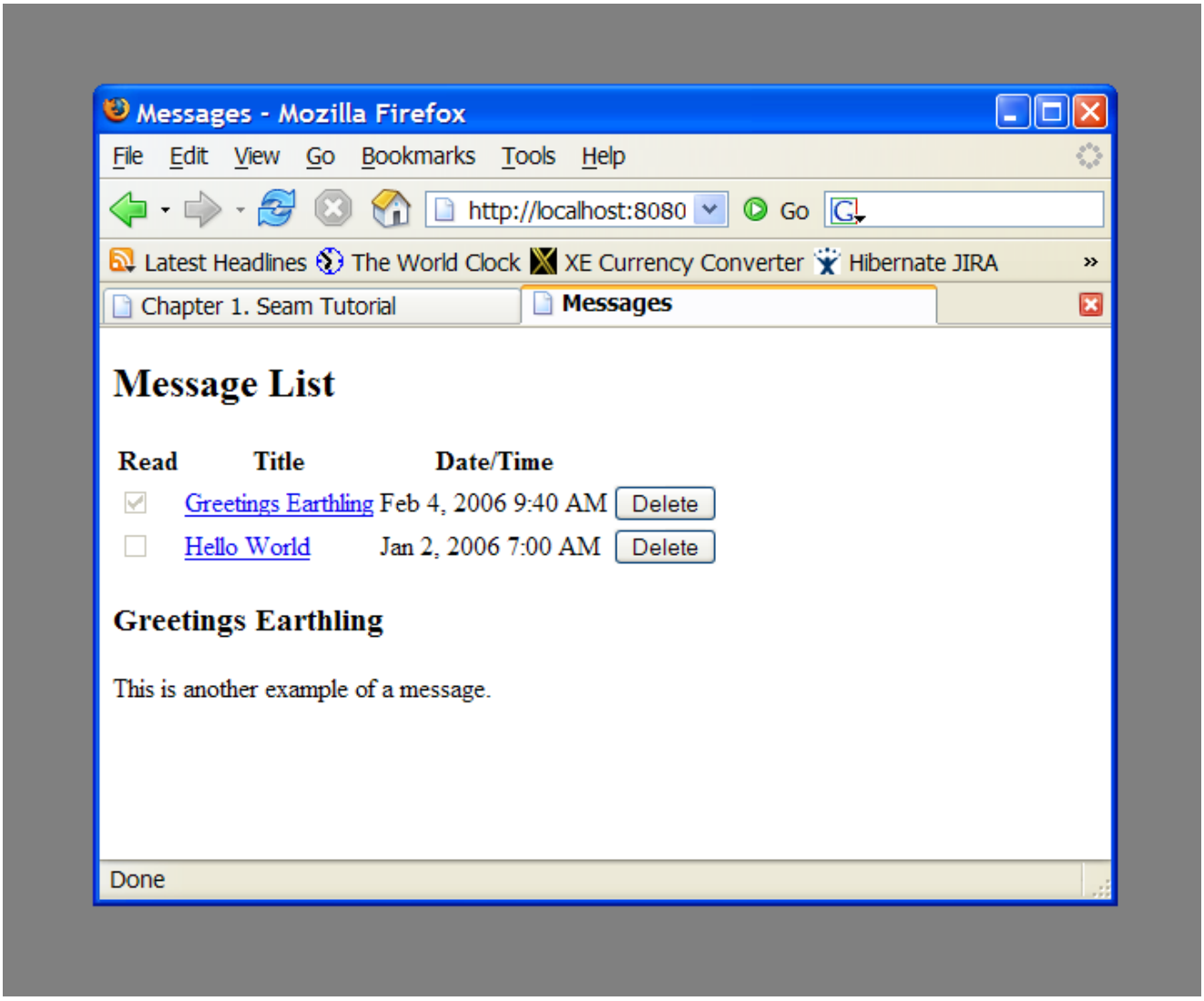
But Seam is not done yet. Seam intercepts the method call and injects the `User` entity from the Seam session context, before allowing the invocation to continue.

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 such an important part of any online application that Seam provides special functionality on top of JSF to make it easier to query data using JPQL 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 JSF.

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 1.10. 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 @Size(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`

Just like in the previous example, we have a session bean, `MessageManagerBean`, which defines the action listener methods for the two buttons on our form. One of the buttons selects a message from the list, and displays that message. The other button deletes a message. So far, this is not so different to the previous example.

But `MessageManagerBean` is also responsible for fetching the list of messages the first time we navigate to the message list page. There are various ways the user could navigate to the page, and not all of them are preceded by a JSF action — the user might have bookmarked the page, for example. So the job of fetching the message list takes 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 1.11. MessageManagerBean.java

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean implements Serializable, MessageManager
{
   @DataModel
   private List<Message> messageList;                                    (1)

   @DataModelSelection
   @Out(required=false)                                                  (2)
   private Message message;                                              (3)

   @PersistenceContext(type=EXTENDED)
   private EntityManager em;                                             (4)

   @Factory("messageList")
   public void findMessages()                                            (5)
   {
     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
   public void destroy() {}                                              (8)

}
```

1. The `@DataModel` annotation exposes an attribute 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 the subsequently *outjected* to the event context variable named `message`.
4. This stateful bean has an JPA *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 JSF 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* have a method with no parameters marked `@Remove` that Seam uses to remove the stateful bean when the Seam context ends, and clean up any server-side state.

Note that this is a session-scoped Seam component. It is associated with the user login session, and all requests from a login session share the same instance of the component. (In Seam applications, we usually use session-scoped components sparingly.)

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

All session beans have a business interface, of course.

Example 1.12. MessageManager.java

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

From now on, we won't show local interfaces in our code examples.

Let's skip over `components.xml`, `persistence.xml`, `web.xml`, `ejb-jar.xml`, `faces-config.xml` and `application.xml` since they are much the same as the previous example, and go straight to the JSF.

1.3.1.4. The view: `messages.xhtml`

The JSF page is a straightforward use of the JSF `<h:dataTable>` component. Again, nothing specific to Seam.

Example 1.13. messages.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.org/schema/seam/taglib"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core">
 <h:head>
  <title>Messages</title>
 </h:head>
 <h:body>
  <f:view>
    <h2>Message List</h2>
                    <h:outputText id="noMessages" value="No messages to display"
rendered="#{messageList.rowCount==0}"/>
                        <h:dataTable id="messages" var="msg" value="#{messageList}"
rendered="#{messageList.rowCount>0}">
      <h:column>
        <f:facet name="header">
          <h:outputText value="Read"/>
        </f:facet>
        <h:selectBooleanCheckbox id="read" value="#{msg.read}" disabled="true"/>
      </h:column>
      <h:column>
        <f:facet name="header">
          <h:outputText value="Title"/>
        </f:facet>
        <s:link id="link" value="#{msg.title}" action="#{messageManager.select}"/>
      </h:column>
      <h:column>
        <f:facet name="header">
          <h:outputText value="Date/Time"/>
        </f:facet>
```

```
        <h:outputText id="date" value="#{msg.datetime}">
          <f:convertDateTime type="both" dateStyle="medium" timeStyle="short"/>
        </h:outputText>
      </h:column>
      <h:column>
        <s:button id="delete" value="Delete" action="#{messageManager.delete}"/>
      </h:column>
    </h:dataTable>
    <h3><h:outputText id="title" value="#{message.title}"/></h3>
    <div><h:outputText id="text" value="#{message.text}"/></div>
  </f:view>
 </h:body>
</html>
```

1.3.2. How it works

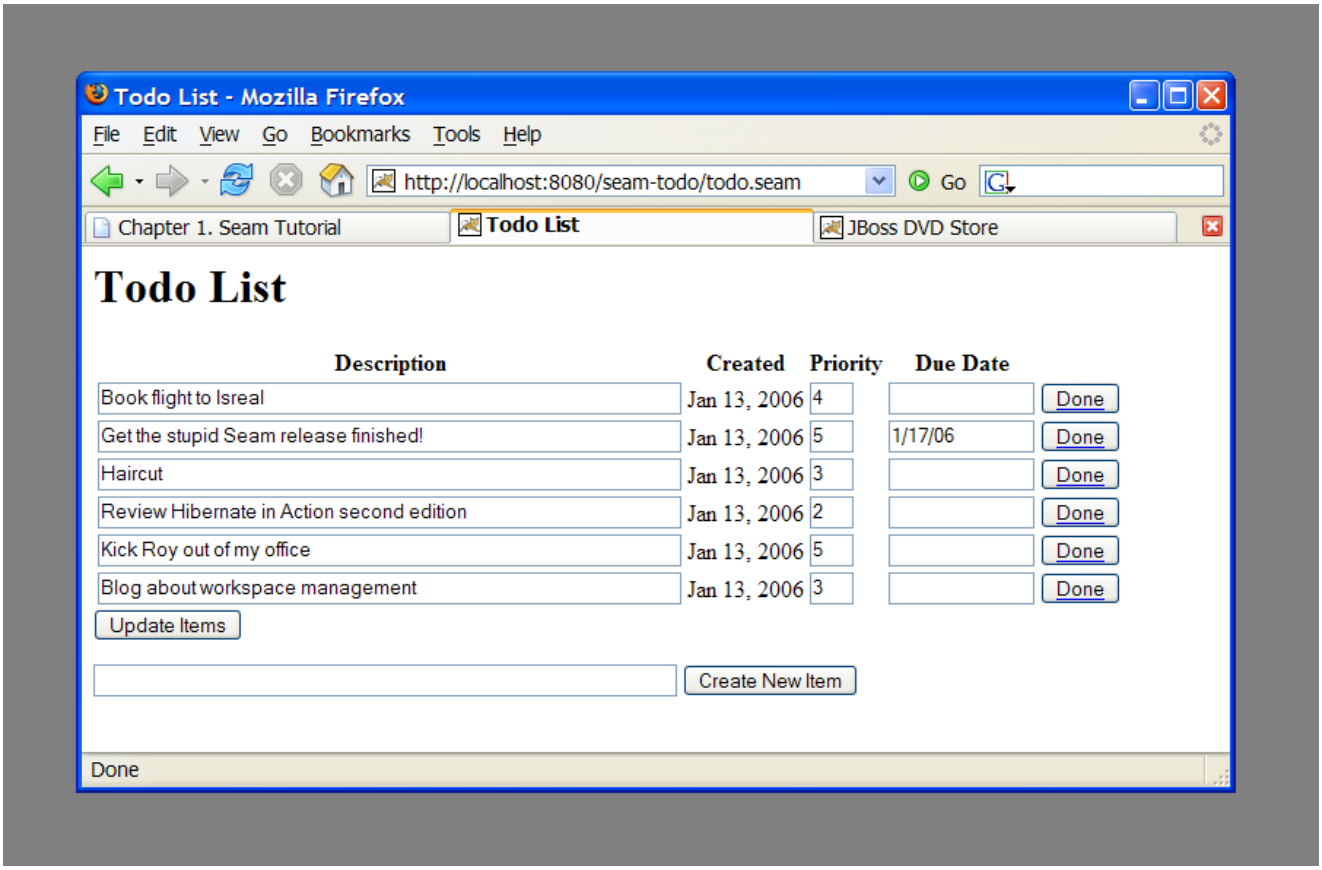
The first time we navigate to the `messages.xhtml` page, the page will try to resolve the `messageList` context variable. Since this context variable is not initialized, Seam will call the factory method `findMessages()`, which performs a query against the database and results in a `DataModel` being outjected. This `DataModel` provides the row data needed for rendering 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 context variable named `message`. 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.

If 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 `messageManager` component. The action listener fires, removing the selected `Message` from the list, and also calling `remove()` on the `EntityManager`. At the end of the call, Seam refreshes the `messageList` context variable and clears the context variable named `message`. 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. To get a small taste of how jBPM integrates with Seam, we'll show you a simple "todo list" application. Since managing lists of tasks is such core functionality for jBPM, there is hardly any Java code at all in this example.



1.4.1. Understanding the code

The center of this example is the jBPM process definition. There are also two JSFs and two trivial JavaBeans (There was no reason to use session beans, since they do not access the database, or have any other transactional behavior). Let's start with the process definition:

Example 1.14. 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.

This document defines our *business process* as a graph of nodes. This is the most trivial 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.xhtml`. Its job is just to initialize the jBPM actor id using the `actor` component. In a real application, it would also need to authenticate the user.

Example 1.15. 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.xhtml";
  }
}
```

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

The JSF itself is trivial:

Example 1.16. login.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.org/schema/seam/taglib"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Login</title>
  </h:head>
  <h:body>
    <h1>Login</h1>
    <f:view>
      <h:form id="login">
        <div>
          <h:inputText id="username" value="#{login.user}"/>
          <h:commandButton id="submit" value="Login" action="#{login.login}"/>
        </div>
      </h:form>
    </f:view>
  </h:body>
</html>
```

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

Example 1.17. TodoList.java

```
@Name("todoList")
```

```
public class TodoList
{
  private String description;

  public String getDescription()                    (1)
  {
    return description;
  }

  public void setDescription(String description)
  {
    this.description = description;
  }
                                                    (2)
  @CreateProcess(definition="todo")
  public void createTodo() {}
                                                    (3)
  @StartTask @EndTask
  public void done() {}

}
```

1. The description property accepts user input from the JSF 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 there is usually work to be done using the application in order to complete the task.

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

Example 1.18. todo.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.org/schema/seam/taglib"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
```

```
<title>Todo List</title>
</head>
<body>
<h1>Todo List</h1>
<f:view>
  <h:form id="list">
    <div>
          <h:outputText id="noItems" value="There are no todo items." rendered="#{empty
taskInstancePriorityList}"/>
        <h:dataTable id="items" value="#{taskInstancePriorityList}" var="task" rendered="#{not
empty taskInstancePriorityList}">
        <h:column>
          <f:facet name="header">
            <h:outputText value="Description"/>
          </f:facet>
          <h:inputText id="description" value="#{task.description}" style="width: 400"/>
        </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 id="priority" value="#{task.priority}" style="width: 30"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Due Date"/>
          </f:facet>
          <h:inputText id="dueDate" value="#{task.dueDate}" style="width: 100">
            <f:convertDateTime type="date" dateStyle="short"/>
          </h:inputText>
        </h:column>
        <h:column>
          <s:button id="done" action="#{todoList.done}" taskInstance="#{task}" value="Done"/>
        </h:column>
      </h:dataTable>
    </div>
```

```
    <div>
    <h:messages/>
    </div>
    <div>
                <h:commandButton id="update" value="Update Items" rendered="#{not empty
 taskInstanceList}"/>
    </div>
  </h:form>
  <h:form id="new">
    <div>
      <h:inputText id="description" value="#{todoList.description}" style="width: 400"/>
      <h:commandButton id="create" value="Create New Item" action="#{todoList.createTodo}"/>
    </div>
  </h:form>
</f:view>
</body>
</html>
```

Let's take this one piece at a time.

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

Example 1.19. todo.xhtml

```
<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 simply displays the interesting properties of each task in the list. For the description, priority and due date, we use input controls, to allow the user to update these values.

```
<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 to a date (no time). Thus, the converter is not 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 is using a Seam `<s:button>` JSF control from the `seam-ui.jar` package. This button is used to update 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 is used to create new items, by calling 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.xhtml uses the `taskInstanceList` component to display a table of outstanding todo items for a the current user. Initially there are none. It also presents a form to enter a new entry. When the user types the todo item and hits the "Create New Item" button, `#{todoList.createTodo}` is called. This starts the todo process, as defined in `todo.jpdl.xml`.

The process instance is created, starting in the start state and immediately transition to the `todo` state, where a new task is created. The task description is set based on the user's input, which was saved to `#{todoList.description}`. Then, the task is assigned to the current user, which was stored in the seam actor component. Note that in this example, the process has no extra process state. All the state in this example is stored in the task definition. The process and task information is stored in the database at the end of the request.

When `todo.xhtml` is redisplayed, `taskInstanceList` now finds the task that was just created. The task is shown 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 back to the database.

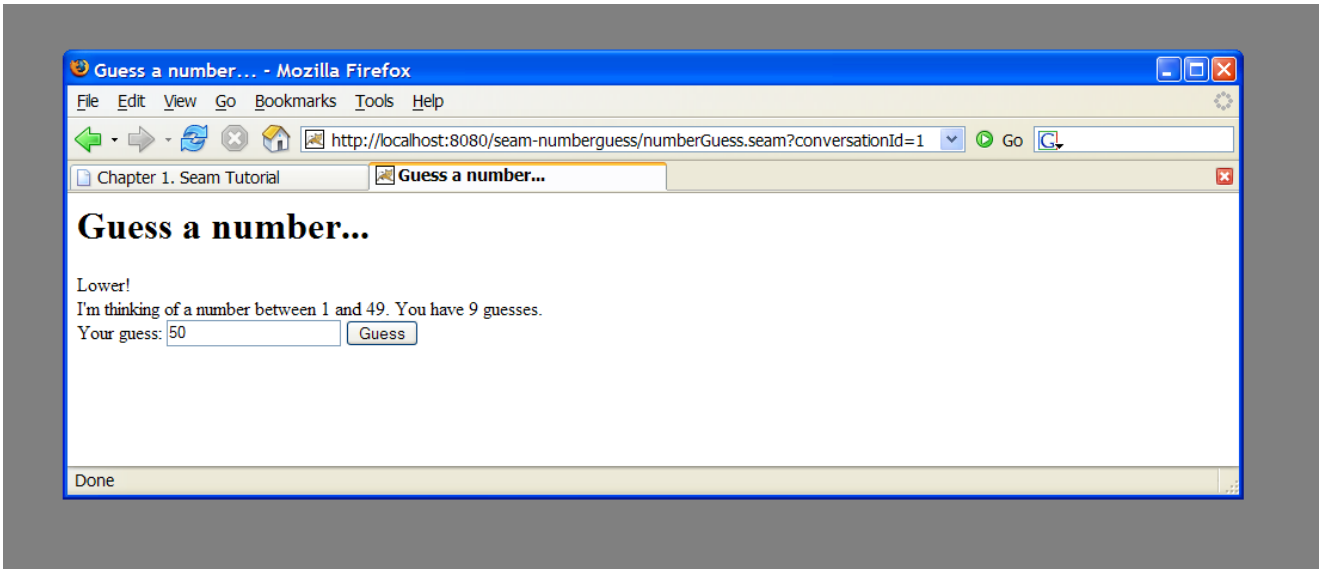
Each todo item also has "Done" button, which calls `#{todoList.done}`. The `todoList` component knows which task the button is for because each s:button specificies `taskInstance="#{task}"`, referring to the task for that particular line of the table. The `@StartTast` and `@EndTask` annotations cause seam to make the task active and immediately complete the task. The original process then transitions into the `done` state, according to the process definition, where it ends. The state of the task and process are both updated in the database.

When `todo.xhtml` is displayed again, the now-completed task is no longer displayed in the `taskInstanceList`, since that component only display active tasks for the user.

1.5. Seam pageflow: the numberguess example

For Seam applications with relatively freeform (ad hoc) navigation, JSF/Seam navigation rules are a perfectly good way to define the page flow. For applications with a more constrained style of navigation, especially for user interfaces which are more stateful, navigation rules make it difficult to really understand the flow of the system. To understand the flow, you need to piece it together from the view pages, the actions and the navigation rules.

Seam allows you to use a jPDL process definition to define pageflow. The simple number guessing example shows how this is done.



1.5.1. Understanding the code

The example is implemented using one JavaBean, three JSF pages and a jPDL pageflow definition. Let's begin with the pageflow:

Example 1.20. pageflow.jpdl.xml

```
<pageflow-definition
      xmlns="http://jboss.org/schema/seam/pageflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.org/schema/seam/pageflow
                  http://jboss.org/schema/seam/pageflow-2.3.xsd"
      name="numberGuess">

  <start-page name="displayGuess" view-id="/numberGuess.xhtml">          (1)
    <redirect/>
    <transition name="guess" to="evaluateGuess">                        (2)
      <action expression="#{numberGuess.guess}"/>                       (3)
```

```
    </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.xhtml">
    <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.xhtml">
    <redirect/>
    <end-conversation/>
  </page>

  <page name="lose" view-id="/lose.xhtml">
    <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.

Now that we have seen the pageflow, it is very, very easy to understand the rest of the application!

Here is the main page of the application, `numberGuess.xhtml`:

Example 1.21. numberGuess.xhtml

```
<!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:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:s="http://jboss.org/schema/seam/taglib">
  <h:head>
    <title>Guess a number...</title>
    <link href="niceforms.css" rel="stylesheet" type="text/css" />
    <script language="javascript" type="text/javascript" src="niceforms.js"><!-- --></script>
  </h:head>
  <h:body>
    <h1>Guess a number...</h1>
      <h:form id="NumberGuessMain" styleClass="niceform">

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

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

```
    </div>

    <div>
    Your guess:
        <h:inputText id="inputGuess" value="#{numberGuess.currentGuess}" required="true"
size="3"
          rendered="#{(numberGuess.biggest-numberGuess.smallest) gt 20}">
      <f:validateLongRange maximum="#{numberGuess.biggest}"
                   minimum="#{numberGuess.smallest}"/>
    </h:inputText>
          <h:selectOneMenu id="selectGuessMenu" value="#{numberGuess.currentGuess}"
required="true"
                       rendered="#{(numberGuess.biggest-numberGuess.smallest) le 20 and
(numberGuess.biggest-numberGuess.smallest) gt 4}">
        <s:selectItems id="PossibilitiesMenuItems" value="#{numberGuess.possibilities}" var="i"
label="#{i}"/>
    </h:selectOneMenu>
          <h:selectOneRadio id="selectGuessRadio" value="#{numberGuess.currentGuess}"
required="true"
                rendered="#{(numberGuess.biggest-numberGuess.smallest) le 4}">
        <s:selectItems id="PossibilitiesRadioItems" value="#{numberGuess.possibilities}" var="i"
label="#{i}"/>
    </h:selectOneRadio>
    <h:commandButton id="GuessButton" value="Guess" action="guess"/>
    <s:button id="CheatButton" value="Cheat" action="cheat"/>
    <s:button id="GiveUpButton" value="Give up" action="giveup"/>
    </div>

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

   </h:form>
 </h:body>
</html>
```

Notice how the command button names the `guess` transition instead of calling an action directly.

The `win.xhtml` page is predictable:

Example 1.22. win.xhtml

```
<!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:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:s="http://jboss.org/schema/seam/taglib">
  <h:head>
    <title>You won!</title>
    <link href="niceforms.css" rel="stylesheet" type="text/css" />
  </h:head>
  <h:body>
    <h1>You won!</h1>
                                Yes, the answer was <h:outputText id="CurrentGuess"
 value="#{numberGuess.currentGuess}" />.
     It took you <h:outputText id="GuessCount" value="#{numberGuess.guessCount}" /> guesses.
           <h:outputText id="CheatedMessage" value="But you cheated, so it doesn't count!"
 rendered="#{numberGuess.cheat}"/>
      Would you like to <a href="numberGuess.seam">play again</a>?
  </h:body>
</html>
```

The `lose.xhtml` looks roughly the same, so we'll skip over it.

Finally, we'll look at the actual application code:

Example 1.23. 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                                                    (1)
   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 JSF 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* (much more about that later), and specifies the pageflow definition to use for the conversation's page flow.

Example 1.24. pages.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.org/schema/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.org/schema/seam/pages http://jboss.org/schema/seam/
pages-2.3.xsd">

  <page view-id="/numberGuess.xhtml">
    <begin-conversation join="true" pageflow="numberGuess"/>
  </page>

</pages>
```

As you can see, this Seam component is pure business logic! It doesn't need to know anything at all about the user interaction flow. This makes the component potentially more reuseable.

1.5.2. How it works

We'll step through basic flow of the application. The game starts with the `numberGuess.xhtml` view. When the page is first displayed, the `pages.xml` configuration causes conversation to begin and associates the `numberGuess` pageflow with that conversation. The pageflow starts with a `start-page` tag, which is a wait state, so the `numberGuess.xhtml` is rendered.

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

The "Guess" button triggers the `guess` action. Seam defers to the pageflow to handle the action, which says that the pageflow should transition to the `evaluateGuess` state, first invoking `#{numberGuess.guess}`, which updates the guess count and highest/lowest suggestions in the `numberGuess` component.

The `evaluateGuess` state checks the value of `#{numberGuess.correctGuess}` and transitions either to the `win` or `evaluatingRemainingGuesses` state. We'll assume the number was incorrect, in which case the pageflow transitions to `evaluatingRemainingGuesses`. That is also a decision state, which tests the `#{numberGuess.lastGuess}` state to determine whether or not the user has more guesses. If there are more guesses (`lastGuess` is `false`), we transition back to the original `displayGuess` state. Finally we've reached a page state, so the associated page `/numberGuess.xhtml` is displayed. Since the page has a redirect element, Seam sends a redirect to the user's browser, starting the process over.

We won't follow the state any more except to note that if on a future request either the `win` or the `lose` transition were taken, the user would be taken to either the `/win.xhtml` or `/lose.xhtml`. Both states specify that Seam should end the conversation, tossing away all the game state and pageflow state, before redirecting the user to the final page.

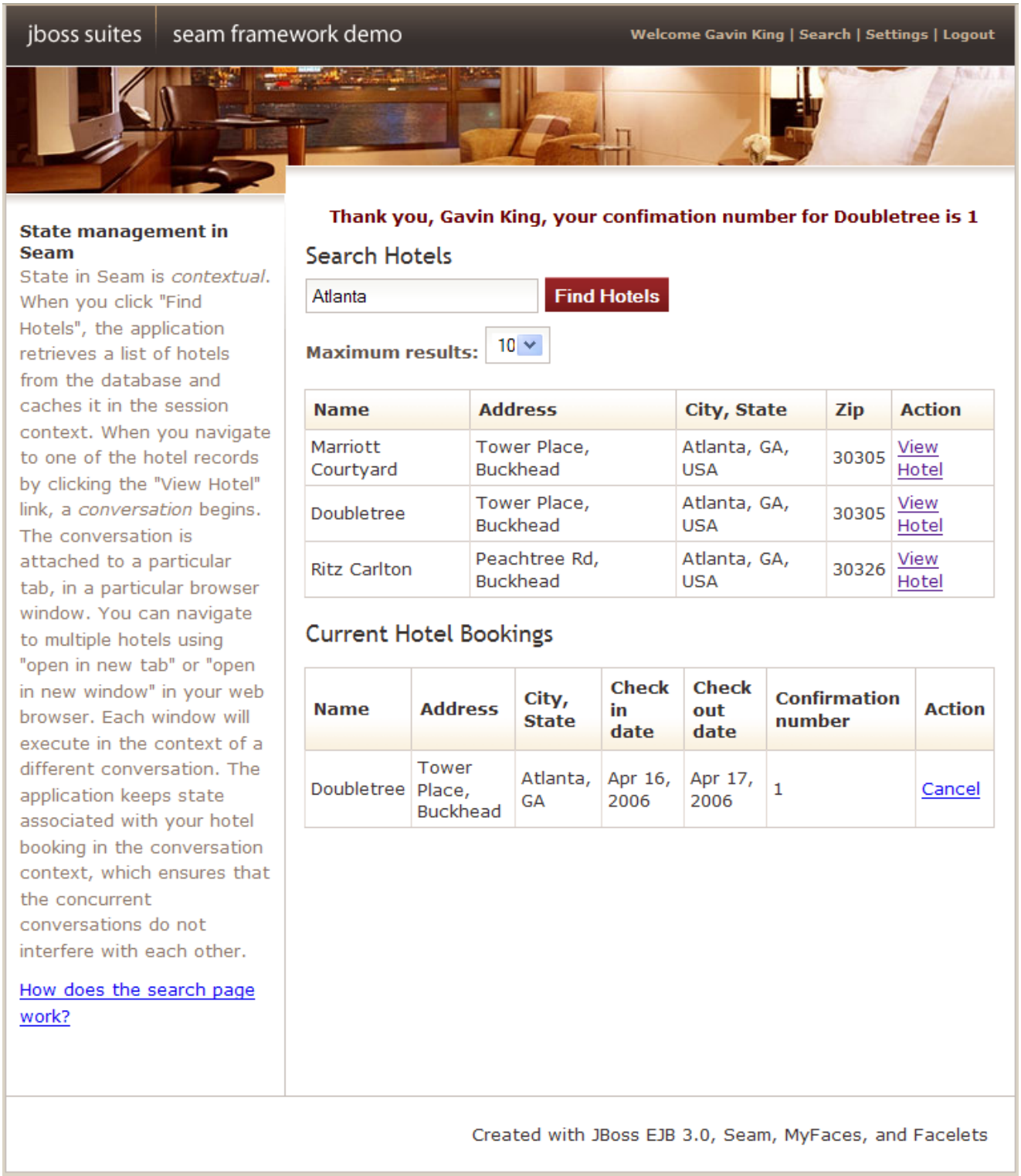
The numberguess example also contains Giveup and Cheat buttons. You should be able to trace the pageflow state for both actions relatively easily. Pay particular attention to the `cheat` transition, which loads a sub-process to handle that flow. Although it's overkill for this application, it does demonstrate how complex pageflows can be broken down into smaller parts 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 2, EJB 3.0 and Seam, together with Facelets for the view. There is also a port of this application to JSF 2, Seam, JavaBeans and Hibernate4.

One of the things you'll notice if you play with this application for long enough is that it is extremely *robust*. You can play with back buttons and browser refresh and opening multiple windows and entering nonsensical data as much as you like and you will find it very difficult to make the

application crash. You might think that we spent weeks testing and fixing bugs to achieve this. Actually, this is not the case. Seam was designed to make it very straightforward to build robust web applications and a lot of robustness that you are probably used to having to code yourself comes naturally and automatically with Seam.

As you browse the sourcecode of the example application, and learn how the application works, observe how 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 is identical to the previous one, to install and deploy this application, please refer to *Section 1.1, "Using the Seam examples"*. Once you've successfully started the application, you can access it by pointing your browser to `http://localhost:8080/seam-booking/` [http://localhost:8080/seam-booking/]

The application uses six session beans for to implement the business logic for the listed features.

- `AuthenticatorAction` provides the login 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 functionality is implemented as a *conversation*, so this is one of the most interesting 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 represent a hotel
- `Booking` is an entity bean that represents an existing booking
- `User` is an entity bean to represents a user who can make hotel bookings

1.6.3. Understanding Seam conversations

We encourage you browse the sourcecode at your pleasure. In this tutorial we'll concentrate upon one particular piece of functionality: hotel search, selection, booking and confirmation. From the point of view of the user, everything from selecting a hotel to confirming a booking is one continuous unit of work, a *conversation*. Searching, however, is *not* part of the conversation. The user can select multiple hotels from the same search results page, in different browser tabs.

Most web application architectures have no first class construct to represent a conversation. This causes enormous problems managing conversational state. Usually, Java web applications use a combination of several techniques. Some state can be transfered in the URL. What can't is either

thrown into the `HttpSession` or flushed to the database after every request, and reconstructed from the database at the beginning of each new request.

Since the database is the least scalable tier, this often results in an utterly unacceptable lack of scalability. Added latency is also a problem, due to the extra traffic to and from the database on every request. To reduce this redundant traffic, Java applications often introduce a data (second-level) cache that keeps commonly accessed data between requests. This cache is necessarily inefficient, because invalidation is based upon an LRU policy instead of being based upon when the user has finished working with the data. Furthermore, because the cache is shared between many concurrent transactions, we've introduced a whole raft of problem's associated with keeping the cached state consistent with the database.

Now consider the state held in the `HttpSession`. The HttpSession is great place for true session data, data that is common to all requests that the user has with the application. However, it's a bad place to store data related to individual series of requests. Using the session of conversational quickly breaks down when dealing with the back button and multiple windows. On top of that, without careful programming, data in the HTTP Session can grow quite large, making the HTTP session difficult to cluster. Developing mechanisms to isolate session state associated with different concurrent conversations, and incorporating failsafes to ensure that conversation state is destroyed when the user aborts one of the conversations by closing a browser window or tab is not for the faint hearted. Fortunately, with Seam, you don't have to worry about that.

Seam introduces the *conversation context* as a first class construct. You can safely keep conversational state in this context, and be assured that it will have a well-defined lifecycle. Even better, you won't need to be continually pushing data back and forth between the application server and the database, since the conversation context is a natural cache of data that the user is currently working with.

In this application, we'll use the conversation context to store stateful session beans. There is an ancient canard in the Java community that stateful session beans are a scalability killer. This may have been true in the early days of enterprise Java, but it is no longer true today. Modern application servers have extremely sophisticated mechanisms for stateful session bean state replication. JBoss AS, for example, performs fine-grained replication, replicating only those bean attribute values which actually changed. Note that all the traditional technical arguments for why stateful beans are inefficient apply equally to the `HttpSession`, so the practice of shifting state from business tier stateful session bean components to the web session to try and improve performance is unbelievably misguided. It is certainly possible to write unscalable applications using stateful session beans, by using stateful beans incorrectly, or by using them for the wrong thing. But that doesn't mean you should *never* use them. If you remain unconvinced, Seam allows the use of POJOs instead of stateful session beans. With Seam, the choice is yours.

The booking example application shows how stateful components with different scopes can collaborate together to achieve complex behaviors. The main page of the booking application allows the user to search for hotels. The search results are kept in the Seam session scope. When the user navigates to one of these hotels, a conversation begins, and a conversation scoped component calls back to the session scoped component to retrieve the selected hotel.

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

The search functionality is implemented using a session-scope stateful session bean, similar to the one we saw in the message list example.

Example 1.25. HotelSearchingAction.java

```
@Stateful                                                    (1)
@Name("hotelSearch")
@Scope(ScopeType.SESSION)
@Restrict("#{identity.loggedIn}")                            (2)
public class HotelSearchingAction implements HotelSearching
{

  @PersistenceContext
  private EntityManager em;

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

  @DataModel                                                 (3)
  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 method with no parameters marked `@Remove`. This method will be called when Seam destroys the session context.

The main page of the application is a Facelets page. Let's look at the fragment which relates to searching for hotels:

Example 1.26. main.xhtml

```
<div class="section">

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

  <h1>Search Hotels</h1>

  <h:form id="searchCriteria">
  <fieldset>
    <h:inputText id="searchString" value="#{hotelSearch.searchString}" style="width: 165px;">
     <a:ajax event="keyup" render="searchResults" listener="#{hotelSearch.find}"/>
    </h:inputText>                                                    1
     
    <a:commandButton id="findHotels" value="Find Hotels" actionListener="#{hotelSearch.find}"
 render="searchResults"/>
     
    <a:status id="status">
      <f:facet id="StartStatus" name="start">
        <h:graphicImage id="SpinnerGif" value="/img/spinner.gif"/>        2
      </f:facet>
    </a:status>
    <br/>
            <h:outputLabel id="MaximumResultsLabel" for="pageSize">Maximum results:</
h:outputLabel> 
    <h:selectOneMenu id="pageSize" value="#{hotelSearch.pageSize}">
      <f:selectItem id="PageSize5" itemLabel="5" itemValue="5"/>
      <f:selectItem id="PageSize10" itemLabel="10" itemValue="10"/>
      <f:selectItem id="PageSize20" itemLabel="20" itemValue="20"/>
    </h:selectOneMenu>
```

```
   </fieldset>
   </h:form>

</div>

<a:outputPanel id="searchResults">
 <div class="section">
   <h:outputText id="NoHotelsFoundMessage" 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 id="column1">
         <f:facet id="NameFacet" name="header">Name</f:facet>
         #{hot.name}
      </h:column>
      <h:column id="column2">
         <f:facet id="AddressFacet" name="header">Address</f:facet>
         #{hot.address}
      </h:column>
      <h:column id="column3">
         <f:facet id="CityStateFacet" name="header">City, State</f:facet>
         #{hot.city}, #{hot.state}, #{hot.country}
      </h:column>
      <h:column id="column4">
         <f:facet id="ZipFacet" name="header">Zip</f:facet>
         #{hot.zip}
      </h:column>
      <h:column id="column5">
         <f:facet id="ActionFacet" name="header">Action</f:facet>
         <s:link id="viewHotel" value="View Hotel" action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
   </h:dataTable>
        <s:link id="MoreResultsLink" value="More results" action="#{hotelSearch.nextPage}"
 rendered="#{hotelSearch.nextPageAvailable}"/>
 </div>
</a:outputPanel>
```

1. The RichFaces `<a:ajax>` tag allows a JSF action event listener to be called by asynchronous `XMLHttpRequest` when a JavaScript event like `onkeyup` occurs. Even better, the `render` 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 `<a:status>` tag lets us display an animated image while we wait for asynchronous requests to return.

3. The RichFaces `<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".

   If you're 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 the search results dynamically as we type, and lets us choose a hotel and pass it to the `selectHotel()` method of the `HotelBookingAction`, which is where the *really* interesting stuff is going to happen.

Now let's see how the booking example application uses a conversation-scoped stateful session bean to achieve a natural cache of persistent data related to the conversation. The following code example is pretty long. But if you think of it as a list of scripted actions that implement the various steps of the conversation, it's understandable. Read the class from top to bottom, as if it were a story.

Example 1.27. 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                                                    4
  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                                                 5
  public void destroy() {}
```

1. This bean uses an EJB3 *extended persistence context*, so that any entity instances remain managed for the whole lifecycle 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. Don't 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. We think you'll agree that this code is much cleaner and simpler than getting and setting `HttpSession` attributes.

Even better, a user can have multiple isolated conversations per login session. Try it! 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 backbutton 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`. The Seam debug page will be available if this jar is deployed in `WEB-INF/lib`, along with the Facelets, and if you set the debug property of the `init` component:

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

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

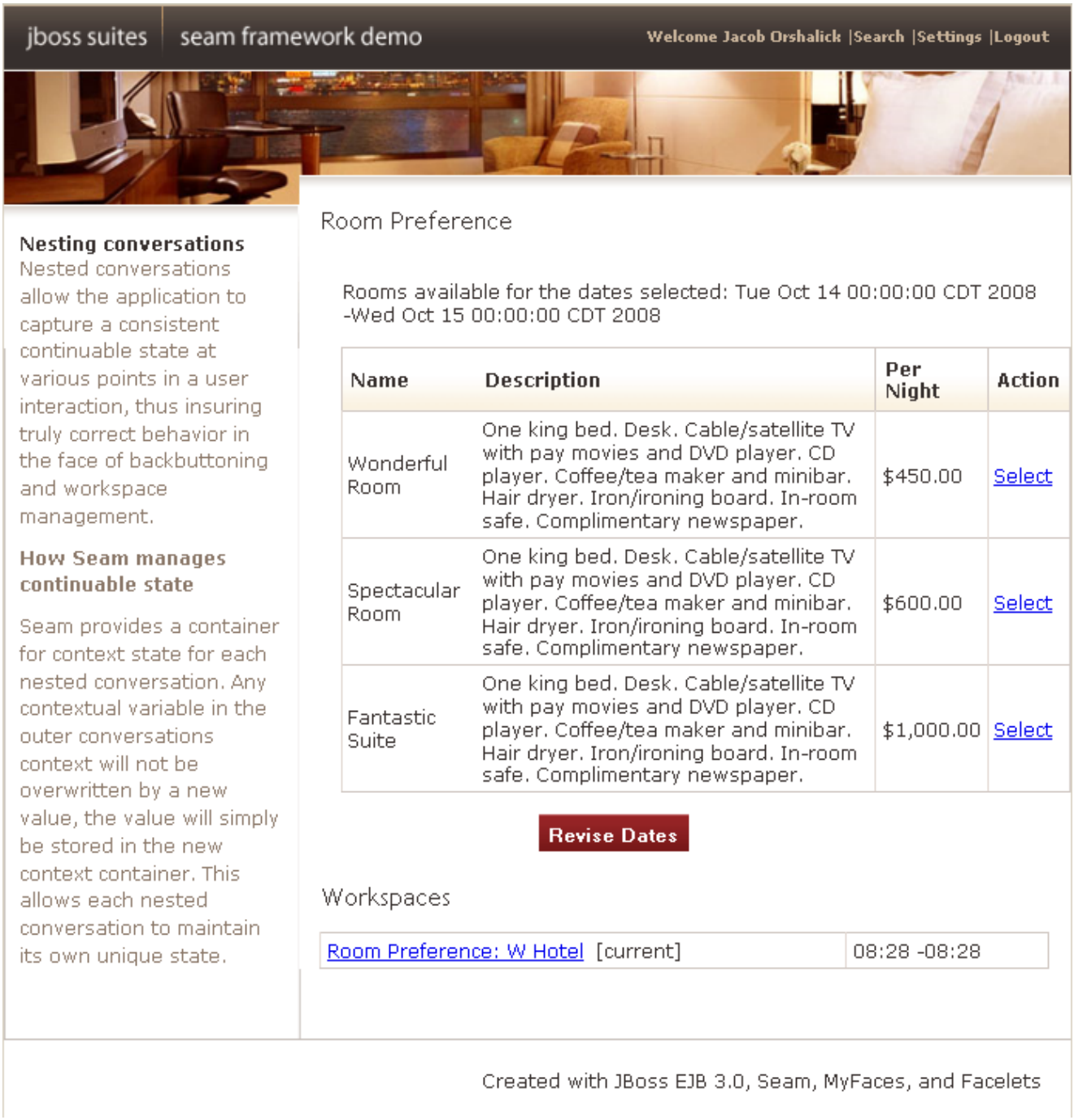
+ Application Context

1.7. Nested conversations: extending the Hotel Booking example

1.7.1. Introduction

Long-running conversations make it simple to maintain consistency of state 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 what the user's expectations and the reality of the application's state can still result.

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



The user now has the option to select any available room to be included in the booking. As with the hotel booking application we saw previously, this can lead to issues with state consistency. As with storing state in the `HTTPSession`, if a conversation variable changes it affects all windows operating within the same conversation context.

To demonstrate this, let's 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 see just how much it would cost to live the high-life, 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 decides that practicality wins out and returns to the window showing *Wonderful Room* to confirm.

In this scenario, if we simply store all state in the conversation, we are not protected from multi-window operation within the same conversation. Nested conversations allow us to achieve correct behavior even when context can vary within the same conversation.

1.7.2. Understanding Nested Conversations

Now let's see how the nested booking example extends the behavior of the hotel booking application through use of nested conversations. Again, we can read the class from top to bottom, as if it were a story.

Example 1.28. RoomPreferenceAction.java

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

  @Logger
  private Log log;

  @In private Hotel hotel;

  @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()                    (1)
  {
```

```
    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)                                    (2)
  public String selectPreference()
  {
    log.info("Room selected");

    this.room = this.roomSelection;                      (3)

    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()                                 (4)
  {
    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 outer long-running conversation (the booking) and each of the nested conversations (room selections).

Example 1.29. 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}"/>

                              <h:dataTable id="rooms" value="#{availableRooms}" var="room"
 rendered="(1)#{availableRooms.rowCount &amp;gt; 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(2).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"/>     (3)
        </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 we have seen how to nest a conversation, let's see how we can confirm the booking once a room has been selected. This can be achieved by simply extending the behavior of the `HotelBookingAction`.

Example 1.30. 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)
  public void confirm()                                          (1)
  {
    // 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.

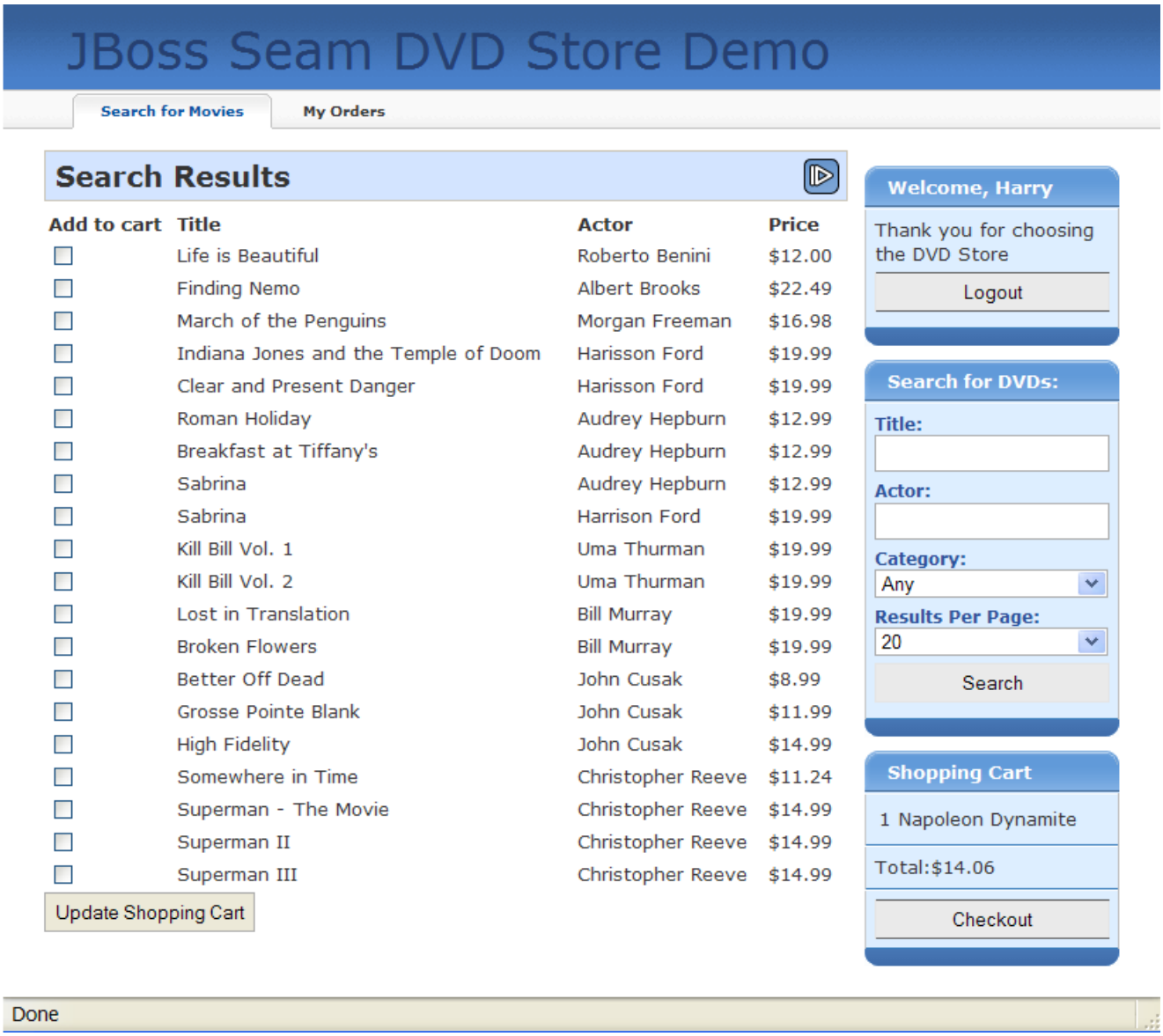
(3) By simply annotating the cancellation action with `@End(root=true, beforeRedirect=true)` we can easily destroy and release all state associated with the workspace prior to redirecting the user back to the hotel selection view.

Feel free to deploy the application, open many windows or tabs and attempt combinations of various hotels with various room preferences. Confirming a booking always results in the correct hotel and room preference thanks to 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 searching and shopping cart functionality.

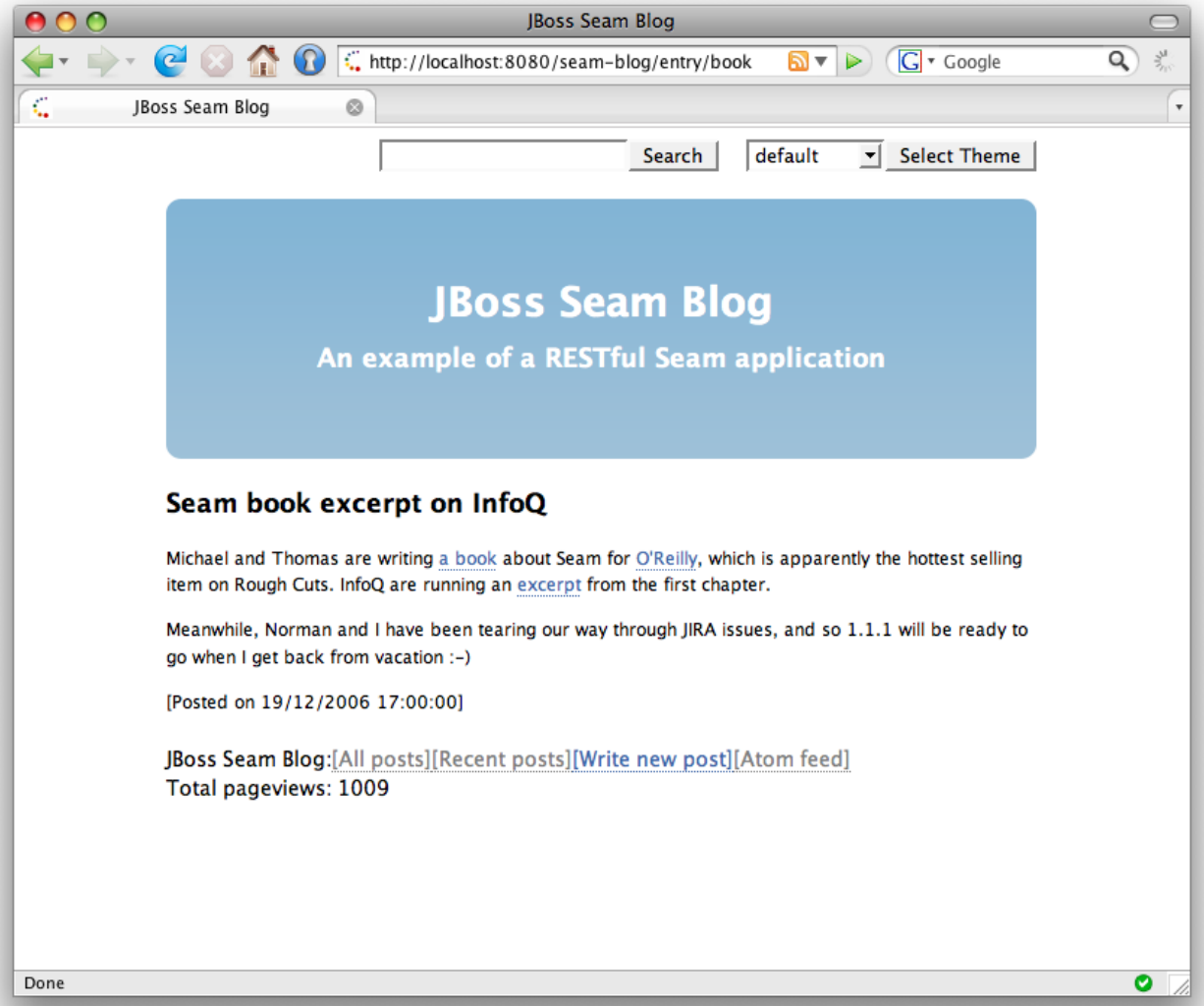


The administration screens take use jBPM to manage the approval and shipping cycle for orders. The business process may even be changed dynamically, by selecting a different process definition!

The Seam DVD Store demo can be run from `dvdstore` directory, just like the other demo applications.

1.9. Bookmarkable URLs with the Blog example

Seam makes it very easy to implement applications which keep state on the server-side. However, server-side state is not always appropriate, especially in for functionality that serves up *content*. For this kind of problem we often want to keep application state in the URL so that any page can be accessed at any time through a bookmark. The blog example shows how to a implement an application that supports bookmarking throughout, even on the search results page. This example demonstrates how Seam can manage application state in the URL as well as how Seam can rewrite those URLs to be even



The Blog example demonstrates the use of "pull"-style MVC, where instead of using action listener methods to retrieve data and prepare the data for the view, the view pulls data from components as it is being rendered.

1.9.1. Using "pull"-style MVC

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

Example 1.31.

```
<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, how does the `#{blog.recentBlogEntries}` data used by the `<h:dataTable>` actually get initialized? The `Blog` is retrieved lazily — "pulled" — when needed, by a Seam component named `blog`. This is the opposite flow of control to what is used in traditional action-based web frameworks like Struts.

Example 1.32.

```
@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 is good so far, but what about bookmarking the result of form submissions, such as a search results page?

1.9.2. Bookmarkable search results page

The blog example has a tiny form in the top right of each page that allows the user to search for blog entries. This is defined in a file, `menu.xhtml`, included by the facelets template, `template.xhtml`:

Example 1.33.

```
<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, we need to perform a browser redirect after processing the search form submission. Because we used the JSF view id as the action outcome, Seam automatically redirects to the view id when the form is submitted. Alternatively, we could have defined a navigation rule like this:

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

Then 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>
```

But when we redirect, we need to include the values submitted with the form in the URL to get a bookmarkable URL like `http://localhost:8080/seam-blog/search/`. JSF does not provide an easy way to do this, but Seam does. We use two Seam features to accomplish this: *page parameters* and *URL rewriting*. Both are defined in `WEB-INF/pages.xml`:

Example 1.34.

```
<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 request parameter named `searchPattern` to the value of `#{searchService.searchPattern}`, both whenever a request for the Search page comes in and whenever a link to the search page is generated. Seam takes responsibility for maintaining the link between URL state and application state, and you, the developer, don't have to worry about it.

Without URL rewriting, the URL for a search on the term `book` would be `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book`. This is nice, but Seam can make the URL even simpler using a rewrite rule. The first rewrite rule, for the pattern `/search/{searchPattern}`, says that any time we have a URL for search.xhtml with a searchPattern request parameter, we can fold that URL into the simpler URL. So,the URL we saw earlier, `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book` can be written instead as `http://localhost:8080/seam-blog/search/book`.

Just like with page parameters, URL rewriting is bi-directional. That means that Seam forwards requests for the simpler URL to the right view, and it also automatically generates the simpler

view for you. You never need to worry about constructing URLs. It's all handled transparently behind the scenes. The only requirement is that to use URL rewriting, the rewrite filter needs to be enabled 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>
```

Which again uses "pull"-style MVC to retrieve the actual search results using 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

Very occasionally, it makes more sense to use push-style MVC for processing RESTful pages, and so Seam provides the notion of a *page action*. The Blog example uses a page action for the blog entry page, `entry.xhtml`. Note that this is a little bit contrived, it would have been easier to use pull-style MVC here as well.

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>
```

Notice that the example is using page actions for post validation and the pageview counter. Also notice the use of a parameter in the page action method binding. This was not a standard feature of JSF EL in Java EE 5, but now it is and works like Seam lets you use it before, not just for page actions but also in JSF method bindings.

When the `entry.xhtml` page is requested, Seam first binds the page parameter `blogEntryId` to the model. Keep in mind that because of the URL rewriting, the blogEntryId parameter name won't show up in the URL. Seam then runs the page action, which retrieves the needed data — the `blogEntry` — and places it in the Seam event context. Finally, the following is rendered:

```
<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>
```

It is a matter of taste which implementation you prefer.

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

Getting started with Seam, using seam-gen

The Seam distribution includes a command line utility that makes it really easy to set up an Eclipse project, generate some simple Seam skeleton code, and reverse engineer an application from a preexisting database.

This is the easy way to get your feet wet with Seam, and gives you some ammunition for next time you find yourself trapped in an elevator with one of those tedious Ruby guys ranting about how great and wonderful his new toy is for building totally trivial applications that put things in databases.

In this release, seam-gen works best for people with JBoss AS. You can use the generated project with other J2EE or Java EE 5 application servers by making a few manual changes to the project configuration.

You *can* use seam-gen without Eclipse, but in this tutorial, we want to show you how to use it in conjunction with Eclipse for debugging and integration testing. If you don't want to install Eclipse, you can still follow along with this tutorial—all steps can be performed from the command line.

seam-gen is basically just an intricate Ant script wrapped around Hibernate Tools, together with some templates. That makes it easy to customize if you need to.

2.1. Before you start

Make sure you have JDK 6 (see *Section 39.1, "JDK Dependencies"* for details), JBoss AS 7.1.1 and Maven 3.x, along with recent versions of Eclipse, the JBoss IDE plugin for Eclipse correctly installed before starting. Add your JBoss installation to the JBoss Server View in Eclipse. Start JBoss in debug mode. Finally, start a command prompt in the directory where you unzipped the Seam distribution.

JBoss has sophisticated support for hot re-deployment of WARs and EARs. Unfortunately, due to bugs in the JVM, repeated redeployment of an EAR—which is common during development—eventually causes the JVM to run out of perm gen space. For this reason, we recommend running JBoss in a JVM with a large perm gen space at development time. If you're 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
```

If you don't have so much memory available, the following is our minimum recommendation:

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

If you're running JBoss from the command line, you can configure the JVM options in `bin/standalone.conf`.

If you don't want to bother with this stuff now, you don't have to—come back to it later, when you get your first `OutOfMemoryException`.

2.2. Setting up a new project

The first thing we need to do is configure seam-gen for your environment: JBoss AS installation directory, project workspace, and database connection. It's easy, just type:

```
cd jboss-seam-2.3.0
seam setup
```

And you will be prompted for the needed information:

```
~/workspace/jboss-seam$ ./seam setup
Buildfile: build.xml

init:

setup:
     [echo] Welcome to seam-gen :-)
     [input] Enter your project workspace (the directory that contains your Seam projects) [C:/
Projects] [C:/Projects]
/Users/pmuir/workspace
   [input] Enter your JBoss AS home directory [C:/Program Files/jboss-as-7.1.1.Final] [C:/Program
 Files/jboss-as-7.1.1.Final]
/Applications/jboss-as-7.1.1.Final
    [input] Enter the project name [myproject] [myproject]
helloworld
     [echo] Accepted project name as: helloworld
    [input] Select a RichFaces skin [blueSky] ([blueSky], emeraldTown, ruby, classic, japanCherry,
 wine, deepMarine, DEFAULT, plain)

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

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

```
org.jboss.helloworld
      [input] Enter the Java package name for your entity beans [org.jboss.helloworld]
[org.jboss.helloworld]

      [input] Enter the Java package name for your test cases [org.jboss.helloworld.test]
[org.jboss.helloworld.test]

   [input] What kind of database are you using? [h2]  ([h2], hsql, mysql, oracle, postgres, mssql,
db2, sybase, enterprisedb)
mysql
    [input] Enter the Hibernate dialect for your database [org.hibernate.dialect.MySQLDialect]
[org.hibernate.dialect.MySQLDialect]

   [input] Enter the filesystem path to the JDBC driver jar [lib/hsqldb.jar] [lib/hsqldb.jar]
/Users/pmuir/java/mysql.jar
         [input] Enter JDBC driver class for your database [com.mysql.jdbc.Driver]
[com.mysql.jdbc.Driver]

   [input] Enter the JDBC URL for your database [jdbc:mysql:///test] [jdbc:mysql:///test]
jdbc:mysql:///helloworld
   [input] Enter database username [sa] [sa]
pmuir
   [input] Enter database password [] []

   [input] skipping input as property hibernate.default_schema.new has already been set.
   [input] Enter the database catalog name (it is OK to leave this blank) [] []

   [input] Are you working with tables that already exist in the database? [n]  (y, [n], )
y
   [input] Do you want to drop and recreate the database tables and data in import.sql each time
you deploy? [n]  (y, [n], )
n

[propertyfile] Creating new property file: /Users/pmuir/workspace/jboss-seam/seam-gen/
build.properties
    [echo] Installing JDBC driver jar to JBoss server
    [echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
Total time: 1 minute 32 seconds
~/workspace/jboss-seam $
```

The tool provides sensible defaults, which you can accept by just pressing enter at the prompt.

The most important choice you need to make is between EAR deployment and WAR deployment of your project. EAR projects support EJB 3.0 and require Java EE 5. WAR projects do not support EJB 3.0, but may be deployed to a J2EE environment. The packaging of a WAR is also simpler to understand. If you installed an EJB3-ready application server like JBoss, choose `ear`. Otherwise, choose `war`. We'll assume that you've chosen an EAR deployment for the rest of the tutorial, but you can follow exactly the same steps for a WAR deployment.

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

The settings are stored in `seam-gen/build.properties`, but you can also modify them simply by running `seam setup` a second time.

Now we can create a new project in our Eclipse workspace directory, by typing:

```
seam new-project
```

```
C:\Projects\jboss-seam>seam new-project
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 jars, dependent jars and the JDBC driver jar to a new Eclipse project, and generates all needed 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 as soon as you add the project using `New -> Project... -> General -> Project -> Next`, typing the `Project name` (`helloworld` in this case), and then clicking `Finish`. Do not select `Java Project` from the New Project wizard.

If your default JDK in Eclipse is not a Java SE 6 JDK, you will need to select a Java SE 6 compliant JDK using `Project -> Properties -> Java Compiler`.

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

Go to `http://localhost:8080/helloworld` to see a welcome page. This is a facelets page, `view/home.xhtml`, using the template `view/layout/template.xhtml`. You can edit this page, or the template, in Eclipse, and see the results *immediately*, by clicking refresh in your browser.

Don't get scared by the XML configuration documents that were generated into the project directory. They are mostly standard Java EE stuff, the stuff you need to create once and then never look at again, and they are 90% the same between all Seam projects. (They are so easy to write that even seam-gen can do it.)

The generated project includes three database and persistence configurations. The `persistence-test.xml` and `import-test.sql` files are used when running the TestNG unit tests against HSQLDB. The database schema and the test data in `import-test.sql` is always exported to the database before running tests. The `myproject-dev-ds.xml`, `persistence-dev.xml`and `import-dev.sql` files are for use when deploying the application to your development database. The schema might be exported automatically at deployment, depending upon whether you told seam-gen that you are working with an existing database. The `myproject-prod-ds.xml`, `persistence-prod.xml`and `import-prod.sql` files are for use when deploying the application to your production database. The schema is not exported automatically at deployment.

2.3. Creating a new action

If you're used to traditional action-style web frameworks, you're probably wondering how you can create a simple web page with a stateless action method in Java. If you type:

```
seam new-action
```

Seam will prompt for some information, and generate a new facelets page and Seam component for your project.

```
C:\Projects\jboss-seam>seam new-action
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>
```

Because we've added a new Seam component, we need to restart the exploded directory deployment. You can do this by typing `seam restart`, or by running the `restart` target in the generated project `build.xml` file from inside Eclipse. Another way to force a restart is to edit the file `resources/META-INF/application.xml` in Eclipse. *Note that you do not need to restart JBoss each time you change the application.*

Now go to `http://localhost:8080/helloworld/ping.seam` and click the button. You can see the code behind this action by looking in the project `src` directory. Put a breakpoint in the `ping()` method, and click the button again.

Finally, locate the `PingTest.xml` file in the test package and run the integration tests using the TestNG plugin for Eclipse. Alternatively, run the tests using `seam test` or the `test` target of the generated build.

2.4. Creating a form with an action

The next step is to create a form. Type:

```
seam new-form
```

```
C:\Projects\jboss-seam>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`. Then take a look at the generated code. Run the test. Try adding some new fields to the form and Seam component (remember to restart the deployment each time you change the Java code).

2.5. Generating an application from an existing database

Manually create some tables in your database. (If you need to switch to a different database, just 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. If you look at the generated code, you'll probably be amazed how simple it is! Seam was designed so that data access code is easy to write by hand, even for people who don't want to cheat by using seam-gen.

2.6. Generating an application from existing JPA/EJB3 entities

Place your existing, valid entity classes inside the `src/main`. Now type

```
seam generate-ui
```

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

2.7. Deploying the application as an EAR

Finally, we want to be able to deploy the application using standard Java EE 5 packaging. First, we need to remove the exploded directory by running `seam unexplode`. To deploy the EAR, we can type `seam deploy` at the command prompt, or run the `deploy` target of the generated project build script. You can undeploy using `seam undeploy` or the `undeploy` target.

By default, the application will be deployed with the *dev profile*. The EAR will include the `persistence-dev.xml` and `import-dev.sql` files, and the `myproject-dev-ds.xml` file will be deployed. You can change the profile, and use the *prod profile*, by typing

```
seam -Dprofile=prod deploy
```

You can even 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 using `-Dprofile=staging`.

2.8. Seam and incremental hot deployment

When you deploy your Seam application as an exploded directory, you'll get some support for incremental hot deployment at development time. You need to enable debug mode in both Seam and Facelets, by adding this line to `components.xml`:

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

Now, the following files may be redeployed without requiring a full restart of the web application:

- any facelets page
- any `pages.xml` file

But if we want to change any Java code, we still need to do a full restart of the application. (In JBoss this can be handled by configuring *deployment scanner mode* [https://docs.jboss.org/author/display/AS7/Deployment+Scanner+configuration] - more details how to do that are in `$JBOSS_HOME/standalone/deployments/README.txt`

But if you really want a fast edit/compile/test cycle, Seam supports incremental redeployment of JavaBean components. To make use of this functionality, you must deploy the JavaBean components into the `WEB-INF/dev` directory, so that they will be loaded by a special Seam classloader, instead of by the WAR or EAR classloader.

You need to be aware of the following limitations:

- the components must be JavaBean components, they cannot be EJB3 beans (we are working on fixing 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`
- You must have the Seam filter installed in web.xml
- You may see errors if the system is placed under any load and debug is enabled.

If you create a WAR project using 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.

Getting started with Seam, using JBoss Tools

JBoss Tools is a collection of Eclipse plugins. JBoss Tools a project creation wizard for Seam, Content Assist for the Unified Expression Language (EL) in both facelets and Java code, a graphical editor for Seam configuration files, support for running Seam integration tests from within Eclipse, and much more.

In short, if you are an Eclipse user, then you'll want JBoss Tools!

Please read the latest JBoss Tools documentation at *http://docs.jboss.org/tools/latest/en/seam_tools_ref_guide/html/index.html*.

JBoss Tools, as with seam-gen, works best with JBoss AS, but it's possible with a few tweaks to get your app running on other application servers. The changes are much like those described for seam-gen later in this reference manual.

3.1. Before you start

Make sure you have JDK 6, JBoss AS 7.1.1.Final, Eclipse 3.7, the JBoss Tools plugins (at least Seam Tools, the Visual Page Editor and JBoss AS Tools) and the JUnit plugin for Eclipse correctly installed before starting.

Please see the official *JBoss Tools Getting started* [http://docs.jboss.org/tools/latest/en/GettingStartedGuide/html_single/index.html] page for the quickest way to get JBoss Tools setup in Eclipse.

Migration from 2.2 to 2.3

Before you get started with Seam 2.3, there are a few things you should be aware of. This process should not be too painful - if you get stuck, just refer back to the updated Seam examples in Seam distribution.

This migration guide assumes you are using Seam 2.2, if you are migrating from Seam 1.2 or 2.0, see the `jboss-seam-x.y.z.Final/seam2migration.txt` and `jboss-seam-x.y.z.Final/seam21migration.txt` guide as well.

4.1. Migration of XML Schemas

4.1.1. Seam schema migration

XML schemas for validation Files that use the Seam 2.2 XSDs should be updated to refer to the 2.3 XSDs, notice the version change. Current namespace pattern is `www.jboss.org/schema/seam/*` and schemaLocation URL was changed to `www.jboss.org/schema/seam/*_-2.3.xsd`, where `*` is Seam module.

Following snippet is an example of component declaration for 2.2 version:

Example 4.1. Before migration of Seam `components.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
  <components xmlns="http://jboss.com/products/seam/components"
        xmlns:core="http://jboss.com/products/seam/core"
        xmlns:persistence="http://jboss.com/products/seam/persistence"
        xmlns:security="http://jboss.com/products/seam/security"
        xmlns:theme="http://jboss.com/products/seam/theme"
        xmlns:cache="http://jboss.com/products/seam/cache"
        xmlns:web="http://jboss.com/products/seam/web"
        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/persistence  http://jboss.com/products/seam/
persistence-2.2.xsd
          http://jboss.com/products/seam/security http://jboss.com/products/seam/security-2.2.xsd
             http://jboss.com/products/seam/theme http://jboss.com/products/seam/theme-2.2.xsd
             http://jboss.com/products/seam/cache http://jboss.com/products/seam/cache-2.2.xsd
             http://jboss.com/products/seam/web http://jboss.com/products/seam/web-2.2.xsd
                      http://jboss.com/products/seam/components  http://jboss.com/products/seam/
components-2.2.xsd">
```

And finally migrated declaration of `components.xml` for 2.3 version:

Example 4.2. Migrated `components.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
    <components xmlns="http://jboss.org/schema/seam/components"
                xmlns:core="http://jboss.org/schema/seam/core"
                xmlns:persistence="http://jboss.org/schema/seam/persistence"
                xmlns:security="http://jboss.org/schema/seam/security"
                xmlns:theme="http://jboss.org/schema/seam/theme"
                xmlns:cache="http://jboss.org/schema/seam/cache"
                xmlns:web="http://jboss.org/schema/seam/web"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation=
                    "http://jboss.org/schema/seam/core http://jboss.org/schema/seam/core-2.3.xsd
                        http://jboss.org/schema/seam/persistence http://jboss.org/schema/seam/
persistence-2.3.xsd
                http://jboss.org/schema/seam/security http://jboss.org/schema/seam/security-2.3.xsd
                    http://jboss.org/schema/seam/theme http://jboss.org/schema/seam/theme-2.3.xsd
                    http://jboss.org/schema/seam/cache http://jboss.org/schema/seam/cache-2.3.xsd
                    http://jboss.org/schema/seam/web http://jboss.org/schema/seam/web-2.3.xsd
                            http://jboss.org/schema/seam/components http://jboss.org/schema/seam/
components-2.3.xsd">
```

Next remainning migration step is `pages.xml` file(s) as well as other files only requires that the schemas be upgraded.

Example 4.3. Before migration of Seam `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">
      ...
</pages>
```

Example 4.4. After migration of Seam `pages.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.org/schema/seam/pages"
```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.org/schema/seam/pages http://jboss.org/schema/seam/
pages-2.3.xsd">
      ...
</pages>
```

4.1.2. Java EE 6 schema changes

Seam 2.3 technology upgrade includes also Java EE 6 upgrade so you need to update the following descriptors

- `persistence.xml` for using JPA 2
- `web.xml` for using Servlet 3.0 and Web application
- `application.xml` for using Enterprise Java 6 application
- `faces-config.xml` if you need to specify some advanced configuration for JSF 2 (this desciptor file is not mandatory, you don't have to use/include it in your application)

Examples of changed headers with correct versions are the following:

Example 4.5. persistence.xml

```
<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_2_0.xsd"
      version="2.0">
```

Example 4.6. application.xml

```
<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_6.xsd"
      version="6">
```

Example 4.7. web.xml

```
<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_3_0.xsd"
        version="3.0">
```

Example 4.8. faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
  <faces-config version="2.1"
            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_2_1.xsd">
```

4.2. Java EE 6 upgrade

Seam 2.3 can integrate with the major upgrades in Java EE (from 5 to 6). You can use persistence with JPA 2, EJB 3.1 and Bean Validation. Almost all EE 6 technology upgrade requires to change XML schema declaration. See *Section 4.1.2, "Java EE 6 schema changes"*

4.2.1. Using Bean Validation standard instead of Hibernate Validator

Bean Validation is a standard included in Java EE 6 as new technology. Seam already uses for validation Hibernate Validator which is a reference implementation.

You need to migrate from using of `org.hibernate.validator.*` Hibernate validator annotations to `javax.validation.constraint.*` equivalent for instance Seam examples used a lot of the following annotations and you can use this list as a helper (*Using Bean Validation* [http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html]):

- `org.hibernate.validator.Length` to `javax.validation.constraint.Size`,
- `org.hibernate.validator.NotNull` to `javax.validation.constraint.NotNull`,
- `org.hibernate.validator.Pattern` to `javax.validation.constraint.Pattern`.

4.2.2. Migration of JSF 1 to JSF 2 Facelets templates

Configuration file `faces-config.xml` is not required to be in your application, so for simple using of JSF 2 you need to migrate only `web.xml`. If you anyway would like to have it, change the XML schema declaration as is described in *Example 4.8, "faces-config.xml"*.

All your application JSF templates should use only facelets technology as JSP is deprecated.

In facelet templates there are required to convert `<head>/<body>` tags to `><h:head>/<h:body>`respectively.

Depending on what JSF components that you use like Richfaces or Icefaces, there has to be some differences when upgrading from JSF 1.x to JSF 2.x. You may need to upgrade libraries entirely. Consult any component framework documentation on those changes. This migration doesn't cover these migration steps.

4.2.3. Migration to JPA 2.0

Using JPA 2 requires to change version to 2.0 in `persistence.xml`, see *Example 4.5, "persistence.xml"* file and version in `application.xml` should be 6 if you are using EAR - see *Example 4.6, "application.xml"* or version in `web.xml` file change to 3.0 if you use only WAR - look at *Example 4.7, "web.xml"*.

What is important for developers, most application can use just WAR with EJB 3.1 and doesn't have to package application as EAR.

JPA 2.0 is backward compatible with JPA 1.0, so you don't have to migrate any JPA annotation or classes. JPA 2.0 is more like enhancement to JPA 1.0.

4.2.4. Using compatible JNDI for resources

Java EE 6 brings new standardized global rules for creating portable JNDI syntax. So you have to change all JNDI strings from `_your_application_/#{ejbName}/local` to `java:app/_application-module-name_/#{ejbName}` like for instance in `WEB-INF/components.xml` change of `jndiPattern` from:

```
seam-mail/#{ejbName}/local
```

to

```
java:app/seam-mail-ejb/#{ejbName}
```

4.3. JBoss AS 7.1 deployment

4.3.1. Deployment changes

Next level is migration of your target runtime. Seam 2.3 uses JBoss AS 7 as default target runtime.

If you are using for development or testing default datasource in JBoss AS 7.1, you need to change datasource JNDI in your `persistence.xml` from `java:/DefaultDS` to `java:jboss/datasources/ExampleDS`.

JBoss AS 7 has got refactored classloading model. Classloading of bundled or provided libraries can be managed in `jboss-deployment-structure.xml` or in `META-INF/MANIFEST.MF` file in section `Dependencies`. This migration documentation prefers using of `jboss-deployment-structure.xml` file, which should be placed in `META-INF` directory of your WAR or EAR application according to your application type.

For full EAR projects, the `jboss-deployment-structure.xml` will be located in the `_your_ear_/META-INF` directory.

For Web (non-ear) projects, the `jboss-deployment-structure.xml` will be located in the `_your_war_/WEB-INF` directory.

Minimal content for Seam 2.3 based application is:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
    <deployment>
        <dependencies>
            <module name="org.dom4j" export="true"/>
            <module name="org.apache.commons.collections" export="true"/>
                <module name="javax.faces.api" export="true"/> <!-- keep there only if you use JSF
 as view technology -->
        </dependencies>
    </deployment>
  </jboss-deployment-structure>
```

More details are described in *JBoss AS 7 documentation* [https://docs.jboss.org/author/display/AS7/Class+Loading+in+AS7].

4.3.2. Datasource migration

You can also include now any database descriptor (*-ds.xml) files into your project in the `META-INF` directory, and the data source will be deployed automatically when deployed to a JBoss AS 7.1 Application Server. The structure of the file though has changed. Before the datasource file was a simple xml based file, but now is an *IronJacamar* [https://www.jboss.org/ironjacamar] based file. Iron-Jacamar is the JBoss' JCA (Java Connector Architecture) project. Below on *Example 4.9, “Sample Seam 2.2 Datasource Descriptor File”* is the former datasource for JBoss AS 4/5, and *Example 4.10, “Ironjacamar Datasource Descriptor File”* shows the conversion to IronJacamar using the same driver, url, and credentials.

Example 4.9. Sample Seam 2.2 Datasource Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE datasources
          PUBLIC "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
```

```
        "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
  <datasources>
    <local-tx-datasource>
      <jndi-name>seamdiscsDatasource</jndi-name>
      <connection-url>jdbc:hsqldb:.</connection-url>
      <driver-class>org.hsqldb.jdbcDriver</driver-class>
      <user-name>sa</user-name>
      <password></password>
    </local-tx-datasource>
  </datasources>
```

Example 4.10. Ironjacamar Datasource Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
  <datasources xmlns="http://www.jboss.org/ironjacamar/schema">
    <datasource
      jndi-name="java:/jboss/seamdiscsDatasource"
      enabled="true"
      use-java-context="true" pool-name="seamdiscs">
      <connection-url>jdbc:hsqldb:.</connection-url>
      <driver>org.hsqldb.jdbcDriver</driver>
      <security>
        <user-name>sa</user-name>
        <password></password>
      </security>
    </datasource>
  </datasources>
```

4.4. Changes in testing framework

SeamTest and JBoss Embedded are legacy components and have many limitations and we doesn't support it like we did in Seam 2.2.

We now bring Arquillian as the replacement of JBoss Embedded and you should extend `org.jboss.seam.mock.JUnitSeamTest` instead of `org.jboss.seam.mock.SeamTest`, DBUnit testing is provided by `org.jboss.seam.mock.DBJUnitSeamTest` instead of `org.jboss.seam.mock.DBUnitSeamTest`. Due assertion issues with TestNG framework and Arquillian, we use JUnit as preferred test framework. Migration to Junit and Arquillian goes in the following steps:

1. Add

```
@RunWith(Arquillian.class)
```

annotation to your test class.

2. Your test class should extend `org.jboss.seam.mock.JUnitSeamTest` instead of `org.jboss.seam.mock.SeamTest`.

3. Add a method like

```
@Deployment(name="_your_test_name_")
@OverProtocol("Servlet 3.0")
public static org.jboss.shrinkwrap.api.Archive<?> createDeployment(){}
```

for creating test deployment archive. The following example is taken from Registration example testsuite. It imports the .ear archive buit with "mvn package", adds the test class to the deployment and replaces the WEB-INF/web.xml file with a SeamTest-specific version.

```
@Deployment(name="RegisterTest")
@OverProtocol("Servlet 3.0")
public static Archive<?> createDeployment()
{
        EnterpriseArchive er = ShrinkWrap.create(ZipImporter.class, "seam-registration.ear").importFrom(new File("../registration-ear/target/seam-registration.ear"))
        .as(EnterpriseArchive.class);
  WebArchive web = er.getAsType(WebArchive.class, "registration-web.war");
  web.addClasses(RegisterTest.class);

  // Install org.jboss.seam.mock.MockSeamListener
  web.delete("/WEB-INF/web.xml");
  web.addAsWebInfResource("web.xml");

  return er;
}
```

4. Create a SeamTest-specific web.xml contaning the org.jboss.seam.mock.MockSeamListener, instead of the usual org.jboss.seam.servlet.SeamListener.

```
<?xml version="1.0" ?>
<web-app version="3.0"
```

```
        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_3_0.xsd">

   <listener>
      <listener-class>org.jboss.seam.mock.MockSeamListener</listener-class>
   </listener>

</web-app>
```

5. Add `arquillian.xml` file into root of your classpath for running Arquillian test(s). The file content should specify path to remote or managed container and some specific options for JVM or Arquillian. The example of arquillian file is at `jboss-seam-x.y.z.Final/examples/booking/booking-tests/src/test/resources-integration/arquillian.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://jboss.org/schema/arquillian"
   xsi:schemaLocation="http://jboss.org/schema/arquillian http://jboss.org/schema/arquillian/
arquillian_1_0.xsd">
   <engine>
      <property name="deploymentExportPath">target/</property>
   </engine>
   <container qualifier="jboss" default="true">
      <configuration>
      <property name="javaVmArguments">-Xmx1024m -XX:MaxPermSize=512m</property>
         <property name="jbossHome">target/jboss-as-7.1.1.Final</property>
      </configuration>
   </container>
</arquillian>
```

More details in Seam reference documentation guide in *Section 38.2, "Integration testing Seam components"*.

4.5. Dependency changes with using Maven

The "provided" platform is now JBoss AS 7.1.x as is written above, therefore all Java EE dependencies included in AS 7 are now marked as provided.

4.5.1. Seam Bill of Materials

A Bill of materials is a set of dependeny elements in `<dependencyManagement>` section that can be used to import into your application maven build and be able to declare which dependencies

and their versions that you wish to use in your application. The nice thing about the Seam BOM is that the dependencies and their versions are there recommended dependencies that would work well with Seam 2.3. The usage of Seam BOM is shown in *Example 4.11, "Seam BOM usage"*. The Seam BOM is deployed in *JBoss Maven repository* [https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.jboss.seam~bom~~~].

Example 4.11. Seam BOM usage

```
<dependencyManagement>
        <dependencies>
          <dependency>
            <groupId>org.jboss.seam</groupId>
            <artifactId>bom</artifactId>
            <version>2.3.1.Final</version>
            <type>pom</type>
            <scope>import</scope>
          </dependency>
         ...
       </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.jboss.seam</groupId>
      <artifactId>jboss-seam</artifactId>
      <type>ejb</type>
    <dependency>
    ...
  </dependencies>
```

The contextual component model

The two core concepts in Seam are the notion of a *context* and the notion of a *component*. Components are stateful objects, usually EJBs, and an instance of a component is associated with a context, and given a name in that context. *Bijection* provides a mechanism for aliasing internal component names (instance variables) to contextual names, allowing component trees to be dynamically assembled, and reassembled by Seam.

Let's start by describing the contexts built in to Seam.

5.1. Seam contexts

Seam contexts are created and destroyed by the framework. The application does not control context demarcation via explicit Java API calls. Context are usually implicit. In some cases, however, contexts are demarcated via annotations.

The basic Seam contexts are:

- Stateless context
- Event (i.e., request) context
- Page context
- Conversation context
- Session context
- Business process context
- Application context

You will recognize some of these contexts from servlet and related specifications. However, two of them might be new to you: *conversation context*, and *business process context*. One reason 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 from the point of view of the business logic. A user login session, for example, is a fairly arbitrary construct in terms of the actual application work flow. Therefore, most Seam components are scoped to the conversation or business process contexts, since they are the contexts which are most meaningful in terms of the application.

Let's look at each context in turn.

5.1.1. Stateless context

Components which are truly stateless (stateless session beans, primarily) always live in the stateless context (which is basically the absence of a context since the instance Seam resolves is not stored). Stateless components are not very interesting, and are arguably not very object-

oriented. Nevertheless, they do get developed and used and are thus an important part of any Seam application.

5.1.2. Event context

The event context is the "narrowest" stateful context, and is a generalization of the notion of the web request context to cover other kinds of events. Nevertheless, the event context associated with the lifecycle 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 lifecycle of the request.

When you invoke a Seam component via 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 actually rendering the page, and then have access to it from any event that originates from that page. This is especially useful for functionality like 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 truly central concept in Seam. A *conversation* is a unit of work from the point of view of the user. It might span several interactions with the user, several requests, and several database transactions. But to the user, a conversation solves a single problem. For example, "book hotel", "approve contract", "create order" are all conversations. You might like to think of a conversation implementing a single "use case" or "user story", but the relationship is not necessarily quite exact.

A conversation holds state associated with "what the user is doing now, in this window". A single user may have multiple conversations in progress at any point in time, usually in multiple windows. The conversation context allows us to ensure that state from the different conversations does not collide and cause bugs.

It might take you some time to get used to thinking of applications in terms of conversations. But once you get used to it, we think you'll love the notion, and never be able to not think in terms of conversations again!

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

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

Conversations may be *nested*, with one conversation taking place "inside" a wider conversation. This is an advanced feature.

Usually, conversation state is actually held by Seam in the servlet session between requests. Seam implements configurable *conversation timeout*, automatically destroying inactive conversations, and thus ensuring that the state held by a single user login session does not grow without bound if the user abandons conversations.

Seam serializes processing of concurrent requests that take place in the same long-running conversation context, in the same process.

Alternatively, Seam may be configured to keep conversational state in the client browser.

5.1.5. Session context

A session context holds state associated with the user login session. While there are some cases where it is useful to share state between several conversations, we generally frown on the use of session context for holding 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 the long running business process. This state is managed and made persistent by the BPM engine (JBoss jBPM). The business process spans multiple interactions with multiple users, so this state is shared between multiple users, but in a well-defined manner. The current task determines the current business process instance, and the lifecycle of the business process is defined externally using a *process definition language*, so there are no special annotations for business process demarcation.

5.1.7. Application context

The application context is the familiar servlet context from the servlet spec. Application context is mainly useful for holding 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, a set of *context variables*. These work much the same as session or request attributes in the servlet spec. You may bind any value you like to a context variable, but usually we bind Seam component instances to context variables.

So, within a context, a component instance is identified by the context variable name (this is usually, but not always, the same as the component name). You may programmatically access a named component instance in a particular scope via 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);
```

Usually, however, we obtain components from a context via injection, and put component instances into a context via outjection.

5.1.9. Context search priority

Sometimes, as above, component instances are obtained from a particular known scope. Other times, all stateful scopes are searched, in *priority order*. The order is as follows:

- 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 any facilities for managing concurrent requests originating from the same client. The servlet container simply lets all threads run concurrently and leaves enforcing thread safeness to application code. The EJB container allows stateless components to be accessed concurrently, and throws an exception if multiple threads access a stateful session bean.

This behavior might have been okay in old-style web applications which were based around fine-grained, synchronous requests. But for modern applications which make heavy use of many fine-grained, asynchronous (AJAX) requests, concurrency is a fact of life, and must be supported by the programming model. Seam weaves a concurrency management layer into its context model.

The Seam session and application contexts are multithreaded. Seam will allow concurrent requests in a context to be processed concurrently. The event and page contexts are by nature

single threaded. The business process context is strictly speaking multi-threaded, but in practice concurrency is sufficiently rare that this fact may be disregarded most of the time. Finally, Seam enforces a *single thread per conversation per process* model for the conversation context by serializing concurrent requests in the same long-running conversation context.

Since the session context is multithreaded, and often contains volatile state, session scope components are always protected by Seam from concurrent access so long as the Seam interceptors are not disabled for that component. If interceptors are disabled, then any thread-safety that is required must be implemented by the component itself. Seam serializes requests to session scope JavaBeans by default (and detects and breaks any deadlocks that occur). This is not the default behaviour for application scoped components however, since application scoped components do not usually hold volatile state and because synchronization at the global level is *extremely* expensive. However, you can force a serialized threading model on any JavaBean component by adding the `@Synchronized` annotation.

Note

Seam 2.3 removed the serialization of Stateful session beans by Seam synchronization interceptor because stateful session beans are serialized by EJB 3.1 container by default .

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.

Warning

Be warned that Statefull session Beans are not serialized by Seam anymore. Serialization of Statefull session beans are controlled by EJB container, so there is no need for Seam to duplicate that. So @Synchronized annotation is ignored on Statefull session beans.

5.2. Seam components

Seam components are POJOs (Plain Old Java Objects). In particular, they are JavaBeans or EJB 3.0 enterprise beans. While Seam does not require that components be EJBs and can even be used without an EJB 3.0 compliant container, Seam was designed with EJB 3.0 in mind and includes deep integration with EJB 3.0. Seam supports the following *component types*.

- EJB 3.0 stateless session beans
- EJB 3.0 stateful session beans
- EJB 3.0 entity beans (i.e., JPA entity classes)

- JavaBeans
- EJB 3.0 message-driven beans
- Spring beans (see *Chapter 28, Spring Framework integration*)

5.2.1. Stateless session beans

Stateless session bean components are not able to hold state across multiple invocations. Therefore, they usually work by operating upon the state of other components in the various Seam contexts. They may be used as JSF action listeners, but cannot provide properties to JSF components for display.

Stateless session beans always live in the stateless context.

Stateless session beans can be accessed concurrently as a new instance is used for each request. Assigning the instance to the request is the responsibility of the EJB3 container (normally instances will be allocated from a reusable pool meaning that you may find any instance variables contain data from previous uses of the bean).

Stateless session beans are the least interesting kind of Seam component.

Seam stateless session bean components may be instantiated using `Component.getInstance()` or `@In(create=true)`. They should not be directly instantiated via JNDI lookup or the `new` operator.

5.2.2. Stateful session beans

Stateful session bean components are able to hold state not only across multiple invocations of the bean, but also across multiple requests. Application state that does not belong in the database should usually be held by stateful session beans. This is a major difference between Seam and many other web application frameworks. Instead of sticking information about the current conversation directly in the `HttpSession`, you should keep it in instance variables of a stateful session bean that is bound to the conversation context. This allows Seam to manage the lifecycle of this state for you, and ensure that there are no collisions between state relating to different concurrent conversations.

Stateful session beans are often used as JSF action listener, and as backing beans that 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 stateless contexts.

Concurrent requests to session-scoped stateful session beans are not serialized by Seam as long as EJB 3.1 has changed that. This is a difference in comparison to previous Seam 2.2.x.

Seam stateful session bean components may be instantiated using `Component.getInstance()` or `@In(create=true)`. They should not be directly instantiated via JNDI lookup or the `new` operator.

5.2.3. Entity beans

Entity beans may be bound to a context variable and function as a seam component. 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 do often function as backing beans that provide properties to JSF components for display or form submission. In particular, it is common to use an entity as a backing bean, together with a stateless session bean action listener to implement create/update/delete type functionality.

By default, entity beans are bound to the conversation context. They may never be bound to the stateless context.

Note that it in a clustered environment is somewhat less efficient to bind an entity bean directly to a conversation or session scoped Seam context variable than it would be to hold a reference to the entity bean in a stateful session bean. For this reason, not all Seam applications define entity beans to be Seam components.

Seam entity bean components may be instantiated using `Component.getInstance()`, `@In(create=true)` or directly using the `new` operator.

5.2.4. JavaBeans

JavaBeans may be used just like a stateless or stateful session bean. However, they do not provide the functionality of a session bean (declarative transaction demarcation, declarative security, efficient clustered state replication, EJB 3.0 persistence, timeout methods, etc).

In a later chapter, we show you how to use Seam and Hibernate without an EJB container. In this use case, components are JavaBeans instead of session beans. Note, however, that in many application servers it is somewhat 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.

Concurrent requests to session-scoped JavaBeans are always serialized by Seam.

Seam JavaBean components may be instantiated using `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 may function as a seam component. However, message-driven beans are called quite differently to other Seam components - instead of invoking them via the context variable, they listen for messages sent to a JMS queue or topic.

Message-driven beans may not be bound to a Seam context. Nor do they have access to 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

In order to perform its magic (bijection, context demarcation, validation, etc), Seam must intercept component invocations. For JavaBeans, Seam is in full control of instantiation of the component, and no special configuration is needed. For entity beans, interception is not required since bijection and context demarcation are not defined. For session beans, we must register an EJB interceptor for the session bean component. We could use an annotation, as follows:

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

But a much better way is 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 need a name. We can assign a name to a component using the `@Name` annotation:

```
@Name("loginAction")
```

```
@Stateless
public class LoginAction implements Login {
   ...
}
```

This name is the *seam component name* and is not related to any other name defined by the EJB specification. However, seam component names work just like JSF managed bean names and you can think of the two concepts as identical.

`@Name` is not the only way to define a component name, but we always need to specify the name *somewhere*. If we don't, then none of the other Seam annotations will function.

Whenever Seam instantiates a component, it binds the new instance to a variable in the scope configured for the component that matches the component name. This behavior is identical to how JSF managed beans work, except that Seam allows you to configure this mapping using annotations rather than XML. You can also programmatically bind a component to a context variable. This is useful if a particular component serves more than one role in the system. For example, the currently logged in `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. Be careful, though, because through a programmatic assignment, it's possible to overwrite a context variable that has a reference to a Seam component, potentially confusing matters.

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 {
   ...
}
```

We may use the qualified component name 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 of the built-in Seam components have qualified names but can be accessed through their unqualified names due to the namespace import feature of Seam. The `components.xml` file included in the Seam JAR defines the following namespaces.

```
<components xmlns="http://jboss.org/schema/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 those namespaces, in order. You can include additional namespaces in your application's `components.xml` file for application-specific namespaces.

5.2.8. Defining the component scope

We can override the default scope (context) of a component using the `@Scope` annotation. This lets us define what context a component instance is bound to, when it is 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 more than one role in the system. For example, we often have a `User` class which is usually used as a session-scoped component representing the current user but is used in user administration screens as 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* may be bound to multiple context variables, but this lets us do it at the class level, and take advantage of auto-instantiation.)

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

The `@Roles` annotation lets us specify as many additional roles as we like.

```
@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

Like many good frameworks, Seam eats its own dogfood and is implemented mostly as a set of built-in Seam interceptors (see later) and Seam components. This makes it easy for applications to interact with built-in components at runtime or even customize the basic functionality of Seam by replacing the built-in components with custom implementations. The built-in components are defined in the Seam namespace `org.jboss.seam.core` and the Java package of the same name.

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

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

5.3. Bijection

Dependency injection or *inversion of control* is by now a familiar concept to most Java developers. Dependency injection allows a component to obtain a reference to another component by having the container "inject" the other component to a setter method or instance variable. In all dependency injection implementations that we have seen, injection occurs when the component is constructed, and the reference does not subsequently change for the lifetime of the component instance. For stateless components, this is reasonable. From the point of view of a client, all instances of a particular stateless component are interchangeable. On the other hand, Seam emphasizes the use of stateful components. So traditional dependency injection is no longer a very useful construct. 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 may even have a reference to a component from a "narrower" context)
- *bidirectional* - values are injected from context variables into attributes of the component being invoked, and also *outjected* from the component attributes back out to the context, allowing the component being invoked to manipulate the values of contextual variables 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. Of course, we use annotations 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 will do a priority search of all contexts, using the name of the property or instance variable that is 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 when 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, it can be repetitive to have to specify `@In(create=true)` everywhere they are used. In such cases, you can annotate the component `@AutoCreate`, and then it will always be created, whenever needed, 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 (i.e., set to `null`) immediately after method completion and outjection.

(There is much more information about component lifecycle and injection 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. Lifecycle methods

Session bean and entity bean Seam components support all the usual EJB 3.0 lifecycle callback (`@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 lifecycle 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, a related annotation is the `@Startup` annotation, which may 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 startup components by specifying `@Startup(depends={...})`.

5.5. Conditional installation

The `@Install` annotation lets you control conditional installation of components that are required in some deployment scenarios and not in others. This is useful if:

- You want to mock out some infrastructural component in tests.
- You want change the implementation of a component in certain deployment scenarios.
- You want to install some components only if their dependencies are available (useful for framework authors).

`@Install` works by letting 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. There are some predefined precedence values (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, we don't have a JMS queue available, 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.

This is nice if we are able to control exactly which classes are in the classpath. But if I'm writing a reusable framework with many dependencies, I don't want to have to break that framework across many jars. I want to be able to decide which components to install depending upon what other components are installed, and upon what classes are available in the classpath. The `@Install` annotation also controls this functionality. Seam uses this mechanism internally to

enable conditional installation of many of the built-in components. However, you probably won't need to use it in your application.

5.6. Logging

Who is not totally fed up with seeing noisy code like this?

```
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);
}
```

It is difficult to imagine how the code for a simple log message could possibly be more verbose. There is more lines of code tied up in logging than in the actual business logic! I remain totally astonished that the Java community has not come up with anything better in 10 years.

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);
}
```

It doesn't matter if you declare the `log` variable static or not — it will work either way, except for entity bean components which require the `log` variable to be static.

Note that we don't need the noisy `if ( log.isDebugEnabled() )` guard, since string concatenation happens *inside* the `debug()` method. Note also that we don't usually need to specify the log category explicitly, since Seam knows what component it is injecting the `Log` into.

If `User` and `Product` are Seam components available in the current contexts, it gets even better:

```
@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 automagically chooses whether to send output to log4j or JDK logging. If log4j is in the classpath, Seam with use it. If it is not, Seam will use JDK logging.

5.7. The `Mutable` interface and `@ReadOnly`

Many application servers feature an amazingly broken implementation of `HttpSession` clustering, where changes to the state of mutable objects bound to the session are only replicated when the application calls `setAttribute()` explicitly. This is a source of bugs that can not effectively be tested for at development time, since they will only manifest when failover occurs. Furthermore, the actual replication message contains the entire serialized object graph bound to the session attribute, which is inefficient.

Of course, EJB stateful session beans must perform automatic dirty checking and replication of mutable state and a sophisticated EJB container can introduce optimizations such as attribute-level replication. Unfortunately, not all Seam users have the good fortune to be working in an environment that supports EJB 3.0. So, for session and conversation scoped JavaBean and entity bean components, Seam provides an extra layer of cluster-safe state management over the top of the web container session clustering.

For session or conversation scoped JavaBean components, Seam automatically forces replication to occur by calling `setAttribute()` once in every request that the component was invoked by the application. Of course, this strategy is inefficient for read-mostly components. You can 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 to occur by calling `setAttribute()` once in every request, *unless the (conversation-scoped) entity is currently associated with a Seam-managed persistence context, in which case no replication is needed*. 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 provides a great example of managing an entity bean instance using a Seam component.

5.8. Factory and manager components

We often need to work with objects that are not Seam components. But we still want to be able to inject them into our components using `@In` and use them in value and method binding expressions, etc. Sometimes, we even need to tie them into the Seam context lifecycle (`@Destroy`, for example). So the Seam contexts can contain objects which are not Seam components, and Seam provides a couple of nice features that make it easier to work 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. We define factory methods using 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 both cases, the factory method is called when we reference the `customerList` context variable and its value is null, and then has no further part to play in the lifecycle of the value. An even more powerful pattern is the *manager component pattern*. In this case, we have a Seam component that is bound to a context variable, that 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 if we have an object where you need more control over the lifecycle of the component. 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 which change the underlying object. The component manages these actions itself, and because the object is unwrapped on every access, a consistent view is provided.

Configuring Seam components

The philosophy of minimizing XML-based configuration is extremely strong in Seam. Nevertheless, there are various reasons why we might want to configure a Seam component using XML: to isolate deployment-specific information from the Java code, to enable the creation of re-usable frameworks, to configure Seam's built-in functionality, etc. Seam provides two basic approaches to configuring components: configuration via property settings in a properties file or in `web.xml`, and configuration via `components.xml`.

6.1. Configuring components via property settings

Seam components may be provided with configuration properties either via servlet context parameters, via system properties, or via a properties file named `seam.properties` in the root of the classpath.

The configurable Seam component must expose JavaBeans-style property setter methods for the configurable attributes. If a Seam component named `com.jboss.myapp.settings` has a setter method named `setLocale()`, we can provide 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 startup, or as a servlet context parameter, and Seam will set the value of the `locale` attribute whenever it instantiates the component.

The same mechanism is used to configure Seam itself. For example, to set the 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 a bit more powerful than property settings. It lets you:

- Configure components that have been installed automatically — including both built-in components, and application components that have been annotated with the `@Name` annotation and picked up by Seam's deployment scanner.
- Install classes with no `@Name` annotation as Seam components — this is most useful for certain kinds of 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 may appear in one of three different places:

- The `WEB-INF` directory of a `war`.
- The `META-INF` directory of a `jar`.
- Any directory of a `jar` that contains classes with an `@Name` annotation.

Usually, Seam components are installed when the deployment scanner discovers a class with a `@Name` annotation sitting in an archive with a `seam.properties` file or a `META-INF/components.xml` file. (Unless the component has an `@Install` annotation indicating it should not be installed by default.) The `components.xml` file lets us handle special cases where we need to override the annotations.

For example, the following `components.xml` file installs jBPM:

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:bpm="http://jboss.org/schema/seam/bpm">
   <bpm:jbpm/>
</components>
```

This example does the same thing:

```
<components>
   <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

This one installs and configures two different Seam-managed persistence contexts:

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:persistence="http://jboss.org/schema/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>
```

As does this one:

```
<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>
```

This example creates a session-scoped Seam-managed persistence context (this is not recommended in practice):

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:persistence="http://jboss.org/schema/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>
```

It is common to use the `auto-create` option for infrastructural objects like persistence contexts, which saves you from having to explicitly specify `create=true` when you use the `@In` annotation.

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:persistence="http://jboss.org/schema/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 lets you specify a value or method binding expression that will be evaluated to 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>
```

It is especially common to see the use of `auto-create="true"` with the `<factory>` declaration:

```
<components>

    <factory name="session" value="#{entityManager.delegate}" scope="STATELESS" auto-
create="true"/>

</components>
```

Sometimes we want to reuse the same `components.xml` file with minor changes during both deployment and testing. Seam lets you place wildcards of the form `@wildcard@` in the `components.xml` file which can be replaced either by your Ant build script (at deployment time) or by providing a file named `components.properties` in the classpath (at development time). You'll see this approach used in the Seam examples.

6.3. Fine-grained configuration files

If you have a large number of components that need to be configured in XML, it makes much more sense to split up the information in `components.xml` into many small files. Seam lets you put configuration for a class named, for example, `com.helloworld.Hello` in a resource named `com/helloworld/Hello.component.xml`. (You might be familiar with this pattern, since it is the same one we use in Hibernate.) The root element of the file may be either a `<components>` or `<component>` element.

The first option 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>
```

The second option only lets you define or configure one component, but is less noisy:

```
<component name="hello">
```

```
    <property name="name">#{user.name}</property>
</component>
```

In the second option, the class name 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 may be configured just as you would expect:

```
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, by default, Seam will preserve the order in which you place the attributes in `components.xml` (unless you use a `SortedSet`/`SortedMap` then Seam will use `TreeMap`/`TreeSet`). If the property has a concrete type (for example `LinkedList`) 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 may wire together components using a value-binding expression. Note that this is quite different to injection using `@In`, since it happens at component instantiation time instead of invocation time. It is therefore much more similar to the dependency injection facilities offered by traditional IoC containers like 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 an EL expression string prior to assigning the initial value to the bean property of the component. So you can inject some contextual data into your 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 type of the property to which the initial value is being assigned is either a Seam `ValueExpression` or `MethodExpression`, then the evaluation of the EL is deferred. Instead, the appropriate expression wrapper is created and assigned to the property. The message templates on the Home component from the Seam Application Framework serve as an example.

```
<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."/>
```

Inside the component, you can access the expression string by calling `getExpressionString()` on the `ValueExpression` or `MethodExpression`. If the property is a `ValueExpression`, you can resolve the value using `getValue()` and if the property is a `MethodExpression`, you can invoke the method using `invoke(Object args...)`. Obviously, to assign a value to a `MethodExpression` property, the entire initial value must be a single EL expression.

6.5. Using XML Namespaces

Throughout the examples, there have been two competing ways of declaring components: with and without the use of XML namespaces. The following shows a typical `components.xml` file without namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.org/schema/seam/components"
      xsi:schemaLocation="http://jboss.org/schema/seam/components http://jboss.org/schema/
seam/components-2.3.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 is somewhat verbose. Even worse, the component and attribute names cannot be validated at development time.

The version with using namespaces looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:core="http://jboss.org/schema/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
               "http://jboss.org/schema/seam/core http://jboss.org/schema/seam/core-2.3.xsd
                            http://jboss.org/schema/seam/components  http://jboss.org/schema/seam/
components-2.3.xsd">

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

</components>
```

Even though the schema declarations are verbose, the actual XML content is lean and easy to understand. The schemas provide detailed information about each component and the attributes available, allowing XML editors to offer intelligent autocomplete. The use of namespaced elements makes generating and maintaining correct `components.xml` files much simpler.

Now, this works great for the built-in Seam components, but what about user components? There are two options. First, Seam supports mixing the two models, allowing the use of the generic `<component>` declarations for user components, along with namespaced declarations for built-in components. But even better, Seam allows you to quickly declare namespaces for your own components.

Any Java package can be associated with an XML namespace by annotating the package with the `@Namespace` annotation. (Package-level annotations are declared in a file named `package-info.java` in the package directory.) Here is an example from the seampay demo:

```
@Namespace(value="http://jboss.org/schema/seam/examples/seampay")
package org.jboss.seam.example.seampay;

import org.jboss.seam.annotations.Namespace;
```

That is all you need to do to use the namespaced style in `components.xml`! Now we can write:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:pay="http://jboss.org/schema/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.org/schema/seam/components"
            xmlns:pay="http://jboss.org/schema/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>
```

These examples illustrate the two usage models of a namespaced element. In the first declaration, the `<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 form 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
{
   ...
}
```

If we want validation and autocompletion to work for user-defined components, we will need a schema. Seam does not yet provide a mechanism to automatically generate a schema for a set of components, so it is necessary to generate one manually. The schema definitions for the standard Seam packages can be used for guidance.

The following are the namespaces used by Seam:

- components — `http://jboss.org/schema/seam/components`
- core — `http://jboss.org/schema/seam/core`
- drools — `http://jboss.org/schema/seam/drools`
- framework — `http://jboss.org/schema/seam/framework`
- jms — `http://jboss.org/schema/seam/jms`
- remoting — `http://jboss.org/schema/seam/remoting`
- theme — `http://jboss.org/schema/seam/theme`

- security — `http://jboss.org/schema/seam/security`
- mail — `http://jboss.org/schema/seam/mail`
- web — `http://jboss.org/schema/seam/web`
- pdf — `http://jboss.org/schema/seam/pdf`
- spring — `http://jboss.org/schema/seam/spring`

Events, interceptors and exception handling

Complementing the contextual component model, there are two further basic concepts that facilitate the extreme loose-coupling that is the distinctive feature of Seam applications. The first is a strong event model where events may be mapped to event listeners via JSF-like method binding expressions. The second is the pervasive use of annotations and interceptors to apply cross-cutting concerns to components which 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. Events in Seam come in several types, most of which we have already seen:

- JSF events
- jBPM transition events
- Seam page actions
- Seam component-driven events
- Seam contextual events

All of these various kinds of events are mapped to Seam components via 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.xhtml">
   <transition to="hello">
      <action expression="#{helloWorld.sayHello}"/>
   </transition>
</start-page>
```

You can find out more information about JSF events and jBPM events elsewhere. Let's concentrate for now upon the two additional kinds of events defined by Seam.

7.2. Page actions

A Seam page action is an event that occurs just before we render a page. We declare page actions in `WEB-INF/pages.xml`. We can define a page action for either a particular JSF view id:

```
<pages>
  <page view-id="/hello.xhtml" 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 the pattern:

```
<pages>
  <page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
</pages>
```

Keep in mind that if the `<page>` element is defined in a fine-grained page descriptor, the `view-id` attribute can be left off since it is 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 non-null, Seam will use the defined navigation rules to navigate to a view.

Furthermore, the view id mentioned in the `<page>` element need not correspond to a real JSP or Facelets page! So, we can reproduce the functionality of a traditional action-oriented framework like Struts or WebWork using page actions. This is quite useful if you want to do complex things in response to non-faces requests (for example, HTTP GET requests).

Multiple or conditional page actions my be specified using the `<action>` tag:

```
<pages>
  <page view-id="/hello.xhtml">
    <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 are using the page action to load data, this operation may conflict with the standard JSF

action(s) being executed on a postback. One way to disable the page action is to setup a condition that resolves to true only on 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()`.

To save you from the verbosity of JSF's API, Seam offers a built-in condition that allows you to accomplish the same result with a heck of a lot less typing. You can disable a page action on postback by simply setting the `on-postback` to `false`:

```
<pages>
  <page view-id="/dashboard.xhtml">
    <action execute="#{dashboard.loadData}" on-postback="false"/>
  </page>
</pages>
```

For backwards compatibility reasons, the default value of the `on-postback` attribute is true, though likely you will end up using the opposite setting more often.

7.3. Page parameters

A JSF faces request (a form submission) encapsulates both an "action" (a method binding) and "parameters" (input value bindings). A page action might also needs parameters!

Since GET requests are bookmarkable, page parameters are passed as human-readable request parameters. (Unlike JSF form inputs, which are anything but!)

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.xhtml" action="#{helloWorld.sayHello}">
      <param name="firstName" value="#{person.firstName}"/>
      <param name="lastName" value="#{person.lastName}"/>
    </page>
  </pages>
```

The `<param>` declaration is bidirectional, just like a value binding for a JSF input:

- When a non-faces (GET) request for the view id occurs, Seam sets the value of the named request parameter onto the model object, after performing appropriate type conversions.
- Any `<s:link>` or `<s:button>` transparently includes the request parameter. The value of the parameter 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 transparently includes the request parameter. The value of the parameter 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.

The essential idea behind all this is that *however* we get from any other page to `/hello.xhtml` (or from `/hello.xhtml` back to `/hello.xhtml`), the value of the model attribute referred to in the value binding is "remembered", without the need for a conversation (or other server-side state).

7.4. Propagating request parameters

If just the `name` attribute is specified then the request parameter is propagated using the `PAGE` context (it isn't mapped to model property).

```
<pages>
    <page view-id="/hello.xhtml" action="#{helloWorld.sayHello}">
      <param name="firstName" />
      <param name="lastName" />
    </page>
  </pages>
```

Propagation of page parameters is especially useful if you want to build multi-layer master-detail CRUD pages. You can use it to "remember" which view you were previously on (e.g. 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.

This all sounds pretty complex, and you're probably wondering if such an exotic construct is really worth the effort. Actually, the idea is very natural once you "get it". It is definitely worth taking the time to understand this stuff. Page parameters are the most elegant way to propagate state across a non-faces request. They are especially cool for problems like search screens with bookmarkable results pages, where we would like to be able to write our application code to handle both POST and GET requests with the same code. Page parameters eliminate repetitive listing of request parameters in the view definition and make redirects much easier to code.

7.5. URL rewriting with page parameters

Rewriting occurs based on rewrite patterns found for views in `pages.xml`. Seam URL rewriting does both incoming and outgoing URL rewriting based on the same pattern. Here's a simple pattern:

```
<page view-id="/home.xhtml">
   <rewrite pattern="/home" />
</page>
```

In this case, any incoming request for `/home` will be sent to `/home.xhtml`. More interestingly, 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 these query paramters 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 using URL rewriting, allowing for another option for hiding 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.

URL rewriting requires the Seam rewrite filter to be enable. Rewrite filter configuration is discussed in *Section 31.1.3.3, "URL rewriting"*.

7.6. Conversion and Validation

You can specify a JSF converter for complex model properties:

```
<pages>
  <page view-id="/calculator.xhtml" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>

<paramname="op"converterId="com.my.calculator.OperatorConverter"value="#{calculator.op}"/
>
  </page>
</pages>
```

Alternatively:

```
<pages>
  <page view-id="/calculator.xhtml" 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:

```
<pages>
   <page view-id="/blog.xhtml">
      <param name="date"
           value="#{blog.date}"
           validatorId="com.my.blog.PastDate"
           required="true"/>
   </page>
</pages>
```

Alternatively:

```
<pages>
   <page view-id="/blog.xhtml">
      <param name="date"
           value="#{blog.date}"
           validator="#{pastDateValidator}"
           required="true"/>
   </page>
</pages>
```

Even better, 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, JSF navigation rules have a number of annoying limitations:

- It is not possible to specify request parameters to be 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.

A further problem is that "orchestration" logic gets scattered between `pages.xml` and `faces-config.xml`. It's better to unify this logic into `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>
```

But it would be even nicer if we didn't have to pollute our `DocumentEditor` component with string-valued return values (the JSF outcomes). So 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 to be used by the subsequent rules. The second approach ignores the outcome and evaluates a value binding for each possible rule.

Of course, when an update succeeds, we probably want to end the current conversation. We can do that like this:

```
<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>
```

As we've ended conversation any subsequent requests won't 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. The null outcome is interpreted to mean "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>
```

If you want to perform navigation when a null outcome occurs, use the following form instead:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <render view-id="/viewDocument.xhtml"/>
    </navigation>

</page>
```

The view-id may 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 definition of navigation, page actions and parameters

If you have a lot of different page actions and page parameters, or even just a lot of navigation rules, you will almost certainly want to split the declarations up over multiple files. You can define actions and parameters for a page with the view id `/calc/calculator.xhtml` in a resource named `calc/calculator.page.xml`. The root element in this case is the `<page>` 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 can interact by simply calling each others methods. Stateful components may even implement the observer/observable pattern. But to enable components to interact in a more loosely-coupled fashion than is possible when the components call each others methods directly, 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>
```

Where the *event type* is just an arbitrary string.

When an event occurs, the actions registered for that event will be called in the order they appear in `components.xml`. How does a component raise an event? Seam provides a built-in component for this.

```
@Name("helloWorld")
public class HelloWorld {
   public void sayHello() {
      FacesMessages.instance().add("Hello World!");
```

```
        Events.instance().raiseEvent("hello");
    }
}
```

Or you can use an annotation.

```
@Name("helloWorld")
public class HelloWorld {
    @RaiseEvent("hello")
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
    }
}
```

Notice that this event producer has no dependency upon event consumers. The event listener may 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 in `components.xml` above takes care of mapping the event to the consumer. If you don't like futzing about in the `components.xml` file, you can use an annotation instead:

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

You might wonder why I've not mentioned anything about event objects in this discussion. In Seam, there is no need for an event object to propagate state between event producer and listener. State is held in the Seam contexts, and is shared between components. However, if you really want to pass an event object, you can:

```
@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 can use to perform special kinds of framework integration. The events are:

- `org.jboss.seam.validationFailed` — called when JSF validation fails
- `org.jboss.seam.noConversation` — called when there is no long 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 a long-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 pageflow begins
- `org.jboss.seam.beginPageflow.<name>` — called when the pageflow <name> begins
- `org.jboss.seam.endPageflow` — called when a pageflow ends
- `org.jboss.seam.endPageflow.<name>` — called when the pageflow <name> ends
- `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 which manages an entity called `<name>`

- `org.jboss.seam.security.loggedOut` — called when a user logs out
- `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 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 may observe any of these events in just the same way they observe any other component-driven events.

7.11. Seam interceptors

EJB 3.0 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 which acts as an action listener, we must annotate the session bean `@Interceptors(LoggedInInterceptor.class)`. This is a somewhat ugly annotation. 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 our example, we would create an `@LoggedIn` annotation, as follows:

```
@Target(TYPE)
@Retention(RUNTIME)
@Interceptors(LoggedInInterceptor.class)
public @interface LoggedIn {}
```

We can now simply 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() { ... }

}
```

If interceptor ordering is important (it usually is), you can add `@Interceptor` annotations to your interceptor classes to specify a partial 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, that runs around any of the built-in functionality of EJB3:

```
@Interceptor(type=CLIENT)
public class LoggedInInterceptor
{
   ...
}
```

EJB interceptors are stateful, with a lifecycle that is the same as the component they intercept. For interceptors which do not need to maintain state, Seam lets you get a performance optimization by specifying `@Interceptor(stateless=true)`.

Much of the functionality of Seam is implemented as a set of built-in Seam interceptors, including the interceptors named in the previous example. You don't have to explicitly specify these interceptors by annotating your components; they exist for all interceptable Seam components.

You can even use Seam interceptors with JavaBean components, not just EJB3 beans!

EJB defines interception not only for business methods (using `@AroundInvoke`), but also for the lifecycle methods `@PostConstruct`, `@PreDestroy`, `@PrePassivate` and `@PostActive`. Seam supports all these lifecycle 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 is surprisingly limited when it comes to exception handling. As a partial workaround for this problem, Seam lets you define how a particular class of exception is to be treated by annotating the exception class, or declaring the exception class in an XML file. This facility is meant to be combined with the EJB 3.0-standard `@ApplicationException` annotation which specifies whether the exception should cause a transaction rollback.

7.12.1. Exceptions and transactions

EJB specifies well-defined rules that let us control whether an exception immediately marks the current transaction for rollback when it is 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 do 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 that there is a difference between marking a transaction for rollback, and actually rolling it back. The exception rules say that the transaction should be marked rollback only, but it may still be active after the exception is thrown.

Seam applies the EJB 3.0 exception rollback rules also to Seam JavaBean components.

But these rules only apply in the Seam component layer. What about an exception that is uncaught and propagates out of the Seam component layer, and out of the JSF layer? Well, it is always wrong to leave a dangling transaction open, so Seam rolls back any active transaction when an exception occurs and is uncaught in the Seam component layer.

7.12.2. Enabling Seam exception handling

To enable Seam's exception handling, we need to make sure we have the master servlet filter 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>
```

As the second requirement is to add `web:exception-filter` configuration component into `WEB-INF/components.xml`. More details are in *Section 31.1.3.1, "Exception handling"*

You need to disable Facelets development mode in `web.xml` too and Seam debug mode in `components.xml` if you want your exception handlers to fire.

7.12.3. Using annotations for exception handling

The following exception results in a HTTP 404 error whenever it propagates out of 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 out of 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

It is important to note that Seam cannot handle exceptions that occur during JSF's RENDER_RESPONSE phase, as it is not possible to perform a redirect once the response has started being written to.

You can also use EL to specify the `viewId` to redirect to.

This exception results in a redirect, along with a message to the user, when it propagates out of the Seam component layer. 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 we can't add annotations to all the exception classes we are interested in, 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 last `<exception>` declaration does not specify a class, and is a catch-all for any exception for which handling is not otherwise specified 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, e.g. to access the message of the exception:

```
...
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 actually 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 logging level at which the exception will be logged, or to even suppress the exception being logged altogether. The attributes `log` and `log-level` can be used to control exception logging. By setting `log="false"` as per the following example, then no log message will be generated when the specified exception occurs:

```
   <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` (i.e. the exception will be logged). Alternatively, you can specify the `log-level` to control at which log level 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` or `trace`. If the `log-level` is not specified, or if an invalid value is configured, then it 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 don't 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 if the user posts back to a page once 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"*, give you finer-grained control over session expiration if you are accessing a page used within a conversation.

Conversations and workspace management

It's time to understand Seam's conversation model in more detail.

Historically, the notion of a Seam "conversation" came about as a merger of three different ideas:

- The idea of a *workspace*, which I encountered in a project for the Victorian government in 2002. In this project I was forced to implement workspace management on top of Struts, an experience I pray never to repeat.
- The idea of an *application transaction* with optimistic semantics, and the realization that existing frameworks based around a stateless architecture could not provide effective management of extended persistence contexts. (The Hibernate team is truly fed up with copping the blame for `LazyInitializationExceptions`, which are not really Hibernate's fault, but rather the fault of the extremely limiting persistence context model supported by stateless architectures such as the Spring framework or the traditional *stateless session facade* (anti)pattern in J2EE.)
- The idea of a workflow *task*.

By unifying these ideas and providing deep support in the framework, we have a powerful construct that lets us build richer and more efficient applications with less code than before.

8.1. Seam's conversation model

The examples we have seen so far make use of a very simple conversation model that follows these rules:

- There is always a conversation context active during the apply request values, process validations, update model values, invoke application and render response phases of the JSF request lifecycle.
- At the end of the restore view phase of the JSF request lifecycle, Seam attempts to restore any previous long-running conversation context. If none exists, Seam creates a new temporary conversation context.
- When an `@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 lifecycle, Seam 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, but see below for more information on this.
- If the JSF request lifecycle 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. If you don't do anything special, 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.

If you want to propagate a Seam conversation across a non-faces request, you need to explicitly code the Seam *conversation id* as a request parameter:

```
<a href="main.jsf?#{manager.conversationIdParameter}=#{conversation.id}">Continue</a>
```

Or, the more JSF-ish:

```
<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>
```

If you wish to disable propagation of the conversation context for a postback, a similar trick is used:

```
<h:commandLink action="main" value="Exit">
   <f:param name="conversationPropagation" value="none"/>
</h:commandLink>
```

If you use the Seam tag library, this is equivalent:

```
<h:commandLink action="main" value="Exit">
   <s:conversationPropagation type="none"/>
</h:commandLink>
```

Note that disabling conversation context propagation is absolutely not the same thing as ending the conversation.

The `conversationPropagation` request parameter, or the `<s:conversationPropagation>` tag may even be used to begin a conversation, end the current conversation, destroy the entire conversation stack, or begin a nested conversation.

```
<h:commandLink action="main" value="Exit">
   <s:conversationPropagation type="end"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Exit">
   <s:conversationPropagation type="endRoot"/>
</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 needed. Some complex applications have either 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 also have access to the state of the outer conversation.
- The user is able to 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)` inside 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 through 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 directly set into the nested conversation’s context do not affect the objects accessible in the parent conversation’s context.
- Injection or a context lookup from the conversation context will first lookup the value in the current conversation context and, if no value is found, will proceed down the conversation stack if the conversation is nested. As you will see in moment, this behavior can be overriden.

When an `@End` is subsequently encountered, the nested conversation will be destroyed, and the outer conversation will resume, by "popping" the conversation stack. Conversations may be nested to any arbitrary depth.

Certain user activity (workspace management, or the back button) can cause the outer conversation to be resumed before the inner conversation is ended. 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 always destroy all of its descendents. You can achieve this declaratively by specifying `@End(root=true)`.

A conversation may 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 exists

in 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 kind of action listener that is triggered when a page is accessed via a non-faces request (for example, a HTTP GET request). This can occur if the user bookmarks the page, or if we navigate to the page via an `<h:outputLink>`.

Sometimes we want to begin a conversation immediately the page is accessed. Since there is no JSF action method, we can't solve the problem in the usual way, by annotating the action with `@Begin`.

A further problem arises if the page needs some state to be fetched into a context variable. We've already seen two ways to solve this problem. If that 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 none of these options works for you, Seam lets you define a *page action* in the `pages.xml` file.

```
<pages>
   <page view-id="/messageList.xhtml" action="#{messageManager.list}"/>
   ...
</pages>
```

This action method is called at the beginning of the render response phase, 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, possibly resulting in a completely different page being rendered.

If *all* you want to do before rendering the page is begin a conversation, you could use a built-in action method that does just that:

```
<pages>
   <page view-id="/messageList.xhtml" action="#{conversation.begin}"/>
   ...
</pages>
```

Note that you can also call this built-in action from a JSF control, and, similarly, you can use `#{conversation.end}` to end conversations.

If you want more control, to join existing conversations or begin a nested conversion, to begin a pageflow or an atomic conversation, you should use the `<begin-conversation>` element.

```
<pages>
  <page view-id="/messageList.xhtml">
    <begin-conversation nested="true" pageflow="AddItem"/>
  <page>
  ...
</pages>
```

There is also an `<end-conversation>` element.

```
<pages>
  <page view-id="/home.xhtml">
    <end-conversation/>
  <page>
  ...
</pages>
```

To solve the first problem, we now have five options:

- 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 "protect" such a page is to require a long-running conversation as a prerequisite to rendering the page. Fortunately, Seam has a built-in mechanism for enforcing this requirement.

In the Seam page descriptor, you can indicate that the current conversation must be long-running (or nested) in order for a page to be rendered using the `conversation-required` attribute as follows:

```
<page view-id="/book.xhtml" conversation-required="true"/>
```

Note

The only downside is there's no built-in way to indicate *which* long-running conversation is required. You can build on this basic authorization by dually checking if a specific value is present in the conversation within a page action.

When Seam determines that this page is requested outside of a long-running conversation, the following actions are taken:

- A contextual event named `org.jboss.seam.noConversation` is raised
- A warning status message is registered using the bundle key `org.jboss.seam.NoConversation`
- The user is redirected to an alternate page, if defined

The alternate page is defined in the `no-conversation-view-id` attribute on a `<pages>` element in the Seam page descriptor as follows:

```
<pages no-conversation-view-id="/main.xhtml"/>
```

At the moment, you can only define one such page for the entire application.

8.5. Using `<s:link>` and `<s:button>`

JSF command links always perform a form submission via JavaScript, which breaks the web browser's "open in new window" or "open in new tab" feature. In plain JSF, you need to use an `<h:outputLink>` if you need this functionality. But there are two major limitations to `<h:outputLink>`.

- JSF provides no way to attach an action listener to an `<h:outputLink>`.
- JSF does not propagate the selected row of a `DataModel` since there is no actual form submission.

Seam provides the notion of a *page action* to help solve the first problem, but this does nothing to help us with the second problem. We *could* work around this by using the RESTful approach of passing a request parameter and requerying for the selected object on the server side. In some cases — such as the Seam blog example application — this is indeed the best approach. The RESTful style supports bookmarking, since it does not require server-side state. In other cases, where we don't care about bookmarks, the use of `@DataModel` and `@DataModelSelection` is just so convenient and transparent!

To fill in this missing functionality, and to make conversation propagation even simpler to manage, Seam provides the `<s:link>` JSF tag.

The link may specify just the JSF view id:

```
<s:link view="/login.xhtml" value="Login"/>
```

Or, it may specify an action method (in which case the action outcome determines the page that results):

```
<s:link action="#{login.logout}" value="Logout"/>
```

If you specify *both* a JSF view id and an action method, 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` using 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="nested"/>
```

If the link begins a conversation, you can even specify a pageflow to be used:

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
      pageflow="EditDocument"/>
```

The `taskInstance` attribute is for use in jBPM task lists:

```
<s:link action="#{documentApproval.approveOrReject}" taskInstance="#{task}"/>
```

(See the DVD Store demo application for examples of this.)

Finally, if you need the "link" to be rendered as a button, use `<s:button>`:

```
<s:button action="#{login.logout}" value="Logout"/>
```

8.6. Success messages

It is quite common to display a message to the user indicating success or failure of an action. It is convenient to use a JSF `FacesMessage` for this. Unfortunately, a successful action often requires a browser redirect, and JSF does not propagate faces messages across redirects. This makes it quite difficult to display success messages in plain JSF.

The built in conversation-scoped Seam component named `facesMessages` solves this problem. (You must have the Seam redirect filter installed.)

```
@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");
   }
}
```

Any message added to `facesMessages` is used in the very next render response phase for the current conversation. This even works when there is no long-running conversation since Seam preserves even temporary conversation contexts across redirects.

You can even include JSF EL expressions in a faces message summary:

```
facesMessages.add("Document #{document.title} was updated");
```

You may display the messages in the usual way, for example:

```
<h:messages globalOnly="true"/>
```

8.7. Natural conversation ids

When working with conversations that deal with persistent objects, it may be desirable to use the natural business key of the object instead of the standard, "surrogate" conversation id:

Easy redirect to existing conversation

It can be useful to redirect to an existing conversation if the user requests the same operation twice. Take this example: “ You are on ebay, half way through paying for an item you just won as a Christmas present for your parents. Lets say you're sending it straight to them - you enter your payment details but you can't remember their address. You accidentally reuse the same browser window finding out their address. Now you need to return to the payment for the item. ”

With a natural conversation it's really easy to have the user rejoin the existing conversation, and pick up where they left off - just have them to rejoin the payForItem conversation with the itemId as the conversation id.

User friendly URLs

For me this consists of a navigable hierarchy (I can navigate by editing the url) and a meaningful URL (like this Wiki uses - so don't identify things by random ids). For some applications user friendly URLs are less important, of course.

With a natural conversation, when you are building your hotel booking system (or, of course, whatever your app is) you can generate a URL like `http://seam-hotels/book.seam?hotel=BestWesternAntwerpen` (of course, whatever parameter `hotel` maps to on your domain model must be unique) and with URLRewrite easily transform this to http://seam-hotels/book/BestWesternAntwerpen.

Much better!

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 from the above definition is that the conversation has a name, in this case `PlaceBid`. This name uniquely identifies this particular named conversation, and is used by the `page` definition to identify a named conversation to participate in.

The next attribute, `parameter-name` defines the request parameter that will contain the natural conversation id, in place of the default conversation id parameter. In this example, the `parameter-name` is `auctionId`. This means that instead of a conversation parameter like `cid=123` appearing in the URL for your page, it will contain `auctionId=765432` instead.

The last attribute in the above configuration, `parameter-value`, defines an EL expression used 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 which pages will participate 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 starting, or redirecting to, a natural conversation there are a number of options for specifying the natural conversation name. Let's start by looking at 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>
```

From here, we can see that invoking the action `#{bidAction.placeBid}` from our auction view (by the way, all these examples are taken from the seamBay example in Seam), that we will be redirected to `/bid.xhtml`, which, as we saw previously, is configured with the natural conversation `PlaceBid`. The declaration for our action method 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, after the action method has already been invoked. This is a problem when redirecting to an existing conversation, as redirection needs to be occur before the action method is invoked. Therefore it is necessary to specify the conversation name when the action is invoked. One way of doing this is by using the `s:conversationName` tag:

```
<h:commandButton id="placeBidWithAmount" styleClass="placeBid"
action="#{bidAction.placeBid}">
  <s:conversationName value="PlaceBid"/>
</h:commandButton>
```

Another alternative is to specify the `conversationName` attribute when using either `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 makes workspace management completely transparent at the level of the Java code. To enable workspace management, all you need to do is:

- Provide *description* text for each view id (when using JSF or Seam navigation rules) or page node (when using jPDL pageflows). This description text is displayed to the user by the workspace switchers.
- Include one or more of the standard workspace switcher JSF or Facelets fragments in your pages. The standard fragments support workspace management via a drop down menu, a list of conversations, or breadcrumbs.

8.10.1. Workspace management and JSF navigation

When you use JSF or Seam navigation rules, 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` that Seam expects to find in the `WEB-INF` directory, right next to `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 that if this file is missing, the Seam application will continue to work perfectly! The only missing functionality will be the ability to switch workspaces.

8.10.2. Workspace management and jPDL pageflow

When you use a jPDL pageflow definition, Seam switches to a 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 upon 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>
```

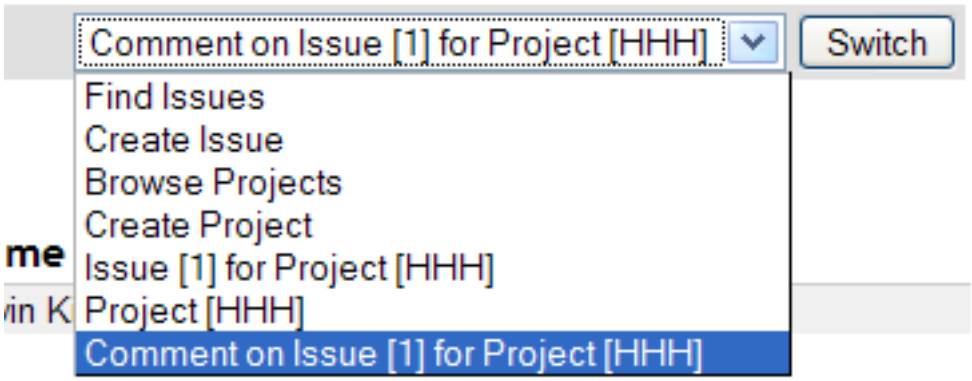
8.10.3. The conversation switcher

Include the following fragment in your JSF page to get a drop-down menu that lets you switch to any current conversation, or to 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"/>
```

In this example, we have a menu that includes an item for each conversation, together with two additional items that let the user begin a new 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 very 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>
```

We imagine that you will want to customize this for your own application.

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.

Notice that the conversation list lets the user destroy workspaces.

8.10.5. Breadcrumbs

Breadcrumbs are useful in applications which use a nested conversation model. The breadcrumbs are a list of links to conversations in the current conversation stack:

```
<ui:repeat value="#{conversationStack}" var="entry">
    <h:outputText value=" | "/>
    <h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</ui:repeat
```

Home | Find Issues | Create Issue | Project [HHH] | Issue [1] for Project [HHH]

Issue Attributes

8.11. Conversational components and JSF component bindings

Conversational components have one minor limitation: they cannot be used to hold bindings to JSF components. (We generally prefer not to use 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.

To work around this use an event scoped component to store the component bindings and inject it into the conversation scoped component that requires it.

```
@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;

   ...
}
```

Also, you can't inject a conversation scoped component into an event scoped component which you bind a JSF control to. This includes Seam built in components like `facesMessages`.

Alternatively, you can access the JSF component tree through the implicit `uiComponent` handle. The following example accesses `getRowIndex()` of the `UIData` component which backs the data table during iteration, it prints the current row number:

```
<h:dataTable id="lineItemTable" var="lineItem" value="#{orderHome.lineItems}">
```

```
  <h:column>
    Row: #{uiComponent['lineItemTable'].rowIndex}
  </h:column>
  ...
</h:dataTable>
```

JSF UI components are available with their client identifier in this map.

8.12. Concurrent calls to conversational components

A general discussion of concurrent calls to Seam components can be found in *Section 5.1.10, "Concurrency model"*. Here we will discuss the most common situation in which you will encounter concurrency — accessing conversational components from AJAX requests. We're going to discuss the options that a Ajax client library should provide to control events originating at the client — and we'll look at the options RichFaces gives you.

Conversational components don't allow real concurrent access therefore Seam queues each request to process them serially. This allows each request to be executed in a deterministic fashion. However, a simple queue isn't that great — firstly, if a method is, for some reason, taking a very long time to complete, running it over and over again whenever the client generates a request is bad idea (potential for Denial of Service attacks), and, secondly, AJAX is often to used to provide a quick status update to the user, so continuing to run the action after a long time isn't useful.

Therefore, when you are working inside a long running conversation, Seam queues the action event for a period of time (the concurrent request timeout); if it can't process the event in time, it creates a temporary conversation and prints out a message to the user to let them know what's going on. It's therefore very important not to flood the server with AJAX events!

We can set a sensible default for the concurrent request timeout (in ms) in components.xml:

```
<core:manager concurrent-request-timeout="500" />
```

We can also fine tune the concurrent request timeout on a page-by-page basis:

```
<page view-id="/book.xhtml"
      conversation-required="true"
      login-required="true"
      concurrent-request-timeout="2000" />
```

So far we've discussed AJAX requests which appear serial to the user - the client tells the server that an event has occur, and then rerenders part of the page based on the result. This approach is great when the AJAX request is lightweight (the methods called are simple e.g. calculating the

sum of a column of numbers). But what if we need to do a complex computation that is going to take a minute?

For heavy computation we should use a poll based approach — the client sends an AJAX request to the server, which causes action to be executed asynchronously on the server (the response to the client is immediate) and the client then polls the server for updates. This is good approach when you have a long-running action for which it is important that every action executes (you don't want some to timeout).

8.12.1. How should we design our conversational AJAX application?

Well first, you need to decide whether you want to use the simpler "serial" request or whether you want to use a polling approach.

If you go for a "serial" requests, then you need to estimate how long your request will take to complete - is it much shorter than the concurrent request timeout? If not, you probably want to alter the concurrent request timeout for this page (as discussed above). You probably want a queue on the client side to prevent flooding the server with requests. If the event occurs often (e.g. a keypress, onblur of input fields) and immediate update of the client is not a priority you should set a request delay on the client side. When working out your request delay, factor in that the event may also be queued on the server side.

Finally, the client library may provide an option to abort unfinished duplicate requests in favor of the most recent.

Using a poll-style design requires less fine-tuning. You just 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 you design your application to queue concurrent requests to your conversational component, there is a risk that the server will become overloaded and be unable to process all the requests before the request will have to wait longer than the `concurrent-request-timeout`. In this case Seam will throw a `ConcurrentRequestTimeoutException` which can be handled in `pages.xml`. We recommend sending an HTTP 503 error:

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException" log-level="trace">
  <http-error error-code="503" />
</exception>
```

503 Service Unavailable (HTTP/1.1 RFC)

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>
```

Seam Remoting and JSF 2 can both handle HTTP error codes. Seam Remoting will pop up a dialog box showing the HTTP error. JSF 2 provides 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">
  jsf.ajax.addOnError(function(data) {
    alert("An error occurred");
  });
</script>
```

JSF 2 javascript documentation

More details about JSF 2 javascript API can be seen at *http://javaserverfaces.java.net/nonav/docs/2.0/jsdocs/symbols/jsf.ajax.html*

If instead of an error code, the server reports that the view has expired, perhaps because the session timed out, you can use a standard *javax.faces.context.ExceptionHandler* [http://docs.oracle.com/javaee/6/api/javax/faces/context/ExceptionHandler.html] to handle this scenario.

Pageflows and business processes

JBoss jBPM is a business process management engine for any Java SE or EE environment. jBPM lets you represent a business process or user interaction as a graph of nodes representing wait states, decisions, tasks, web pages, etc. The graph is defined using a simple, very readable, XML dialect called jPDL, and may be edited and visualised graphically using an eclipse plugin. jPDL is an extensible language, and is suitable for a range of problems, from defining web application page flow, to traditional workflow management, all the way up to orchestration of services in a SOA environment.

Seam applications use jBPM for two different problems:

- Defining the pageflow involved in complex user interactions. A jPDL process definition defines the page flow for a single conversation. A Seam conversation is considered to be a relatively short-running interaction with a single user.
- Defining the overarching business process. The business process may span multiple conversations with multiple users. Its state is persistent in the jBPM database, so it is considered long-running. Coordination of the activities of multiple users is a much more complex problem than scripting an interaction with a single user, so jBPM offers sophisticated facilities for task management and dealing with multiple concurrent paths of execution.

Don't get these two things confused! They operate at very different levels or granularity. *Pageflow*, *conversation* and *task* all refer to a single interaction with a single user. A business process spans many tasks. Futhermore, the two applications of jBPM are totally orthogonal. You can use them together or independently or not at all.

You don't have to know jPDL to use Seam. If you're perfectly happy defining pageflow using JSF or Seam navigation rules, and if your application is more data-driven that process-driven, you probably don't need jBPM. But we're finding that thinking of user interaction in terms of a well-defined graphical representation is helping us build more robust applications.

9.1. Pageflow in Seam

There are two ways to define pageflow in Seam:

- Using JSF or Seam navigation rules - the *stateless navigation model*
- Using jPDL - the *stateful navigation model*

Very simple applications will only need the stateless navigation model. Very complex applications will use both models in different places. Each model has its strengths and weaknesses!

9.1.1. The two navigation models

The stateless model defines a mapping from a set of named, logical outcomes of an event directly to the resulting page of the view. The navigation rules are entirely oblivious to any state held by the

application other than what page was the source of the event. This means that your action listener methods must sometimes make decisions about the page flow, since only they have access to the current state of the application.

Here is an example page flow definition using JSF navigation rules:

```
<navigation-rule>
   <from-view-id>/numberGuess.xhtml</from-view-id>

   <navigation-case>
      <from-outcome>guess</from-outcome>
      <to-view-id>/numberGuess.xhtml</to-view-id>
      <redirect/>
   </navigation-case>

   <navigation-case>
      <from-outcome>win</from-outcome>
      <to-view-id>/win.xhtml</to-view-id>
      <redirect/>
   </navigation-case>

   <navigation-case>
      <from-outcome>lose</from-outcome>
      <to-view-id>/lose.xhtml</to-view-id>
      <redirect/>
   </navigation-case>

</navigation-rule>
```

Here is the same example page flow definition using Seam navigation rules:

```
<page view-id="/numberGuess.xhtml">

   <navigation>
      <rule if-outcome="guess">
         <redirect view-id="/numberGuess.xhtml"/>
      </rule>
      <rule if-outcome="win">
         <redirect view-id="/win.xhtml"/>
      </rule>
      <rule if-outcome="lose">
         <redirect view-id="/lose.xhtml"/>
      </rule>
```

```
    </navigation>

</page>
```

If you find navigation rules overly verbose, you can return view ids directly from your action listener methods:

```
public String guess() {
    if (guess==randomNumber) return "/win.xhtml";
    if (++guessCount==maxGuesses) return "/lose.xhtml";
    return null;
}
```

Note that this results in a redirect. You can even specify parameters to be used in the redirect:

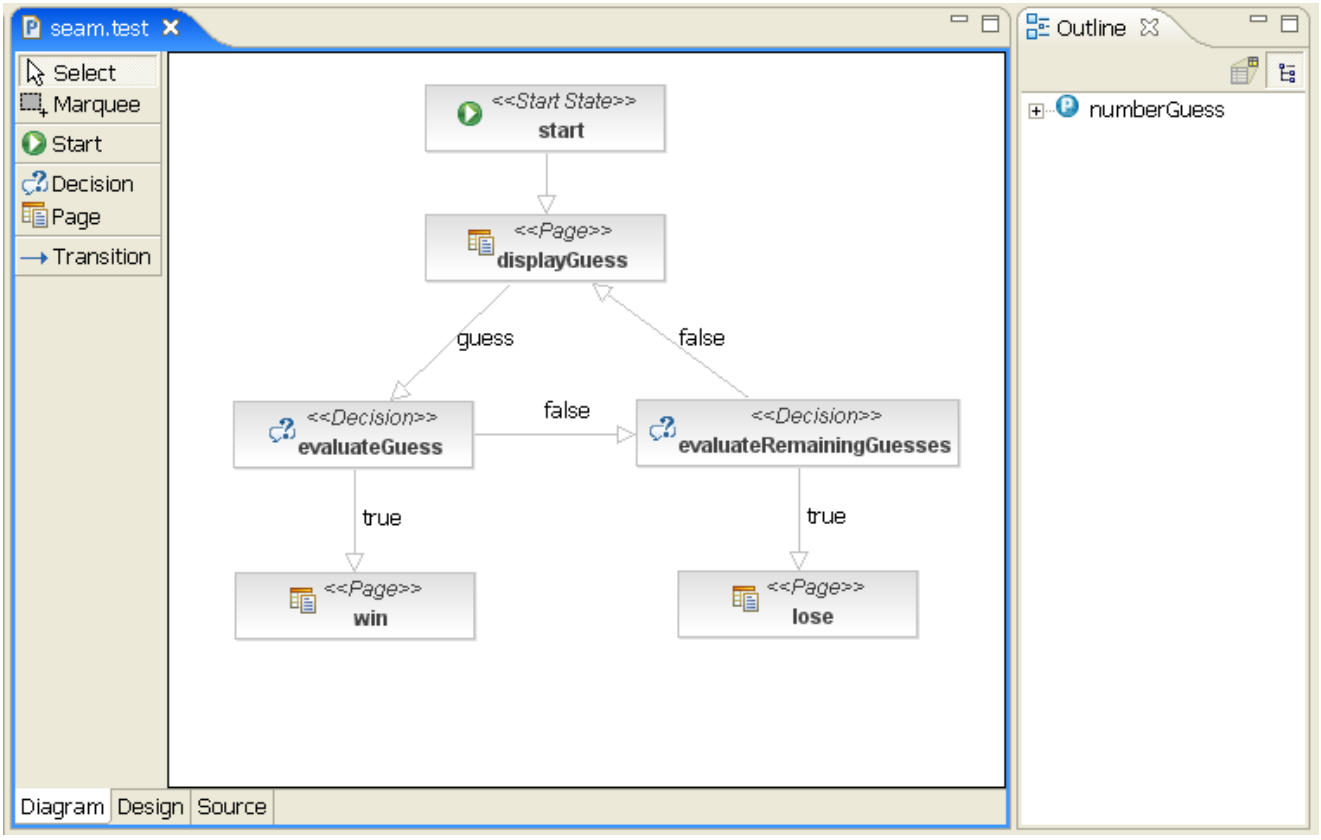
```
public String search() {
    return "/searchResults.xhtml?searchPattern=#{searchAction.searchPattern}";
}
```

The stateful model defines a set of transitions between a set of named, logical application states. In this model, it is possible to 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.xhtml">
    <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.xhtml">
    <redirect/>
    <end-conversation />
  </page>

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

</pageflow-definition>
```



There are two things we notice immediately here:

- The JSF/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 understandable, without us needing to even look at the facelets template or Java code.

In addition, the stateful model is more *constrained*. For each logical state (each step in the page flow), there are a constrained set of possible transitions to other states. The stateless model is an *ad hoc* model which is suitable to relatively unconstrained, freeform navigation where the user decides where he/she wants to go next, not the application.

The stateful/stateless navigation distinction is quite similar to the traditional view of modal/modeless interaction. Now, Seam applications are not usually modal in the simple sense of the word - indeed, avoiding application modal behavior is one of the main reasons for having conversations! However, Seam applications can be, and often are, modal at the level of a particular conversation. It is well-known that modal behavior is something to avoid as much as possible; it is very difficult to predict the order in which your users are going to want to do things! However, there is no doubt that the stateful model has its place.

The biggest contrast between the two models is the back-button behavior.

9.1.2. Seam and the back button

When JSF or Seam navigation rules are used, Seam lets the user freely navigate via the back, forward and refresh buttons. It is the responsibility of the application to ensure that conversational state remains internally consistent when this occurs. Experience with the combination of web application frameworks like Struts or WebWork - that do not support a conversational model - and stateless component models like EJB stateless session beans or the Spring framework has taught many developers that this is close to impossible to do! However, our experience is that in the context of Seam, where there is a well-defined conversational model, backed by stateful session beans, it is actually quite straightforward. Usually it is as simple as combining the use of `no-conversation-view-id` with null checks at the beginning of action listener methods. We consider support for freeform navigation to be almost always desirable.

In this case, the `no-conversation-view-id` declaration goes in `pages.xml`. It tells Seam to redirect to a different page if a request originates from a page rendered during a conversation, and that conversation no longer exists:

```
<page view-id="/checkout.xhtml"
      no-conversation-view-id="/main.xhtml"/>
```

On the other hand, in the stateful model, using 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 not permitted by default 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, and simply redirects the user to the "current" page (and displays a faces message). Whether you consider this a feature or a limitation of the stateful model depends upon your point of view: as an application developer, it is a feature; as a user, it might be frustrating! You can enable backbutton 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, it is not possible to use the back button to return to that page even when back is enabled on a page later in the flow. The reason 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 (i.e., a Faces request). A redirect severs this linkage.

Of course, we still need to 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'll quickly learn to recognize when to prefer one model over the other.

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 which contains 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 using 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 to be already at the page being rendered, 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 which is 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: namely, a post-then-redirect behavior, to overcome problems with the browser's refresh button. (Note that Seam propagates conversation contexts over these browser redirects. So there is no need for a Ruby on Rails style "flash" construct in Seam!)

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 the transition is triggered by clicking this button, jBPM will activate the transition action by calling the `guess()` method of the `numberGuess` component. Notice that the syntax used for specifying actions in the jPDL is just a familiar JSF EL expression, and that the transition action handler is just a method of a Seam component in the current Seam contexts. So we have exactly the same event model for jBPM events that we already have for JSF events! (The *One Kind of Stuff* principle.)

In the case of a null outcome (for example, a command button with no `action` defined), Seam will signal the transition with no name if one exists, or else simply redisplay the page if all transitions have names. So we could slightly simplify our example pageflow and this button:

```
<h:commandButton type="submit" value="Guess"/>
```

Would fire the following un-named transition:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
   <redirect/>
   <transition to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
   </transition>
</page>
```

It is even possible to have the button call an action method, in which case the action outcome will determine the transition to be taken:

```
<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 is considered an inferior style, since it moves responsibility for controlling the flow out of the pageflow definition and back into the other components. It is much better to centralize this concern in the pageflow itself.

9.2.4. Controlling the flow

Usually, we don't need the more powerful features of jPDL when defining pageflows. We do need the `<decision>` node, however:

```
<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 in the Seam contexts.

9.2.5. Ending the flow

We end the conversation using `<end-conversation>` or `@End`. (In fact, for readability, use of *both* is encouraged.)

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

Optionally, we can end a task, specify a jBPM `transition` name. In this case, Seam will signal 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 and have one pageflow pause pause 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.

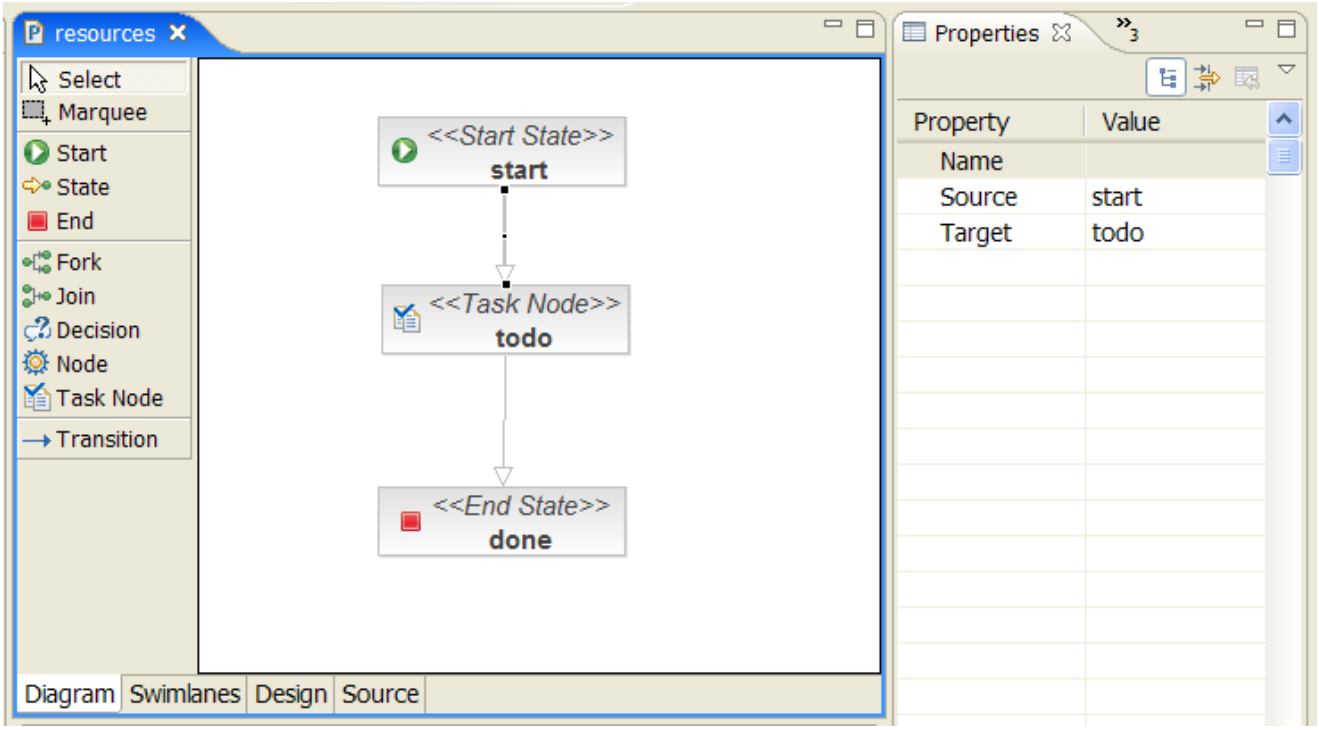
9.3. Business process management in Seam

A business process is a well-defined set of tasks that must be performed by users or software systems according to well-defined rules about *who* can perform a task, and *when* it should be performed. Seam's jBPM integration makes it easy to display lists of tasks to users and let them manage their tasks. Seam also lets the application store state associated with the

business process in the `BUSINESS_PROCESS` context, and have that state made persistent via jBPM variables.

A simple business process definition looks much the same as a page flow definition (*One Kind of Stuff*), except that instead of `<page>` nodes, we have `<task-node>` nodes. In a long-running business process, the wait states are 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>
```



It is perfectly possible that we might have both jPDL business process definitions and jPDL pageflow definitions in the same project. If so, the relationship between the two is that 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

We need to 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>
```

As jBPM processes are persistent across application restarts, when using Seam in a production environment you won't want to install the process definition every time the application starts. Therefore, in a production environment, you'll need to deploy the process to jBPM outside of Seam. In other words, only install process definitions from `components.xml` when developing your application.

9.4.2. Initializing actor ids

We always need to know what user is currently logged in. jBPM "knows" users by their *actor id* and *group actor ids*. We specify the current actor ids using the built in Seam component named `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 hardcode 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 could have used 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 using pages.xml:

```
<page>
   <start-task />
</page>
```

These annotations begin a special kind of conversation that has significance 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 using `@EndTask`, Seam will signal 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 takes over and continues executing the business process definition. (In more complex processes, several tasks might 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.

Seam and Object/Relational Mapping

Seam provides extensive support for the two most popular persistence architectures for Java: Hibernate, and the Java Persistence API 2.0 introduced with EJB 3.1. Seam's unique state-management architecture allows the most sophisticated ORM integration of any web application framework.

10.1. Introduction

Seam grew out of the frustration of the Hibernate team with the statelessness typical of the previous generation of Java application architectures. The state management architecture of Seam was originally designed to solve problems relating to persistence — in particular 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 interesting work involves first displaying data to a user, and then, slightly later, updating the same data. So Hibernate was designed to support the idea of a persistence context which spanned an optimistic transaction.

Unfortunately, the so-called "stateless" architectures that preceded Seam and EJB 3.0 had no construct for representing an optimistic transaction. So, instead, these architectures provided persistence contexts scoped to the atomic transaction. Of course, this resulted in many problems for users, and is the cause of the number one user complaint about Hibernate: the dreaded `LazyInitializationException`. What we need is a construct for representing an optimistic transaction in the application tier.

EJB 3.0 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) however there are two problems:

- The lifecycle of the stateful session bean must be managed manually via code in the web tier (it turns out that this is a subtle problem and much more difficult in practice than it sounds).
- Propagation of the persistence context between stateful components in the same optimistic transaction is possible, but tricky.

Seam solves the first problem by providing conversations, and stateful session bean components scoped to the conversation. (Most conversations actually represent optimistic transactions in the data layer.) This is sufficient for many simple applications (such as the Seam booking demo) where persistence context propagation is not needed. For more complex applications, with many loosely-interacting components in each conversation, propagation of the persistence context across components becomes an important issue. So Seam extends the persistence context management model of EJB 3.0, to provide conversation-scoped extended persistence contexts.

10.2. Seam managed transactions

EJB session beans feature declarative transaction management. The EJB container is able to 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, we can do all the work associated with that action in one transaction, and be sure that it is committed or rolled back when we finish processing the action. This is a great feature, and all that is needed by some Seam applications.

However, there is a problem with this approach. A Seam application may not perform all data access for a request from a single method call to a session bean.

- The request might require processing by several loosely-coupled components, each of which is called independently from the web layer. It is common to see several or even many calls per request from the web layer to EJB components in Seam.
- Rendering of the view might require lazy fetching of associations.

The more transactions per request, the more likely we are to encounter atomicity and isolation problems when our application is processing many concurrent requests. Certainly, all write operations should occur in the same transaction!

Hibernate users developed the *"open session in view"* pattern to work around this problem. In the Hibernate community, "open session in view" was historically even more important because frameworks like Spring use transaction-scoped persistence contexts. So rendering the view would cause `LazyInitializationException`s when unfetched associations were accessed.

This pattern is usually implemented as a single transaction which spans the entire request. There are several problems with this implementation, the most serious being that we can never be sure that a transaction is successful until we commit it — but by the time the "open session in view" transaction is committed, the view is fully rendered, and the rendered response may already have been flushed to the client. How can we notify the user that their transaction was unsuccessful?

Seam solves both the transaction isolation problem and the association fetching problem, while working around the problems with "open session in view". The solution comes in two parts:

- use an extended persistence context that is scoped to the conversation, instead of to the transaction
- use two transactions per request; the first spans the beginning of the restore view phase (some transaction managers begin the transaction later at the beginning of the apply request values phase) until the end of the invoke application phase; the second spans the render response phase

In the next section, we'll tell you how to set up a conversation-scope persistence context. But first we need to tell you how to enable Seam transaction management. Note that you can use conversation-scoped persistence contexts without Seam transaction management, and there are good reasons to use Seam transaction management even when you're not using Seam-managed

persistence contexts. However, the two facilities were designed to work together, and work best when used together.

Seam transaction management is useful even if you're using EJB 3.0 container-managed persistence contexts. But it is especially useful if you use Seam outside a Java EE environment, or in any other case where you would use a Seam-managed persistence context.

10.2.1. Disabling Seam-managed transactions

Seam transaction management is enabled by default for all JSF requests. If you want to *disable* this feature, you can do it 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 a transaction. By default Seam uses a JTA transaction component that integrates with Container Managed and programmatic EJB transactions. If you are working in a Java EE environment, you should install the EJB synchronization component in `components.xml`:

```
<transaction:ejb-transaction />
```

However, if you are working in a non EE 5 container, Seam will try auto detect the transaction synchronization mechanism to use. However, if Seam is unable to detect the correct transaction synchronization to use, you may find you need configure one of the following:

- JPA RESOURCE_LOCAL transactions with the `javax.persistence.EntityTransaction` interface. `EntityTransaction` begins the transaction at the beginning of the apply request values phase.
- Hibernate managed transactions with the `org.hibernate.Transaction` interface. `HibernateTransaction` begins the transaction at the beginning of the apply request values phase.
- 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

Configure JPA RESOURCE_LOCAL transaction management by adding 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 can opt to leave out the `entity-manager` attribute. (see *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. (see *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 configuring Spring managed transactions see *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 it's own transaction synchronization component which requires explicit use of the Seam transaction component when committing a transaction to ensure synchronization callbacks are correctly executed. If in a Java EE environment the `<transaction:ejb-transaction/>` component should be declared in `components.xml` to ensure that Seam synchronization callbacks are correctly called if the container commits a transaction outside of Seam's knowledge.

10.3. Seam-managed persistence contexts

If you're using Seam outside of a Java EE environment, you can't rely upon the container to manage the persistence context lifecycle for you. Even if you are in an EE 5 environment, you might have a complex application with many loosly coupled components that collaborate together in the scope of a single conversation, and in this case you might find that propagation of the persistence context between component is tricky and error-prone.

In either case, you'll 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 is able to perform an optimization that EJB 3.0 specification does not allow containers to use for container-managed extended persistence contexts. Seam supports transparent failover of extended persistence contexts, without the need to replicate any persistence context state between nodes. (We hope to fix this oversight in the next revision of the EJB spec.)

10.3.1. Using a Seam-managed persistence context with JPA

Configuring a managed persistence context is easy. In `components.xml`, we can 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` that manages the lifecycle of `EntityManager` instances for the persistence unit (`EntityManagerFactory` instance) with JNDI name `java:/EntityManagerFactories/bookingData`.

Of course, you need to make sure that you have bound 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 have our `EntityManager` injected using:

```
@In EntityManager bookingDatabase;
```

If you are using EJB3 and mark your class or method `@TransactionAttribute(REQUIRES_NEW)` then the transaction and persistence context shouldn't be propagated to method calls on this object. However as the Seam-managed persistence context is propagated to any component within the conversation, it will be propagated to methods marked `REQUIRES_NEW`. Therefore, if you mark a method `REQUIRES_NEW` then you should access the entity manager using @PersistenceContext.

10.3.2. Using a Seam-managed Hibernate session

Seam-managed Hibernate sessions are similar. In `components.xml`:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

```
<persistence:managed-hibernate-session name="bookingDatabase"
                        auto-create="true"
                session-factory-jndi-name="java:/bookingSessionFactory"/>
```

Where `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>

<propertyname="transaction.factory_class">org.hibernate.transaction.JTATransactionFactory</
property>
   <property name="connection.datasource">java:/bookingDatasource</property>
   ...
</session-factory>
```

Note that Seam does not flush the session, so you should always enable `hibernate.transaction.flush_before_completion` to ensure that the session is automatically flushed before the JTA transaction commits.

We can now have a managed Hibernate `Session` injected into our JavaBean components using the following code:

```
@In Session bookingDatabase;
```

10.3.3. Seam-managed persistence contexts and atomic conversations

Persistence contexts scoped to the conversation allows you to program optimistic transactions that span multiple requests to the server without the need to use the `merge()` operation , without the need to re-load data at the beginning of each request, and without the need to wrestle with the `LazyInitializationException` or `NonUniqueObjectException`.

As with any optimistic transaction management, transaction isolation and consistency can be achieved via use of optimistic locking. Fortunately, both Hibernate and EJB 3.0 make it very easy to use optimistic locking, by providing the `@Version` annotation.

By default, the persistence context is flushed (synchronized with the database) at the end of each transaction. This is sometimes the desired behavior. But very often, we would prefer that all changes are held in memory and only written to the database when the conversation ends successfully. This allows for truly atomic conversations. As the result of a truly stupid and shortsighted decision by certain non-JBoss, non-Sun and non-Sybase members of the EJB 3.0 expert group, there is currently no simple, usable and portable way to implement atomic conversations using EJB 3.0 persistence. However, Hibernate provides this feature as a vendor extension to the `FlushModeTypes` defined by the specification, and it is our expectation that 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 rest ot the 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 the flush to occur:

```
@End
public void commitClaim() {
   em.flush();
}
```

Of course, you could 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.org/schema/seam/components"
  xmlns:core="http://jboss.org/schema/seam/core">
  <core:manager conversation-timeout="120000" default-flush-mode="manual" />
</components>
```

Warning

if you use SMPC in your Stateful bean, manual flush mode is ignored as this mode is specific Hibernate extension to JPA specification. Seam can't control the flush mode of the persistence context on an SFSB - that means no manual flushing on SFSB!

10.4. Using the JPA "delegate"

The `EntityManager` interface lets you access a vendor-specific API via the `getDelegate()` method. Naturally, the most interesting vendor is Hibernate, and the most powerful delegate interface is `org.hibernate.Session`. You'd be nuts to use anything else. Trust me, I'm not biased at all. If you must use a different JPA provider see *Using Alternate JPA Providers*.

But regardless of whether you're using Hibernate (genius!) or something else (masochist, or just not very bright), you'll almost certainly want to use the delegate in your Seam components from time to time. One approach would be the following:

```
@In EntityManager entityManager;

@Create
public void init() {
   ( (Session) entityManager.getDelegate() ).enableFilter("currentVersions");
}
```

But typecasts are unquestionably the ugliest syntax in the Java language, so most people avoid them whenever possible. Here's a different way to get at the delegate. First, add the following line to `components.xml`:

```
<factory name="session"
```

```
        scope="STATELESS"
        auto-create="true"
        value="#{entityManager.delegate}"/>
```

Now we can inject the session 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 using `@PersistenceContext`. This lets you use EL expressions in your query strings, safely and efficiently. 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();
```

Of course, you should never, ever write it like this:

```
User user = em.createQuery("from User where username=" + user.getUsername()) //BAD!
        .getSingleResult();
```

(It is inefficient and vulnerable to SQL injection attacks.)

10.6. Using Hibernate filters

The coolest, and most unique, feature of Hibernate is *filters*. Filters let you provide a restricted view of the data in the database. You can find out more about filters in the Hibernate documentation. But we thought we'd mention an easy way to incorporate filters into a Seam application, one that works especially well with the Seam Application Framework.

Seam-managed persistence contexts may have a list of filters defined, which will be enabled whenever an `EntityManager` or Hibernate `Session` is first created. (Of course, they may 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>
```

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 using Bean Validation.

Let's start by defining our constraints, on our `Location` class:

```
public class Location {
   private String country;
   private String zip;

   @NotNull
   @Size(max=30)
   public String getCountry() { return country; }
   public void setCountry(String c) { country = c; }

   @NotNull
   @Size(max=6)
   @Pattern("^\d*$")
```

```
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

Well, that's a decent first cut, but in practice it might be more elegant to use custom constraints instead of the ones built into Bean Validation:

```
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 route we take, we no longer need to specify the type of validation to be used in the JSF page. Instead, we can 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 requirement for `required="true"` to appear on the control! This is due to a limitation of the JSF validation architecture.

This approach *defines* constraints on the model, and *presents* constraint violations in the view — a significantly better design.

However, it is not much less verbose than what we started with, so let's try `<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 simply adds an `<s:validate>` to every input in the form. For a large form, it can save a lot of typing!

Now we need to do something about displaying feedback to the user when validation fails. Currently we are displaying all messages at the top of the form. In order for the user to correlate the message with an input, you need to define a label using the standard `label` attribute on the input component.

```
<h:inputText value="#{location.zip}" required="true" label="Zip:">
   <s:validate/>
</h:inputText>
```

You can then inject this value into the message string using the placeholder {0} (the first and only parameter passed to a JSF message for a Bean Validation restriction). See the internationalization section for more information regarding where to define these messages.

```
validator.length={0} length must be between {min} and {max}
```

What we would really like to do, though, is display the message next to the field with the error (this is possible in plain JSF), highlight the field and label (this is not possible) and, for good measure, display some image next to the field (also not possible). We also want to display a little colored asterisk next to the label for each required form field. Using this approach, the identifying label is not necessary.

That's quite a lot of functionality we need for each field of our form. We wouldn't want to have to specify highlighting and the layout of the image, message and input field for every field on the form. So, instead, we'll specify the common 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.org/schema/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 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 as the user is navigating 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:ajax event="blur" render="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:ajax event="blur" render="zipDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

It's better style to define explicit ids for important controls on the page, especially if you want to do automated testing for the UI, using some toolkit like Selenium. If you don't provide explicit ids, JSF will generate them, but the generated values will change if you change anything on the page.

```
<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:ajax event="blur" render="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:ajax event="blur" render="zipDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

And what if you want to specify a different message to be displayed when validation fails? You can use the Seam message bundle (and all it's goodies like el expressions inside the message, and per-view message bundles) with the Bean Validation:

```
public class Location {
    private String name;
    private String zip;
```

```
    // Getters and setters for name

    @NotNull
    @Size(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}
```

Groovy integration

One aspect of JBoss Seam is its RAD (Rapid Application Development) capability. While not synonymous with RAD, one interesting tool in this space is dynamic languages. Until recently, choosing a dynamic language was required choosing a completely different development platform (a development platform with a set of APIs and a runtime so great that you would no longer want to use you old legacy Java [sic] APIs anymore, which would be lucky because you would be forced to use those proprietary APIs anyway). Dynamic languages built on top of the Java Virtual Machine, and *Groovy* [http://groovy.codehaus.org] in particular broke this approach in silos.

JBoss Seam now unites the dynamic language world with the Java EE world by seamlessly integrating both static and dynamic languages. JBoss Seam lets the application developer use the best tool for the task, without context switching. Writing dynamic Seam components is exactly like writing regular Seam components. You use the same annotations, the same APIs, the same everything.

12.1. Groovy introduction

Groovy is an agile dynamic language based on the Java language but with additional features inspired by Python, Ruby and Smalltalk. The strengths of Groovy are twofold:

- Java syntax is supported in Groovy: Java code is Groovy code, making the learning curve very smooth
- Groovy objects are Java objects, and Groovy classes are Java classes: Groovy integrates smoothly with existing Java libraries and frameworks.

12.2. Writing Seam applications in Groovy

There is not much to say about it. Since a Groovy object is a Java object, you can virtually write any Seam component, or any class for what it worth, in Groovy and deploy it. You can also mix Groovy classes and Java classes in the same application.

12.2.1. Writing Groovy components

As you should have noticed by now, Seam uses annotations heavily. Be sure to use Groovy 1.1 or above for annotation support. Here are some example of groovy code used in a Seam application.

12.2.1.1. Entity

```
@Entity
@Name("hotel")
class Hotel implements Serializable
```

```
{
    @Id @GeneratedValue
    Long id

    @Size(max=50) @NotNull
    String name

    @Size(max=100) @NotNull
    String address

    @Size(max=40) @NotNull
    String city

    @Size(min=2, max=10) @NotNull
    String state

    @Size(min=4, max=6) @NotNull
    String zip

    @Size(min=2, max=40) @NotNull
    String country

    @Column(precision=6, scale=2)
    BigDecimal price

    @Override
    String toString()
    {
        return "Hotel(Seam Reference Guide,${address},${city},${zip})"
    }
}
```

Groovy natively support the notion of properties (getter/setter), so 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 being generated by the Groovy compiler. This type of syntactic sugar makes the entity code very concise.

12.2.1.2. Seam component

Writing Seam components in Groovy is in no way different than in Java: annotations are used to mark the class as a Seam component.

```
@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])
   }
}
```

12.2.2. seam-gen

Seam gen has a transparent integration with Groovy. You can write Groovy code in seam-gen backed projects without any additional infrastructure requirement. When writing a Groovy entity, simply place your `.groovy` files in `src/main`. Unsurprisingly, when writing an action, simply place your `.groovy` files in `src/hot`.

12.3. Deployment

Deploying Groovy classes is very much like deploying Java classes (surprisingly, no need to write nor comply with a 3-letter composite specification to support a multi-language component framework).

Beyond standard deployments, JBoss Seam has the ability, at development time, to redeploy JavaBeans Seam component classes without having to restart the application, saving a lot of time

in the development / test cycle. The same support is provided for GroovyBeans Seam components when the `.groovy` files are deployed.

12.3.1. Deploying Groovy code

A Groovy class *is* a Java class, with a bytecode representation just like a Java class. To deploy, a Groovy entity, a Groovy Session bean or a Groovy Seam component, a compilation step is necessary. A common approach is to use the *gmaven-plugin* [http://docs.codehaus.org/display/GMAVEN/Home] maven plugin. Once compiles, a Groovy class is in no way different than a Java class and the application server will treat them equally. Note that this allow a seamless mix of Groovy and Java code.

12.3.2. Native .groovy file deployment at development time

JBoss Seam natively supports the deployment of `.groovy` files (ie without compilation) in incremental hotdeployment mode (development only). This enables a very fast edit/test cycle. To set up .groovy deployments, follow the configuration at *Section 2.8, "Seam and incremental hot deployment"* and deploy your Groovy code (`.groovy` files) into the `WEB-INF/dev` directory. The GroovyBean components will be picked up incrementally with no need to restart the application (and obviously not the application server either).

Be aware that the native .groovy file deployment suffers the same limitations as the regular Seam hotdeployment:

- The components must be JavaBeans or GroovyBeans. They cannot be EJB3 bean
- Entities cannot be hotdeployed
- The hot-deployable components will not be visible to any classes deployed outside of `WEB-INF/dev`
- Seam debug mode must be enabled

12.3.3. seam-gen

Seam-gen transparently supports Groovy files deployment and compilation. This includes the native `.groovy` file deployment in development mode (compilation-less). If you create a seam-gen project of type WAR, Java and Groovy classes in `src/hot` will automatically be candidate for the incremental hot deployment. If you are in production mode, the Groovy files will simply be compiled before deployment.

You will find a live example of the Booking demo written completely in Groovy and supporting incremental hot deployment in `examples/groovybooking`.

Writing your presentation layer using Apache Wicket

Seam supports Wicket as an alternative presentation layer to JSF. Take a look at the `wicket` example in Seam which shows the Booking Example ported to Wicket.

Note

Wicket support is new to Seam, so some features which are available in JSF are not yet available when you use Wicket (e.g. pageflow). You'll also notice that the documentation is very JSF-centric and needs reorganization to reflect the first class support for Wicket.

13.1. Adding Seam to your wicket application

The features added to your Wicket application can be split into two categories: bijection and orchestration; these are discussed in detail below.

Extensive use of inner classes is common when building Wicket applications, with the component tree being built in the constructor. Seam fully supports the use of annotation based control in inner classes and constructors (unlike regular Seam components).

Annotations are processed *after* any call to a superclass. This mean's that any injected attributes cannot be passed as an argument in a call to `this()` or `super()` .

When a method is called in an inner class, bijection occurs for any class which encloses it. This allows you to place your bijected variables in the outer class, and refer to them in any inner class.

13.1.1. Bijection

A Seam enabled Wicket application has full access to the all the standard Seam contexts (`EVENT` , `CONVERSATION` , `SESSION` , `APPLICATION` and `BUSINESS_PROCESS`).

To access Seam component's from Wicket, you just need to inject it using `@In` :

```
@In(create=true)
private HotelBooking hotelBooking;
```

Tip

As your Wicket class isn't a full Seam component, there is no need to annotate it `@Name` .

You can also outject an object into the Seam contexts from a Wicket component:

```
@Out(scope=ScopeType.EVENT, required=false)
private String verify;
```

TODO Make this more use case driven

13.1.2. Orchestration

You can secure a Wicket component by using the `@Restrict` annotation. This can be placed on the outer component or any inner components. If `@Restrict` is specified, the component will automatically be restricted to logged in users. You can optionally use an EL expression in the `value` attribute to specify a restriction to be applied. For more refer to the *Chapter 16, Security* .

For example:

```
@Restrict
public class Main extends WebPage
{
  ...
```

Tip

Seam will automatically apply the restriction to any nested classes.

You can demarcate conversations from within a Wicket component through the use of `@Begin` and `@End` . The semantics for these annotations are the same as when used in a Seam component. You can place `@Begin` and `@End` on any method.

Note

The deprecated `ifOutcome` attribute is not supported.

For example:

```
item.add(new Link("viewHotel") {

  @Override
  @Begin
  public void onClick() {
    hotelBooking.selectHotel(hotel);
    setResponsePage(org.jboss.seam.example.wicket.Hotel.class);
  }
};
```

You may have pages in your application which can only be accessed when the user has a long-running conversation active. To enforce this you can use the `@NoConversationPage` annotation:

```
@Restrict
@NoConversationPage(Main.class)
public class Hotel extends WebPage
{
```

If you want to further decouple your application classes, you can use Seam events. Of course, you can raise an event using `Events.instance().raiseEvent("foo")` . Alternatively, you can annotate a method `@RaiseEvent("foo")` ; if the method returns a non-null outcome without exception, the event will be raised.

You can also control tasks and processes in Wicket classes through the use of `@CreateProcess` , `@ResumeTask` , `@BeginTask` , `@EndTask` , `@StartTask` and `@Transition` .

13.2. Setting up your project

Seam needs to instrument the bytecode of your Wicket classes to be able to intercept the annotations you use. The first decision to make is: do you want your code instrumented at runtime as your app is running, or at compile time? The former requires no integration with your build environment, but has a performance penalty when loading each instrumented class for the first time. The latter is faster, but requires you to integrate this instrumentation into your build environment.

13.2.1. Runtime instrumentation

There are two ways to achieve runtime instrumentation. One relies on placing wicket components to be instrumented in a special folder in your WAR deployment. If this is not acceptable or possible, you can also use an instrumentation "agent," which you specify in the command line for launching your container.

13.2.1.1. Location-specific instrumentation

Any classes placed in the `WEB-INF/wicket` folder within your WAR deployment will be automatically instrumented by the seam-wicket runtime. You can arrange to place your wicket pages and components here by specifying a separate output folder for those classes in your IDE, or through the use of ant scripts.

13.2.1.2. Runtime instrumentation agent

The jar file `jboss-seam-wicket.jar` can be used as an instrumentation agent through the Java Instrumentation api. This is accomplished through the following steps:

- Arrange for the `jboss-seam-wicket.jar` file to live in a location for which you have an absolute path, as the Java Instrumentation API does not allow relative paths when specifying the location of an agent lib.
- Add `javaagent:/path/to/jboss-seam-wicket.jar` to the command line options when launching your webapp container:
- In addition, you will need to add an environment variable that specifies packages that the agent should instrument. This is accomplished by a comma separated list of package names:

```
-Dorg.jboss.seam.wicket.instrumented-packages=my.package.one,my.other.package
```

Note that if a package A is specified, classes in subpackages of A are also examined. The classes chosen for instrumentation can be further limited by specifying:

```
-Dorg.jboss.seam.wicket.scanAnnotations=true
```

and then marking instrumentable classes with the `@SeamWicketComponent` annotation, see *Section 13.2.3, " The @SeamWicketComponent annotation "* .

13.2.2. Compile-time instrumentation

Seam supports instrumentation at compile time through either Apache Ant or Apache Maven.

13.2.2.1. Instrumenting with ant

Seam provides an ant task in the `jboss-seam-wicket-ant.jar` . This is used in the following manner:

```
<taskdef name="instrumentWicket"
  classname="org.jboss.seam.wicket.ioc.WicketInstrumentationTask">
 <classpath>
```

```
    <pathelement location="lib/jboss-seam-wicket-ant.jar"/>
    <pathelement location="web/WEB-INF/lib/jboss-seam-wicket.jar"/>
    <pathelement location="lib/javassist.jar"/>
    <pathelement location="lib/jboss-seam.jar"/>
  </classpath>
</taskdef>

<instrumentWicket outputDirectory="${build.instrumented}" useAnnotations="true">
  <classpath refid="build.classpath"/>
  <fileset dir="${build.classes}" includes="**/*.class"/>
</instrumentWicket>
```

This results in the instrumented classes being placed in the directory specified by `${build.instrumented}` . You will then need to instruct ant to copy these classes into `WEB-INF/classes` . If you want to hot deploy the Wicket components, you can copy the instrumented classes to `WEB-INF/dev` ; if you use hot deploy, make sure that your `WicketApplication` class is also hot-deployed. Upon a reload of hot-deployed classes, the entire WicketApplication instance has to be re-initialized, in order to pick up new references to the classes of mounted pages.

The `useAnnotations` attribute is used to make the ant task only include classes that have been marked with the `@SeamWicketComponent` annotation, see *Section 13.2.3, " The @SeamWicketComponent annotation "* .

13.2.2.2. Instrumenting with maven

The jboss maven repository `repository.jboss.org` provides a plugin named `seam-instrument-wicket` with a `process-classes` mojo. An example configuration in your pom.xml might look like:

```
<build>
   <plugins>
      <plugin>
         <groupId>org.jboss.seam</groupId>
         <artifactId>seam-instrument-wicket</artifactId>
         <version>2.2.0</version>
         <configuration>
            <scanAnnotations>true</scanAnnotations>
            <includes>
               <include>your.package.name</include>
            </includes>
         </configuration>
         <executions>
            <execution>
               <id>instrument</id>
```

```
            <phase>process-classes</phase>
            <goals>
               <goal>instrument</goal>
            </goals>
         </execution>
      </executions>
   </plugin>
</plugins>
</build>
```

The above example illustrates that the instrumentation is limited to classes specified by the `includes` element. In this example, the `scanAnnotations` is specified, see *Section 13.2.3, " The @SeamWicketComponent annotation "* .

13.2.3. The `@SeamWicketComponent` annotation

Classes placed in WEB-INF/wicket will unconditionally be instrumented. The other instrumentation mechanisms all allow you to specify that instrumentation should only be applied to classes annotated with the `@SeamWicketComponent` annotation. This annotation is inherited, which means all subclasses of an annotated class will also be instrumented. An example usage is:

```
import org.jboss.seam.wicket.ioc.SeamWicketComponent;

@SeamWicketComponent
public class MyPage extends WebPage
{
 ...
}
```

13.2.4. Defining the Application

A Wicket web application which uses Seam should use `SeamWebApplication` as the base class; this creates hooks into the Wicket lifecycle allowing Seam to automagically propagate the conversation as needed. It also adds status messages to the page.

For example:

The `SeamAuthorizationStrategy` delegates authorization to Seam Security, allowing the use of `@Restrict` on Wicket components. `SeamWebApplication` installs the authorization strategy for you. You can specify the login page by implementing the `getLoginPage()` method.

You'll also need to set the home page of the application by implementing the `getHomePage()` method.

```
public class WicketBookingApplication extends SeamWebApplication
{

  @Override
  public Class getHomePage()
  {
    return Home.class;
  }

  @Override
  protected Class getLoginPage()
  {
    return Home.class;
  }
}
```

Seam automatically installs the Wicket filter for you (ensuring that it is inserted in the correct place for you). But you still need to tell Wicket which `WebApplication` class to use.

```
<components xmlns="http://jboss.org/schema/seam/components"
 xmlns:wicket="http://jboss.org/schema/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.org/schema/seam/wicket
   http://jboss.org/schema/seam/wicket-2.3.xsd">

  <wicket:web-application
    application-class="org.jboss.seam.example.wicket.WicketBookingApplication" />
</components
```

In addition, if you plan to use JSF-based pages in the same application as wicket pages, you'll need to ensure that the jsf exception filter is only enabled for jsf urls:

```
<components xmlns="http://jboss.org/schema/seam/components"
 xmlns:web="http://jboss.org/schema/seam/web"
 xmlns:wicket="http://jboss.org/schema/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.org/schema/seam/web
   http://jboss.org/schema/seam/web-2.3.xsd">

   <!-- Only map the seam jsf exception filter to jsf paths, which we identify with the *.seam path -->
    <web:exception-filter url-pattern="*.seam"/>
```

```
</components
```

Tip

Take a look at the Wicket documentation for more on authorization strategies and other methods you can override on the `Application` class.

The Seam Application Framework

Seam makes it really easy to create applications by writing plain Java classes with annotations, which don't need to extend any special interfaces or superclasses. But we can simplify some common programming tasks even further, by providing a set of pre-built components which can be re-used either by configuration in `components.xml` (for very simple cases) or extension.

The *Seam Application Framework* can reduce the amount of code you need to write when doing basic database access in a web application, using either Hibernate or JPA.

We should emphasize that the framework is extremely simple, just a handful of simple classes that are easy to understand and extend. The "magic" is in Seam itself — the same magic you use when creating any Seam application even without using this framework.

14.1. Introduction

The components provided by the Seam application framework may be used in one of two different approaches. The first way is to install and configure an instance of the component in `components.xml`, just like we have done with other kinds of built-in Seam components. For example, the following fragment from `components.xml` installs a component which can perform 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 that looks a bit too much like "programming in XML" for your taste, you can use extension instead:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

   @In EntityManager personDatabase;

   public EntityManager getEntityManager() {
      return personDatabase;
   }

}
```

The second approach has one huge advantage: you can easily add extra functionality, and override the built-in functionality (the framework classes were carefully designed for extension and customization).

A second advantage is that your classes may be EJB stateful session beans, if you like. (They do not have to be, they can be plain JavaBean components if you prefer.) If you are using JBoss AS, you'll need 4.2.2.GA or later:

```
@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 this time, the Seam Application Framework provides four main built-in components: `EntityHome` and `HibernateEntityHome` for CRUD, along with `EntityQuery` and `HibernateEntityQuery` for queries.

The Home and Query components are written so that they can function with a scope of session, event or conversation. Which scope you use depends upon the state model you wish to use in your application.

The Seam Application Framework only works with Seam-managed persistence contexts. By default, the components will look for a persistence context named `entityManager`.

14.2. Home objects

A Home object provides persistence operations for a particular entity class. Suppose we have our trusty `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 via configuration:

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

Or via extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {}
```

A Home object provides the following operations: `persist()`, `remove()`, `update()` and `getInstance()`. Before you can call the `remove()`, or `update()` operations, you must first set the identifier of the object you are interested in, using the `setId()` method.

We can use a Home directly from a JSF page, for example:

```
<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>
```

Usually, it is much nicer to be able to refer to the `Person` merely as `person`, so let's make that possible by adding a line to `components.xml`:

```
<factory name="person"
        value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                    entity-class="eg.Person" />
```

(If we are using configuration.) Or by adding a `@Factory` method to `PersonHome`:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

   @Factory("person")
   public Person initPerson() { return getInstance(); }

}
```

(If we are using extension.) 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>
```

Well, that lets us create new `Person` entries. Yes, that is all the code that is required! Now, 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`. Page parameters are a great way to do that:

```
<pages>
   <page view-id="/editPerson.xhtml">
      <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.

Suppose we need to create `Person` entries with their nationality initialized. We can do that 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 by 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);
   }

}
```

Of course, the `Country` could be an object managed by another Home object, for example, `CountryHome`.

To add more sophisticated operations (association management, etc), we can just 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 succeeds (a call to `persist()`, `update()` or `remove()` succeeds). By observing this

event we can refresh our queries when the underlying entities are changed. If we only want to refresh certain queries when a particular entity is persisted, updated or removed we can observe the `org.jboss.seam.afterTransactionSuccess.<name>` event (where `<name>` is the simple name of the entity, e.g. an entity called "org.foo.myEntity" has "myEntity" as simple name).

The Home object automatically displays faces messages when an operation is successful. 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"); }

}
```

But the best way to specify the messages is to put them in a resource bundle known to Seam (the bundle named `messages`, by default).

```
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.

The final step is to add validation functionality to the page, using `<s:validateAll>` and `<s:decorate>`, but I'll leave that for you to figure out.

14.3. Query objects

If we need a list of all `Person` instance in the database, we can use a Query object. For example:

```
<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.xhtml" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable>
```

We probably need to support pagination:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"
                        order="lastName"
                        max-results="20"/>
```

We'll use a page parameter to determine the page to display:

```
<pages>
   <page view-id="/searchPerson.xhtml">
      <param name="firstResult" value="#{people.firstResult}"/>
   </page>
</pages>
```

The JSF code for a pagination control is a bit verbose, but manageable:

```
<h1>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.xhtml" 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 a bunch of optional search criteria to narrow the list of results returned. The Query object lets you specify optional "restrictions" to support this important 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.xhtml"/></div>
</h:form>

<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.xhtml" 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 just 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 don't work well with *join fetch* queries - the use of pagination with these queries is not recommended, and you'll have to implement your own method of calculating the total number of results (by overriding `getCountEjbql()`.

The examples in this section have all shown reuse by configuration. However, reuse by extension is equally possible for Query objects.

14.4. Controller objects

A totally optional part of the Seam Application Framework is the class `Controller` and its subclasses `EntityController` `HibernateEntityController` and `BusinessProcessController`. These classes provide nothing more than some convenience methods for access to commonly used built-in components and methods of built-in components. They help save a few keystrokes (characters can add up!) and provide a great launchpad for new users to explore the rich functionality built in to Seam.

For example, here is what `RegisterAction` from the Seam registration example would look like:

```
@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.xhtmlx";
      }
```

```
    else
    {
      addFacesMessage("User #{user.username} already exists");
      return null;
    }
  }

}
```

As you can see, its not an earthshattering improvement...

Seam and JBoss Rules

Seam makes it easy to call JBoss Rules (Drools) rulebases from Seam components or jBPM process definitions.

15.1. 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 it is quite likely that you will need to install multiple rule bases in a rule-driven application.

If you want to use a Drools DSL, you also need to 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>
```

Support for Drools RuleFlow is also available and you can simply add a `.rf` or a `.rfm` as part of your rule files as:

```
<drools:rule-base name="policyPricingRules" rule-files="policyPricingRules.drl,
policyPricingRulesFlow.rf"/>
```

Note that when using the Drools 4.x RuleFlow (`.rfm`) format, you need to specify the -Ddrools.ruleflow.port=true system property on server startup. This is however still an experimental feature and we advise to use the Drools5 (`.rf`) format if possible.

If you want to register a custom consequence exception handler through the RuleBaseConfiguration, you need to write the handler, for 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 need to be dynamically deployable, so a production application will want to use a Drools RuleAgent 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 in `components.xml`:

```
<drools:rule-agent name="insuranceRules"
                   configurationFile="/WEB-INF/deployedrules.properties" />
```

The properties file contains properties specific to the RulesAgent. Here 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, we need to 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 gave the `policyPricingWorkingMemory` a reference back to our rule base via the `ruleBase` configuration property.

We can also add means to be notified of rule engine events, including rules firing, objects being asserted, etc. 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>
```

15.2. 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);
   // if we have a ruleflow, start the process
   policyPricingWorkingMemory.startProcess(startProcessId)
   policyPricingWorkingMemory.fireAllRules();
}
```

15.3. Using rules from a jBPM process definition

You can even allow a rule base to act as a jBPM action handler, decision handler, or assignment handler — in either a pageflow or business process definition.

```
<decision name="approval">

   <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
      <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
      <!-- if a ruleflow was added -->
      <startProcessId>approvalruleflowid</startProcessId>
      <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`.

The `<retractObjects>` element on the other hand specifies EL expressions that return an object or collection of objects to be retracted from the `WorkingMemory`.

There is also support for using Drools for jBPM task assignments:

```
<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 to the rules as Drools globals, namely the jBPM `Assignable`, as `assignable` and a Seam `Decision` object, as `decision`. Rules which handle decisions should call `decision.setOutcome("result")` to determine the result of the decision. Rules which perform assignments should set the actor id using the `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

You can find out more about Drools at *http://www.drools.org*

Caution

Seam comes with enough of Drools' dependencies to implement some simple rules. If you want to add extra capabilities to Drools you should download the full distribution and add in extra dependencies as needed.

Security

16.1. Overview

The Seam Security API provides a multitude of security-related features for your Seam-based application, covering such areas as:

- Authentication - an extensible, JAAS-based authentication layer that allows users to authenticate against any security provider.
- Identity Management - an API for managing a Seam application's users and roles at runtime.
- Authorization - an extremely comprehensive authorization framework, supporting user roles, persistent and rule-based permissions, and a pluggable permission resolver for easily implementing customised security logic.
- Permission Management - a set of built-in Seam components to allow easy management of an application's security policy.
- CAPTCHA support - to assist in the prevention of automated software/scripts abusing your Seam-based site.
- And much more

This chapter will cover each of these features in detail.

16.2. Disabling Security

In some situations it may be necessary to disable Seam Security, for instances during unit tests or because you are using a different approach to security, such as native JAAS. Simply call the static method `Identity.setSecurityEnabled(false)` to disable the security infrastructure. Of course, it's not very convenient to have to call a static method when you want to configure the application, so as an alternative you can control this setting in components.xml:

- Entity Security
- Hibernate Security Interceptor
- Seam Security Interceptor
- Page restrictions
- Servlet API security integration

Assuming you are planning to take advantage of what Seam Security has to offer, the rest of this chapter documents the plethora of options you have for giving your user an identity in the eyes of the security model (authentication) and locking down the application by establishing constraints (authorization). Let's begin with the task of authentication since that's the foundation of any security model.

16.3. Authentication

The authentication features provided by Seam Security are built upon JAAS (Java Authentication and Authorization Service), and as such provide a robust and highly configurable API for handling user authentication. However, for less complex authentication requirements Seam offers a much more simplified method of authentication that hides the complexity of JAAS.

16.3.1. Configuring an Authenticator component

Note

If you use Seam's Identity Management features (discussed later in this chapter) then it is not necessary to create an authenticator component (and you can skip this section).

The simplified authentication method provided by Seam uses a built-in JAAS login module, `SeamLoginModule`, which delegates authentication to one of your own Seam components. This login module is already configured inside Seam as part of a default application policy and as such does not require any additional configuration files. It allows you to write an authentication method using the entity classes that are provided by your own application, or alternatively to authenticate with some other third party provider. Configuring this simplified form of authentication requires the `identity` component to be configured in `components.xml`:

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:core="http://jboss.org/schema/seam/core"
        xmlns:security="http://jboss.org/schema/seam/security"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
                    "http://jboss.org/schema/seam/components  http://jboss.org/schema/seam/
components-2.3.xsd
          http://jboss.org/schema/seam/security http://jboss.org/schema/seam/security-2.3.xsd">

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

</components>
```

The EL expression `#{authenticator.authenticate}` is a method binding that indicates the `authenticate` method of the `authenticator` component will be used to authenticate the user.

16.3.2. Writing an authentication method

The `authenticate-method` property specified for `identity` in `components.xml` specifies which method will be used by `SeamLoginModule` to authenticate users. This method

takes no parameters, and is expected to return a boolean, which indicates whether authentication is successful or not. The user's username and password can be obtained from `Credentials.getUsername()` and `Credentials.getPassword()`, respectively (you can get a reference to the `credentials` component via `Identity.instance().getCredentials()`). Any roles that the user is a member of should be assigned using `Identity.addRole()`. Here's 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;
    }

  }

}
```

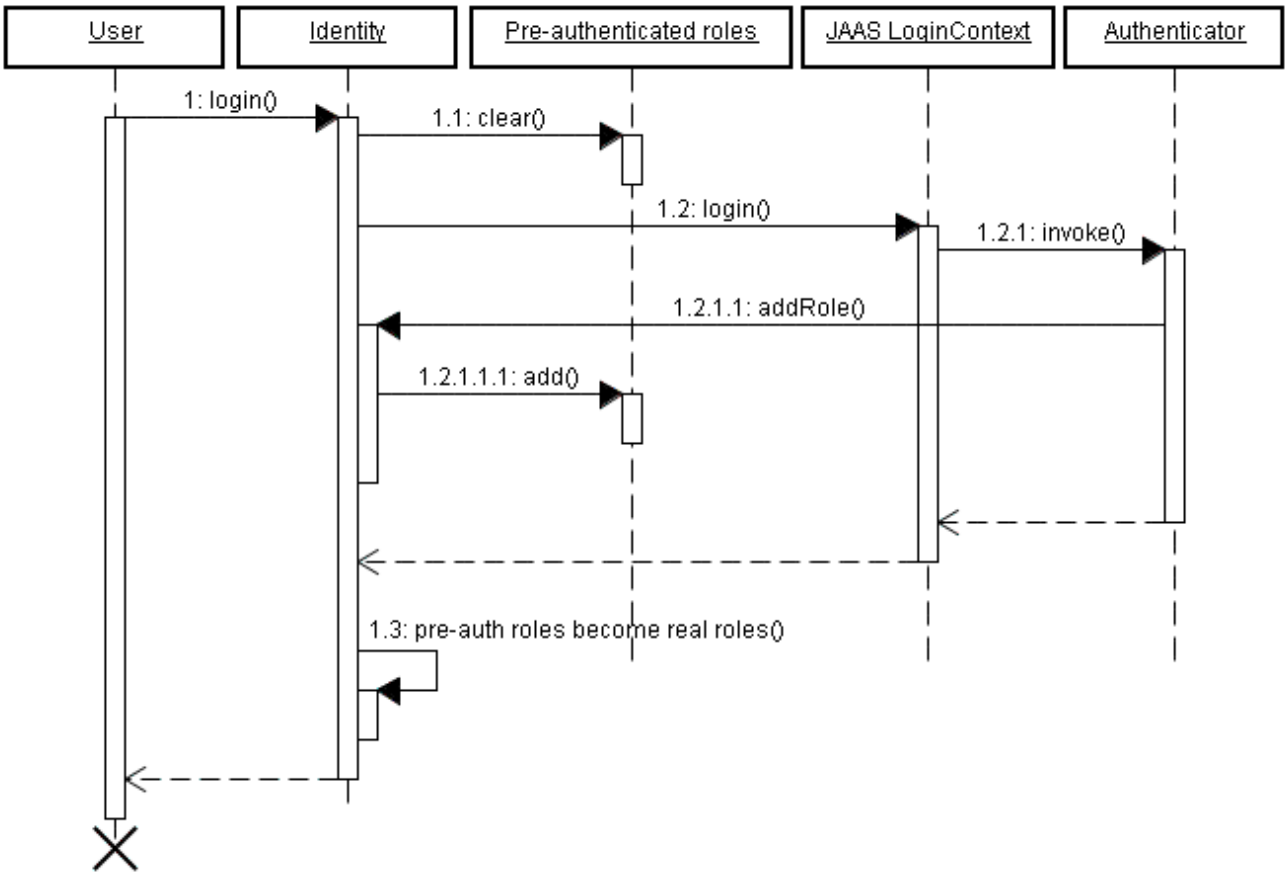
In the above example, both `User` and `UserRole` are application-specific entity beans. The `roles` parameter is populated with the roles that the user is a member of, which should be added to the `Set` as literal string values, e.g. "admin", "user". In this case, if the user record is not found and a `NoResultException` thrown, the authentication method returns `false` to indicate the authentication failed.

Tip

When writing an authenticator method, it is important that it is kept minimal and free from any side-effects. This is because there is no guarantee as to how many times the authenticator method will be called by the security API, and as such it may be invoked multiple times during a single request. Because of this, any special code that should execute upon a successful or failed authentication should be written by implementing an event observer. See the section on Security Events further down in this chapter for more information about which events are raised by Seam Security.

16.3.2.1. Identity.addRole()

The `Identity.addRole()` method behaves differently depending on whether the current session is authenticated or not. If the session is not authenticated, then `addRole()` should *only* be called during the authentication process. When called here, the role name is placed into a temporary list of pre-authenticated roles. Once authentication is successful, the pre-authenticated roles then become "real" roles, and calling `Identity.hasRole()` for those roles will then return true. The following sequence diagram represents the list of pre-authenticated roles as a first class object to show more clearly how it fits in to the authentication process.



If the current session is already authenticated, then calling `Identity.addRole()` will have the expected effect of immediately granting the specified role to the current user.

16.3.2.2. Writing an event observer for security-related events

Say for example, that upon a successful login that some user statistics must be updated. This would be done by writing 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. You can find more information about security-related events later in this chapter.

16.3.3. 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()` will authenticate the user using the provided credentials. Here's an example of a simple login form:

```
<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, logging out the user is done by calling `#{identity.logout}`. Calling this action will clear the security state of the currently authenticated user, and invalidate the user's session.

16.3.4. Configuration Summary

So to sum up, there are the 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.

16.3.5. Remember Me

Seam Security supports the same kind of "Remember Me" functionality that is commonly encountered in many online web-based applications. It is actually supported in two different "flavours", or modes - the first mode allows the username to be stored in the user's browser as a cookie, and leaves the entering of the password up to the browser (many modern browsers are capable of remembering passwords).

The second mode supports the storing of a unique token in a cookie, and allows a user to authenticate automatically upon returning to the site, without having to provide a password.

Warning

Automatic client authentication with a persistent cookie stored on the client machine is dangerous. While convenient for users, any cross-site scripting security hole in your website would have dramatically more serious effects than usual. Without the authentication cookie, the only cookie to steal for an attacker with XSS is the cookie of the current session of a user. This means the attack only works when the user has an open session - which should be a short timespan. However, it is much more attractive and dangerous if an attacker has the possibility to steal a persistent Remember Me cookie that allows him to login without authentication, at any time. Note that this all depends on how well you protect your website against XSS attacks - it's up to you to make sure that your website is 100% XSS safe - a non-trivial achievement for any website that allows user input to be rendered on a page.

Browser vendors recognized this issue and introduced a "Remember Passwords" feature - today almost all browsers support this. Here, the browser remembers the login username and password for a particular website and domain, and fills out the login form automatically when you don't have an active session with the website. If you as a website designer then offer a convenient login keyboard shortcut, this approach is almost as convenient as a "Remember Me" cookie and much

safer. Some browsers (e.g. Safari on OS X) even store the login form data in the encrypted global operation system keychain. Or, in a networked environment, the keychain can be transported with the user (between laptop and desktop for example), while browser cookies are usually not synchronized.

To summarize: While everyone is doing it, persistent "Remember Me" cookies with automatic authentication are a bad practice and should not be used. Cookies that "remember" only the users login name, and fill out the login form with that username as a convenience, are not an issue.

To enable the remember me feature for the default (safe, username only) mode, no special configuration is required. In your login form, simply bind the remember me checkbox to `rememberMe.enabled`, like 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>
```

16.3.5.1. Token-based Remember-me Authentication

To use the automatic, token-based mode of the remember me feature, you must first configure a token store. The most common scenario is to store these authentication tokens within a database (which Seam supports), however it is possible to implement your own token store by implementing the `org.jboss.seam.security.TokenStore` interface. This section will assume you will be using the provided `JpaTokenStore` implementation to store authentication tokens inside a database table.

The first step is to create a new Entity which will contain the tokens. The following example shows a possible structure that you may use:

```
@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;
  }
}
```

As you can see from this listing, a couple of 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 will contain the authentication tokens.

The next step is to configure `JpaTokenStore` to use this entity bean to store and retrieve authentication tokens. This is done in `components.xml` by specifying the `token-class` attribute:

```
<security:jpa-token-store                                                    token-
class="org.jboss.seam.example.seamspace.AuthenticationToken" />
```

Once this is done, the last thing to do is to configure the `RememberMe` component in `components.xml` also. Its `mode` should be set to `autoLogin`:

```
<security:remember-me mode="autoLogin"/>
```

That is all that is required - automatic authentication will now occur for users revisiting your site (as long as they check the "remember me" checkbox).

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>
```

16.3.6. Handling Security Exceptions

To prevent users from receiving the default error page in response to a security error, it's recommended that `pages.xml` is configured to redirect security errors to a more "pretty" page. The two main types of exceptions thrown by the security API are:

- `NotLoggedInException` - This exception is thrown if 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`, it is recommended that the user is redirected to either 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 even more sophisticated handling of login redirection, so Seam includes some special functionality for handling this problem.

16.3.7. Login Redirection

You can ask Seam to redirect the user to a login screen when an unauthenticated user tries to access a particular view (or wildcarded view id) as follows:

```
<pages login-view-id="/login.xhtml">

   <page view-id="/members/*" login-required="true"/>

   ...

</pages>
```

Tip

This is less of a blunt instrument than the exception handler shown above, but should probably be used in conjunction with it.

After the user logs in, we want to automatically send them back where they came from, so they can retry the action that required logging in. If you add the following event listeners to `components.xml`,

attempts to access a restricted view while not logged in will be remembered, so that upon the user successfully logging in they will be 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 that login redirection is implemented as a conversation-scoped mechanism, so don't end the conversation in your `authenticate()` method.

16.3.8. HTTP Authentication

Although not recommended for use unless absolutely necessary, Seam provides means for authenticating using either HTTP Basic or HTTP Digest (RFC 2617) methods. To use either form of authentication, the `authentication-filter` component must be enabled in components.xml:

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

To enable the filter for basic authentication, set `auth-type` to `basic`, or for digest authentication, set it to `digest`. If using digest authentication, the `key` and `realm` must also be set:

```
<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.

16.3.8.1. 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;
  }
}
```

16.3.9. Advanced Authentication Features

This section explores some of the advanced features provided by the security API for addressing more complex security requirements.

16.3.9.1. Using your container's JAAS configuration

If you would rather not use the simplified JAAS configuration provided by the Seam Security API, you may instead delegate to the default system JAAS configuration by providing a `jaas-config-name` property in `components.xml`. For example, if you are using JBoss AS and wish to use the `other` policy (which uses the `UsersRolesLoginModule` login module provided by JBoss AS), then the entry in `components.xml` would look like this:

```
<security:identity jaas-config-name="other"/>
```

Please keep in mind that doing this does not mean that your user will be authenticated in whichever container your Seam application is deployed in. It merely instructs Seam Security to authenticate itself using the configured JAAS security policy.

16.4. Identity Management

Identity Management provides a standard API for the management of a Seam application's users and roles, regardless of which identity store (database, LDAP, etc) is used on the backend. At

the center of the Identity Management API is the `identityManager` component, which provides all the 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 it may be used, the `identityManager` must first be configured with one or more `IdentityStores`. These components do the actual work of interacting with the backend security provider, whether it be a database, LDAP server, or something else.

16.4.1. Configuring IdentityManager

The `identityManager` component allows for separate identity stores to be configured for authentication and authorization operations. This means that it is possible for users to be authenticated against one identity store, for example an LDAP directory, yet have their roles loaded from another identity store, such as a relational database.

Seam provides two `IdentityStore` implementations out of the box; `JpaIdentityStore` uses a relational database to store user and role information, and is the default identity store that is used if nothing is explicitly configured in the `identityManager` component. The other implementation that is provided is `LdapIdentityStore`, which uses an LDAP directory to store users and roles.

There are two configurable properties for the `identityManager` component - `identityStore` and `roleIdentityStore`. The value for these properties must be an EL expression referring to a Seam component implementing the `IdentityStore` interface. As already mentioned, if left unconfigured then `JpaIdentityStore` will be assumed by default. If only the `identityStore` property is configured, then the same value will be used for `roleIdentityStore` also. 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 both of these identity store implementations in greater detail.

16.4.2. JpaIdentityStore

This identity store allows for users and roles to be stored inside a relational database. It is designed to be as unrestrictive as possible in regards to database schema design, allowing a great deal of flexibility in the underlying table structure. This is achieved through the use of a set of special annotations, allowing entity beans to be configured to store user and role records.

16.4.2.1. Configuring JpaIdentityStore

`JpaIdentityStore` requires that both the `user-class` and `role-class` properties are configured. These properties should refer to the entity classes that are to be used to store both user and role records, respectively. The following example shows the configuration from `components.xml` in the SeamSpace example:

```
<security:jpa-identity-store
  user-class="org.jboss.seam.example.seamspace.MemberAccount"
  role-class="org.jboss.seam.example.seamspace.MemberRole"/>
```

16.4.2.2. Configuring the Entities

As already mentioned, a set of special annotations are used to configure entity beans for storing users and roles. The following table lists each of the annotations, and their descriptions.

Table 16.1. 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`. E.g:

Annotation	Status	Description

		``
		@UserPassword (hash="md5")
		public String getPasswordHash() {
		  return passwordHash;
		}
		``
		If an application requires a hash algorithm that isn't supported natively by Seam, it is possible to extend the `PasswordHash` component to implement other hashing algorithms.

`@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 status of the user. This should be a boolean property, and if not present then 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 further down.

Table 16.2. Role Entity Annotations

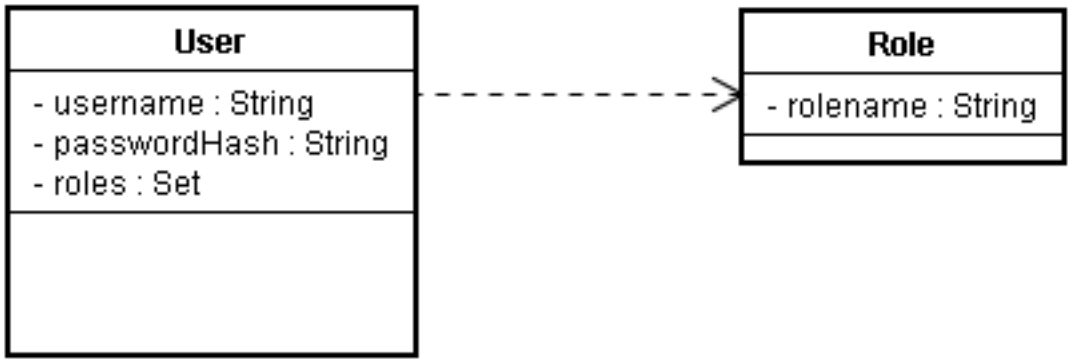
Annotation	Status	Description
`@RoleName`	Required	This annotation marks the field or method containing the name of the role.
`@RoleGroups`	Optional	This annotation marks the field or method containing the group memberships of the role.
`@RoleConditional`	Optional	This annotation marks the field or method indicating whether the role is conditional or not. Conditional roles are explained later in this chapter.

16.4.2.3. 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.

16.4.2.3.1. Minimal schema example

In this bare minimal example, a simple user and role table are linked via a many-to-many relationship using a cross-reference table named `UserRoles`.

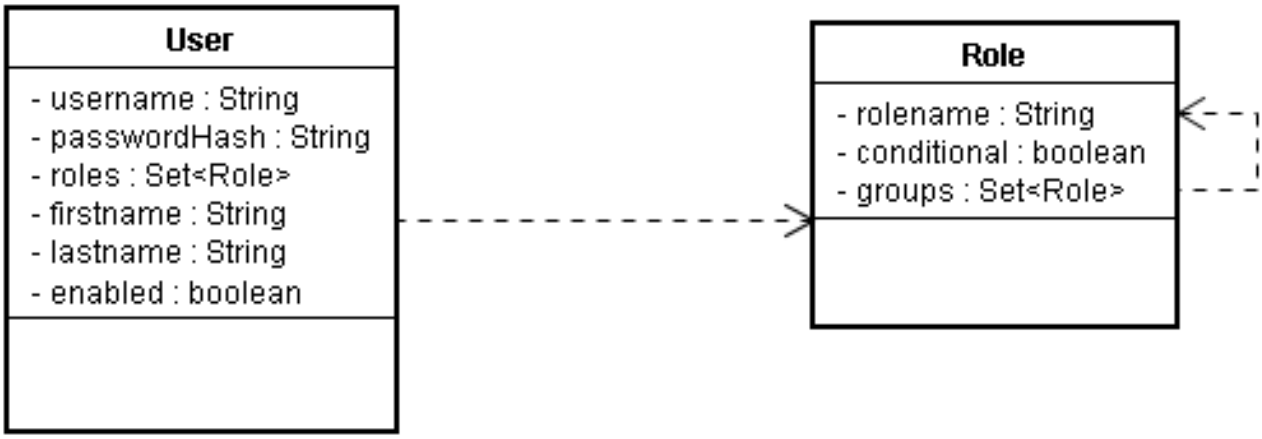



```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;

  @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; }
}
```

16.4.2.3.2. Complex Schema Example

This example builds on the above minimal example by including all of the 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; }

}
```

16.4.2.4. JpaIdentityStore Events

When using `JpaIdentityStore` as the identity store implementation with `IdentityManager`, a few events are raised as a result of invoking certain `IdentityManager` methods.

16.4.2.4.1. 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 will be raised passing the entity instance as an event parameter. The entity will be an instance of the `user-class` configured for `JpaIdentityStore`.

Writing an observer for this event may be useful for setting additional field values on the entity, which aren't set as part of the standard `createUser()` functionality.

16.4.2.4.2. 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 also need to persist other entities that reference the user entity, for example contact detail records or other user-specific data.

16.4.2.4.3. 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 that is being authenticated.

16.4.3. LdapIdentityStore

This identity store implementation is designed for working with user records stored in an LDAP directory. It is very highly configurable, allowing great flexibility in how both users and roles are stored in the directory. The following sections describe the configuration options for this identity store, and provide some configuration examples.

16.4.3.1. Configuring LdapIdentityStore

The following table describes the available properties that can be configured in `components.xml` for `LdapIdentityStore`.

Table 16.3. 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 is listening 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 for locating the role record.
`role-DN-suffix`	`,ou=Roles,dc=acme,dc=com`	This value is appended to the role name to form the DN for locating 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.
`user-role-attribute`	`roles`	This is the name of the attribute of the user record that contains 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.

Property	Default Value	Description
`user-name-attribute`	`uid`	Indicates which attribute of the user record contains the username.
`user-password-attribute`	`userPassword`	Indicates which attribute of the user record contains the user's password.
`first-name-attribute`	`null`	Indicates which attribute of the user record contains the user's first name.
`last-name-attribute`	`sn`	Indicates which attribute of the user record contains the user's last name.
`full-name-attribute`	`cn`	Indicates which attribute of the user record contains the user's full (common) name.
`enabled-attribute`	`null`	Indicates which attribute of the user record determines whether the user is enabled.
`role-name-attribute`	`cn`	Indicates which attribute of the role record contains the name of the role.
`object-class-attribute`	`objectClass`	Indicates which attribute 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.
`security-authentication-type`	`simple`	The security level to use. Possible values are "none", "simple" and "strong".

16.4.3.2. LdapIdentityStore Configuration Example

The following configuration example shows how `LdapIdentityStore` may be configured for an LDAP directory running on fictional host `directory.mycompany.com`. The users are stored within this directory under the context `ou=Person,dc=mycompany,dc=com`, and are identified using the `uid` attribute (which corresponds to their username). Roles are stored in their own context, `ou=Roles,dc=mycompany,dc=com` and 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 may 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"
  />
```

16.4.4. Writing your own IdentityStore

Writing your own identity store implementation allows you to authenticate and perform identity management operations against security providers that aren't supported out of the box by Seam. Only a single class is required to achieve this, and it must implement the `org.jboss.seam.security.management.IdentityStore` interface.

Please refer to the JavaDoc for `IdentityStore` for a description of the methods that must be implemented.

16.4.5. Authentication with Identity Management

If you are using the Identity Management features in your Seam application, then it is not required to provide an authenticator component (see previous Authentication section) to enable authentication. Simply omit the `authenticate-method` from the `identity` configuration in `components.xml`, and the `SeamLoginModule` will by default use `IdentityManager` to authenticate your application's users, without any special configuration required.

16.4.6. Using IdentityManager

The `IdentityManager` can be accessed either by injecting it into your Seam component as follows:

```
@In IdentityManager identityManager;
```

or by accessing it through its static `instance()` method:

```
IdentityManager identityManager = IdentityManager.instance();
```

The following table describes `IdentityManager`'s API methods:

Table 16.4. 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, or `false` if not.
`deleteUser(String name)`	`boolean`	Deletes the user account with the specified name. Returns `true` if successful, or `false` if not.
`createRole(String role)`	`boolean`	Creates a new role, with the specified name. Returns `true` if successful, or `false` if not.
`deleteRole(String name)`	`boolean`	Deletes the role with the specified name. Returns `true` if successful, or `false` if not.
`enableUser(String name)`	`boolean`	Enables the user account with the specified name. Accounts that are not enabled are not able to authenticate. Returns `true` if successful, or `false` if not.
`disableUser(String name)`	`boolean`	Disables the user account with the specified name. Returns `true` if

Method	Returns	Description
		successful, or `false` if not.
`changePassword(String name, String password)`	`boolean`	Changes the password for the user account with the specified name. Returns `true` if successful, or `false` if not.
`isUserEnabled(String name)`	`boolean`	Returns `true` if the specified user account is enabled, or `false` if it isn't.
`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 it is already granted to the user.
`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 doesn't.
`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

Method	Returns	Description
		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 the names of all the roles explicitly granted to the specified user name.
`getImpliedRoles(String name)`	`List`	Returns a list of the names of all the roles implicitly granted to the specified user name. Implicitly granted roles include those that are not directly granted to a user, rather they are granted to the roles that the user is a member of. For example, is 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.
`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	Returns	Description
		method. It does not change the state of the `Identity` component - to perform a proper Seam login 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.

Using the Identity Management API requires that the calling user has the appropriate authorization to invoke its methods. The following table describes the permission requirements for each of the methods in `IdentityManager`. The permission targets listed below are literal String values.

Table 16.5. 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 access to all Identity Management-related methods to members of the `admin` role:

```
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
```

16.5. Error Messages

The security API produces a number of default faces messages for various security-related events. The following table lists the message keys that can be used to override these messages by specifying them in a `message.properties` resource file. To suppress the message, just put the key with an empty value in the resource file.

Table 16.6. Security Message Keys

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 login 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.

16.6. Authorization

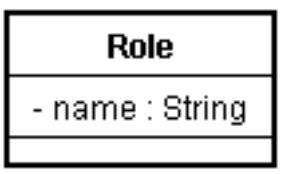
There are a number of authorization mechanisms provided by the Seam Security API for securing access to components, component methods, and pages. This section describes each of these. An important thing to note is that if you wish to use any of the advanced features (such as rule-based permissions) then your `components.xml` may need to be configured to support this - see the Configuration section above.

16.6.1. Core concepts

Seam Security is built around the premise of users being granted roles and/or permissions, allowing them to perform operations that may not otherwise be permissible for users without the necessary security privileges. Each of the authorization mechanisms provided by the Seam Security API are built upon this core concept of roles and permissions, with an extensible framework providing multiple ways to secure application resources.

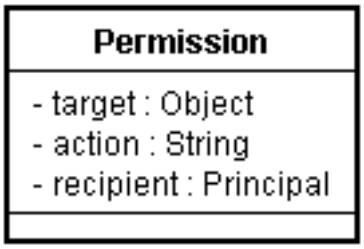
16.6.1.1. What is a role?

A role is a *group*, or *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 just a name such as "admin", "user", "customer", etc. They can be granted either to users (or in some cases to other roles), and are used to create logical groups of users for the convenient assignment of specific application privileges.



16.6.1.2. What is a permission?

A permission is a privilege (sometimes once-off) for performing a single, specific action. It is entirely possible to build an application using nothing but permissions, however roles offer a higher level of convenience when granting privileges to groups of users. They are slightly more complex in structure than roles, essentially 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".



Within this documentation, permissions are generally represented in the form `target:action` (omitting the recipient, although in reality one is always required).

16.6.2. Securing components

Let's start by examining the simplest form of authorization, component security, starting with the `@Restrict` annotation.

@Restrict vs Typesafe security annotations

While using the `@Restrict` annotation provides a powerful and flexible method for security component methods due to its ability to support EL expressions, it is recommended that the typesafe equivalent (described later) be used, at least for the compile-time safety it provides.

16.6.2.1. The @Restrict annotation

Seam components may be secured either at the method or the class level, using the `@Restrict` annotation. If both a method and it's 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, then an exception will be thrown as per the contract for `Identity.checkRestriction()` (see Inline Restrictions). A `@Restrict` on just the component class itself is equivalent to 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, the implied permission required to call the `delete()` method is `account:delete`. The equivalent of this would be to write `@Restrict("#{s:hasPermission('account','delete')}")`. Now let's look at another example:

```
@Restrict @Name("account")
public class AccountAction {
   public void insert() {
     ...
   }
   @Restrict("#{s:hasRole('admin')}")
   public void delete() {
     ...
   }
}
```

This time, the component class itself is annotated with `@Restrict`. This means that any methods without an overriding `@Restrict` annotation require an implicit permission check. In the case of this example, 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 any further, let's address the `#{s:hasRole()}` expression seen in the above example. Both `s:hasRole` and `s:hasPermission` are EL functions, which 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 reference any objects that exist within a Seam context. This is extremely useful when performing permission checks for a specific object instance. Look at this example:

```
@Name("account")
public class AccountAction {
   @In Account selectedAccount;
   @Restrict("#{s:hasPermission(selectedAccount,'modify')}")
   public void modify() {
```

```
        selectedAccount.modify();
    }
}
```

The interesting thing to note from this example is the reference to `selectedAccount` seen within the `hasPermission()` function call. The value of this variable will be looked up from within the Seam context, and passed to the `hasPermission()` method in `Identity`, which in this case can then determine if the user has the required permission for modifying the specified `Account` object.

16.6.2.2. Inline restrictions

Sometimes it might be desirable to perform a security check in code, without using the `@Restrict` annotation. In this situation, simply use `Identity.checkRestriction()` to evaluate a security expression, like this:

```
public void deleteCustomer() {
    Identity.instance().checkRestriction("#{s:hasPermission(selectedCustomer,'delete')}");
}
```

If the expression specified doesn't evaluate to `true`, either

- if the user is not logged in, a `NotLoggedInException` exception is thrown or
- if the user is logged in, an `AuthorizationException` exception is thrown.

It is also possible to 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");
```

16.6.3. Security in the user interface

One indication of a well designed user interface is that the user is not presented with options for which they don't have the necessary privileges to use. Seam Security allows conditional rendering of either 1) sections of a page or 2) individual controls, based upon the privileges of the user, using the very same EL expressions that are used for component security.

Let's take a look at some examples of interface security. First of all, let's pretend that we have a login form that should only be rendered if the user is not already logged in. Using the `identity.isLoggedIn()` property, we can write this:

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}">
```

If the user isn't logged in, then the login form will be rendered - very straight forward so far. Now let's pretend there is a menu on the page that contains some actions which should only be accessible to users in the `manager` role. Here's one way that these could be written:

```
<h:outputLink action="#{reports.listManagerReports}" rendered="#{s:hasRole('manager')}">
   Manager Reports
</h:outputLink>
```

This is also quite straight forward. If the user is not a member of the `manager` role, then 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.

Now for something more complex. Let's say you have a `h:dataTable` control on a page listing records for which you may or may not wish to render action links depending on the user's privileges. The `s:hasPermission` EL function allows us to pass in an object parameter which can be used to determine whether the user has the requested permission for that object or not. Here's how a dataTable with secured links might look:

```
<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>
```

16.6.4. Securing pages

Page security requires that the application is using a `pages.xml` file, however is extremely simple to configure. Simply include a `<restrict/>` element within the `page` elements that you wish to secure. If no explicit restriction is specified by the `restrict` element, an implied permission of `/viewId.xhtml:render` will be checked when the page is accessed via a non-faces (GET) request, and a permission of `/viewId.xhtml:restore` will be required when any JSF postback (form submission) originates from the page. Otherwise, the specified restriction will be evaluated as a standard security expression. Here's a couple of examples:

```
<page view-id="/settings.xhtml">
   <restrict/>
</page>
```

This page has an implied permission of `/settings.xhtml:render` required 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.

16.6.5. Securing Entities

Seam security also makes it possible to apply security restrictions to read, insert, update and delete actions 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 security check that is performed 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`.

It is also possible to only restrict certain actions, by placing a `@Restrict` annotation on the relevant entity lifecycle 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.

Here's an example of how an entity would be configured to perform a security check for any `insert` operations. Please note that the method is not required to do anything, the only important thing in regard to security is how it is annotated:

```
@PrePersist @Restrict
public void prePersist() {}
```

Using `/META-INF/orm.xml`

You can also specify the call back 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>
```

Of course, you still need to annotate the `prePersist()` method on `Customer` with `@Restrict`

And here's an example of an entity permission rule that checks if the authenticated user is allowed to insert a new `MemberBlog` record (from the seamspace example). The entity for which the security check is being made 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 will grant the permission `memberBlog:insert` if the currently authenticated user (indicated by the `Principal` fact) has the same name as the member for which the blog entry is being created. The "`principal: Principal()`" structure that can be seen in the example code is a variable binding - it binds the instance of the `Principal` object from the working memory (placed there during authentication) and assigns it to a variable called `principal`. Variable bindings allow the value to be referred to in other places, such as the following line which compares the member's username to the `Principal` name. For more details, please refer to the JBoss Rules documentation.

Finally, we need to install a listener class that integrates Seam security with your JPA provider.

16.6.5.1. Entity security with JPA

Security checks for EJB3 entity beans are performed with an `EntityListener`. You can install this listener by using 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>
```

16.6.5.2. Entity security with a Managed Hibernate Session

If you are using a Hibernate `SessionFactory` configured via Seam, and are using annotations, or `orm.xml`, then you don't need to do anything special to use entity security.

16.6.6. Typesafe Permission Annotations

Seam provides a number of annotations that may be used as an alternative to `@Restrict`, which have the added advantage of providing compile-time safety as they don't support arbitrary EL expressions in the same way that `@Restrict` does.

Out of the box, Seam comes with annotations for standard CRUD-based permissions, however it is a simple matter to add your own. The following annotations are provided in the `org.jboss.seam.annotations.security` package:

- @Insert
- @Read
- @Update
- @Delete

To use these annotations, simply place them on the method or parameter for which you wish to perform a security check. If placed on a method, then they should specify a target class for which the permission will be checked. Take the following example:

```
@Insert(Customer.class)
public void createCustomer() {
  ...
}
```

In this example, a permission check will be performed for the user to ensure that they have the rights to create new `Customer` objects. The target of the permission check will be `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`.

It is also possible to annotate the parameters of a component method in the same way. If this is done, then it is not required to specify a permission target (as the parameter value itself will be the target of the permission check):

```
public void updateCustomer(@Update Customer customer) {
  ...
}
```

To create your own security annotation, you simply need to 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 (which is the lower case version of the annotation name) with another value, you can specify it within the `@PermissionCheck` annotation:

```
@PermissionCheck("upgrade")
```

16.6.7. Typesafe Role Annotations

In addition to supporting typesafe permission annotation, Seam Security also provides typesafe role annotations that allow you to restrict access to component methods based on the role memberships of the currently authenticated user. Seam provides one such annotation out of the box, `org.jboss.seam.annotations.security.Admin`, used to restrict access to a method to users that are a member of the `admin` role (so long as your own application supports such a role). To create your own role annotations, simply meta-annotate them with `org.jboss.seam.annotations.security.RoleCheck`, like in the following example:

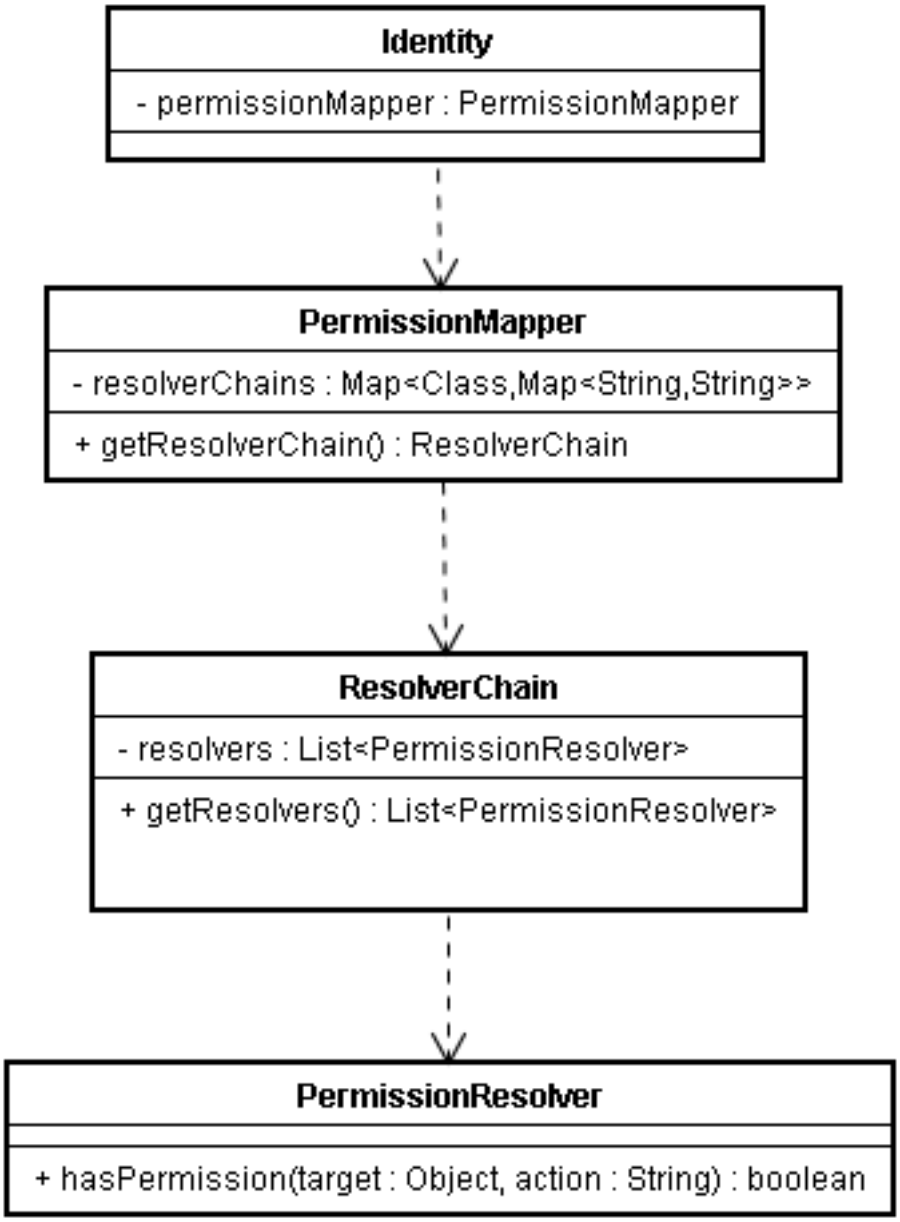
```
@Target({METHOD})
@Documented
```

```
@Retention(RUNTIME)
@Inherited
@RoleCheck
public @interface User {
}
```

Any methods subsequently annotated with the `@User` annotation as shown in the above example will be automatically intercepted and the user checked for the membership of the corresponding role name (which is the lower case version of the annotation name, in this case `user`).

16.6.8. 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.

16.6.8.1. PermissionResolver

This is actually an interface, which provides methods for resolving individual object permissions. Seam provides the following built-in `PermissionResolver` implementations, which are described in more detail later in the chapter:

- `RuleBasedPermissionResolver` - This permission resolver uses Drools to resolve rule-based permission checks.
- `PersistentPermissionResolver` - This permission resolver stores object permissions in a persistent store, such as a relational database.

16.6.8.1.1. Writing your own PermissionResolver

It is very simple to implement your own permission resolver. The `PermissionResolver` interface defines only two methods that must be implemented, as shown by the following table. By deploying your own `PermissionResolver` implementation in your Seam project, it will be automatically scanned during deployment and registered with the default `ResolverChain`.

Table 16.7. PermissionResolver interface

Return type	Method	Description
`boolean`	`hasPermission(Object target, String action)`	This method must resolve whether the currently authenticated user (obtained via a call to `Identity.getPrincipal()`) has the permission specified by the `target` and `action` parameters. It should return `true` if the user has the permission, or `false` if they don't.
`void`	`filterSetByAction(Set<Object> targets, String action)`	This method should remove any objects from the specified set, that would return `true` if passed to the `hasPermission()` method with the same `action` parameter value.

Note

As they are cached in the user's session, any custom `PermissionResolver` implementations must adhere to a couple of restrictions. Firstly, they may not contain any state that is finer-grained than session scope (and the scope of the component itself should either be application or session). Secondly, they must not use dependency injection as they may be accessed from multiple threads simultaneously. In fact, for performance reasons it is recommended that they are annotated with `@BypassInterceptors` to bypass Seam's interceptor stack altogether.

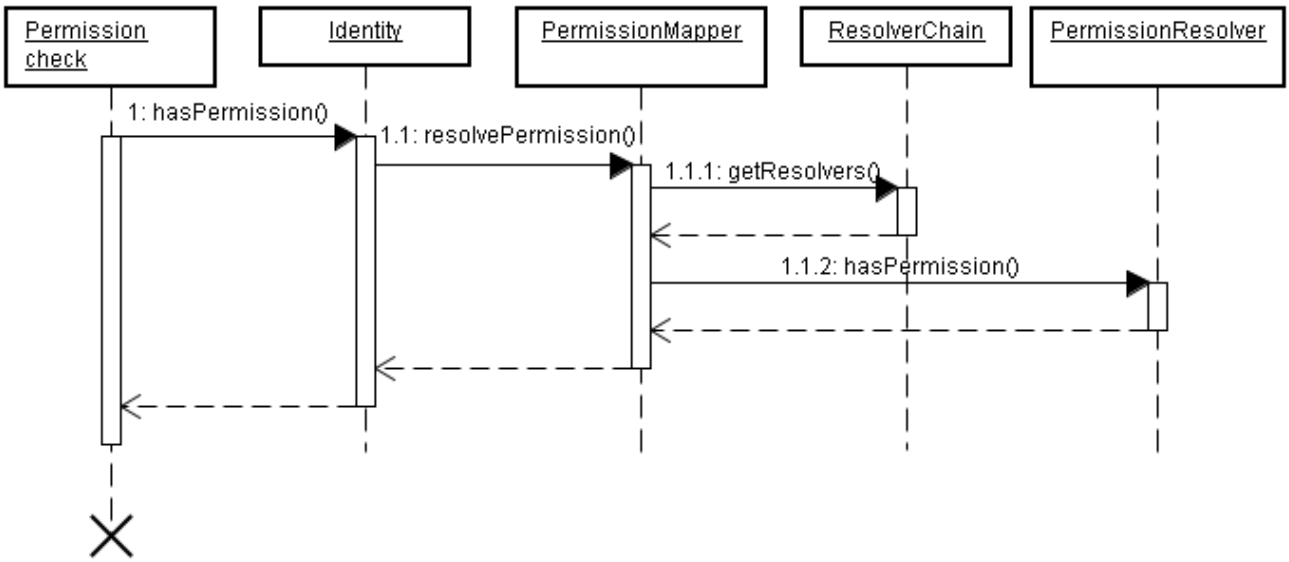
16.6.8.2. ResolverChain

A `ResolverChain` contains an ordered list of `PermissionResolvers`, for the purpose of resolving 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 for some reason 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 (explanation follows). A permission check can originate from a number of possible sources, for example - the security interceptor, the `s:hasPermission` EL function, or via an API call to `Identity.checkPermission`:



- 1. A permission check is initiated somewhere (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 `PermissionResolvers` it contains via a call to `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 any of the `PermissionResolvers` return `true`, then the permission check has succeeded and the `PermissionMapper` also returns `true` to `Identity`. If none of the `PermissionResolvers` return true, then the permission check has failed.

16.6.9. RuleBasedPermissionResolver

One of the built-in permission resolvers provided by Seam, `RuleBasedPermissionResolver` allows permissions to be evaluated based on a set of Drools (JBoss Rules) security rules. A couple of the advantages of using a rule engine are 1) a centralized location for the business logic that

is used to evaluate user permissions, and 2) speed - Drools uses very efficient algorithms for evaluating large numbers of complex rules involving multiple conditions.

16.6.9.1. Requirements

If using the rule-based permission features provided by Seam Security, the following jar files are required by Drools to be distributed with your project:

- knowledge-api.jar
- drools-compiler.jar
- drools-core.jar
- drools-decisiontables.jar
- drools-templates.jar
- janino.jar
- antlr-runtime.jar
- mvel2.jar

16.6.9.2. Configuration

The configuration for `RuleBasedPermissionResolver` requires that a Drools rule base is first configured in `components.xml`. By default, it expects that the rule base is named `securityRules`, as per the following example:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:core="http://jboss.org/schema/seam/core"
            xmlns:security="http://jboss.org/schema/seam/security"
            xmlns:drools="http://jboss.org/schema/seam/drools"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.org/schema/seam/core http://jboss.org/schema/seam/core-2.3.xsd
                          http://jboss.org/schema/seam/components  http://jboss.org/schema/seam/
components-2.3.xsd
                http://jboss.org/schema/seam/drools http://jboss.org/schema/seam/drools-2.3.xsd
             http://jboss.org/schema/seam/security http://jboss.org/schema/seam/security-2.3.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, it's time to write the security rules.

16.6.9.3. 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. Usually this file would be named something like `security.drl`, however you can name it whatever you like as long as it is configured correspondingly in `components.xml`.

So what should the security rules file contain? At this stage it might be a good idea to at least skim through the Drools documentation, however to get started here's an extremely simple example:

```
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
```

Let's break this down step by step. The first thing we see is the package declaration. A package in Drools is essentially a collection of rules. The package name can be anything you want - it doesn't relate to anything else outside the scope of the rule base.

The next thing we can notice is a couple of import statements for the `PermissionCheck` and `Role` classes. These imports inform the rules engine that we'll be referencing these classes within our rules.

Finally we have the code for the rule. Each rule within a package should be given a unique name (usually describing the purpose of the rule). In this case our rule is called

`CanUserDeleteCustomers` and will be used to check whether a user is allowed to delete a customer record.

Looking at the body of the rule definition we can notice two distinct sections. Rules have what is known as a left hand side (LHS) and a right hand side (RHS). The LHS consists of the conditional part of the rule, i.e. a list of conditions which 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 that will only be fired if all of the 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.

If we look at the LHS of the rule, we see two conditions listed there. Let's examine the first condition:

```
c: PermissionCheck(target == "customer", action == "delete")
```

In plain english, this condition is stating that there must exist a `PermissionCheck` object with a `target` property equal to "customer", and an `action` property equal to "delete" within the working memory.

So what is the working memory? Also known as a "stateful session" in Drools terminology, the working memory is a session-scoped object that contains the contextual information that is required by the rules engine 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 that is being checked, so for example if you call `hasPermission("account", "create")` then 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.

Besides the `PermissionCheck` facts, there is also a `org.jboss.seam.security.Role` fact for each of the roles 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 that is inserted into the working memory during the course of a permission check will be removed before the next permission check occurs, if the authenticated user is not actually a member of that role. Besides the `PermissionCheck` and `Role` facts, the working memory also contains the `java.security.Principal` object that was created as a result of the authentication process.

It is also possible to insert additional long-lived facts into the working memory by calling `RuleBasedPermissionResolver.instance().getSecurityContext().insert()`, passing the object as a parameter. The exception to this is `Role` objects, which as already discussed are synchronized at the start of each permission check.

Getting back to our simple example, we can also notice that the first line of our LHS is prefixed with `c:`. This is a variable binding, and is used to refer back to the object that is matched by the condition (in this case, the `PermissionCheck`). Moving on to the second line of our LHS, we see this:

```
Role(name == "admin")
```

This condition simply states that there must be a `Role` object with a `name` of "admin" within the working memory. As already mentioned, user roles are inserted into the working memory at the beginning of each permission check. So, putting both conditions together, this rule is essentially saying "I will fire if you are checking for the `customer:delete` permission and the user is a member of the `admin` role".

So what is the consequence of the rule firing? Let's take a look at the RHS of the rule:

```
c.grant()
```

The RHS consists of Java code, and in this case is invoking the `grant()` method of the `c` object, which as already mentioned is a variable binding for the `PermissionCheck` object. Besides the `name` and `action` properties of the `PermissionCheck` object, there is also a `granted` property which is initially set to `false`. Calling `grant()` on a `PermissionCheck` sets the `granted` property to `true`, which means that the permission check was successful, allowing the user to carry out whatever action the permission check was intended for.

16.6.9.4. Non-String permission targets

So far we have only seen permission checks for String-literal permission targets. It is of course also possible to write security rules for permission targets of more complex types. For example, let's say that you wish to write a security rule to allow your users to create blog comments. The following rule demonstrates how this may be expressed, by requiring the target of the permission check to be an instance of `MemberBlog`, and also requiring that the currently authenticated user is 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
```

16.6.9.5. 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 this:

```
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.

16.6.10. 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 ACL style instance-based security, allowing for specific object permissions to be assigned to individual users and roles. It also allows for persistent, arbitrarily-named permission targets (not necessarily object/class based) to be assigned in the same way.

16.6.10.1. Configuration

Before it can be used, `PersistentPermissionResolver` must be configured with a valid `PermissionStore` in `components.xml`. If not configured, it will attempt to use the default permission store, `JpaIdentityStore` (see section further down for details). To use a permission store other than the default, configure the `permission-store` property as follows:

```
<security:persistent-permission-resolver permission-store="#{myCustomPermissionStore}"/>
```

16.6.10.2. Permission Stores

A permission store is required for `PersistentPermissionResolver` to connect to the backend storage where permissions are persisted. Seam provides one `PermissionStore` implementation out of the box, `JpaPermissionStore`, which is used to store permissions inside a relational database. It is possible to write your own permission store by implementing the `PermissionStore` interface, which defines the following methods:

Table 16.8. 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.
`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 backend storage, returning true if successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	This method should persist all of the `Permission` objects contained in the specified `List`, returning true if successful.
`boolean`	`revokePermission(Permission permission)`	This method should remove the specified `Permission` object from persistent storage.

Return type	Method	Description
`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 the 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 (see section further down).

16.6.10.3. JpaPermissionStore

This is the default `PermissionStore` implementation (and the only one provided by Seam), which uses a relational database to store permissions. Before it can be used it must be configured with either one or two entity classes for storing user and role permissions. These entity classes must be annotated with a special set of security annotations to configure which properties of the entity correspond to various aspects of the permissions being stored.

If you wish to use the same entity (i.e. a single database table) to store both user and role permissions, then only the `user-permission-class` property is required to be configured. If you wish to use separate tables for storing user and role permissions, then in addition to the `user-permission-class` property 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" />
```

16.6.10.3.1. Permission annotations

As mentioned, the entity classes that contain the user and role permissions must be configured with a special set of annotations, contained within the `org.jboss.seam.annotations.security.permission` package. The following table lists each of these annotations along with a description of how they are used:

Table 16.9. Entity Permission annotations

Annotation	Target	Description
`@PermissionTarget`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the permission target. The property should be of type `java.lang.String`.
`@PermissionAction`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the permission action. The property should be of type `java.lang.String`.
`@PermissionUser`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain 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 property of the entity that will contain the recipient role for the permission. It should be of type `java.lang.String` and contain the role name.
`@PermissionDiscriminator`	`FIELD,METHOD`	This annotation should be used when the same entity/table is used to store both user and role permissions. It identifies the property of the entity that is 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`, then it will be treated as a role permission. It is also possible to 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`, the annotation would be written like this:

```
@PermissionDiscriminator
```

Annotation	Target	Description
		(userValue="u", roleValue="r")

16.6.10.3.2. Example Entity

Here is an example of an entity class that is used to store both user and role permissions. The following class can be found inside 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;
  }
}
```

As can be seen in the above example, the `getDiscriminator()` method has been annotated with the `@PermissionDiscriminator` annotation, to allow `JpaPermissionStore` to determine which records represent user permissions and which represent role permissions. In addition, it can also be seen that the `getRecipient()` method is annotated with both `@PermissionUser` and `@PermissionRole` annotations. This is perfectly valid, and simply 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.

16.6.10.3.3. Class-specific Permission Configuration

A further set of class-specific annotations can be used to configure a specific set of allowable permissions for a target class. These permissions can be found in the `org.jboss.seam.annotation.security.permission` package:

Table 16.10. Class Permission Annotations

Annotation	Target	Description
`@Permissions`	`TYPE`	A container annotation, this annotation may 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

Annotation	Target	Description
		specified if permission actions are to be persisted as bitmasked values (see next section).

Here's an example of the above annotations in action. The following class can also be found 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`.

16.6.10.3.4. 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 comma-separated list of the granted actions. To reduce the amount of physical storage required to persist a large number of permissions, it is possible to use a bitmasked integer value (instead of a comma-separated list) to store the list of permission actions.

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. Alternatively, if using bitmasked values for the permission actions, as defined like so:

```
@Permissions({
  @Permission(action = "view", mask = 1),
  @Permission(action = "comment", mask = 2)
})
@Entity
public class MemberImage implements Serializable {
```

The `action` property will instead simply contain "3" (with both the 1 bit and 2 bit switched on). Obviously 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.

Obviously, it is very important that the `mask` values specified are powers of 2.

16.6.10.3.5. Identifier Policy

When storing or looking up permissions, `JpaPermissionStore` must be able to uniquely identify specific object instances to effectively operate on its permissions. To achieve this, an *identifier strategy* may be assigned to each target class for the generation of unique identifier values. Each identifier strategy implementation knows how to generate unique identifiers for a particular type of class, and it is a simple matter to create new identifier strategies.

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()` simply 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 provides two `IdentifierStrategy` implementations, `ClassIdentifierStrategy` and `EntityIdentifierStrategy` (see next sections for details).

To explicitly configure a specific identifier strategy to use for a particular class, it should be annotated with `org.jboss.seam.annotations.security.permission.Identifier`, and the value should be set to a concrete implementation of the `IdentifierStrategy` interface. An optional `name` property can also be specified, the effect of which is dependent upon the actual `IdentifierStrategy` implementation used.

16.6.10.3.6. ClassIdentifierStrategy

This identifier strategy is used to generate unique identifiers for classes, and will use the value of the `name` (if specified) in the `@Identifier` annotation. If there is no `name` property provided, then it will attempt to use the component name of the class (if the class is a Seam component), or as a last resort it will create an identifier based on the name of the class (excluding the package name). 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 {
```

16.6.10.3.7. EntityIdentifierStrategy

This identifier strategy is used to generate unique identifiers for entity beans. It does so by concatenating 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 `ClassIdentifierStrategy`. The primary key value (i.e. the *id* of the entity) is obtained using the `PersistenceProvider` component, which is able to correctly determine the value regardless of which persistence implementation is used within the Seam application. For entities not annotated with `@Entity`, it is necessary to explicitly configure the identifier strategy on the entity class itself, for example:

```
@Identifier(value = EntityIdentifierStrategy.class)
public class Customer {
```

For an example of the type of identifier values generated, 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`".

16.7. Permission Management

In much the same way that Seam Security provides an Identity Management API for the management of users and roles, it also provides a Permissions Management API for the management of persistent user permissions, via the `PermissionManager` component.

16.7.1. PermissionManager

The `PermissionManager` component is an application-scoped Seam component that provides a number of methods for managing permissions. Before it can be used, it must be configured with a permission store (although by default it will attempt to use `JpaPermissionStore` if it is available). To explicitly 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 available methods provided by `PermissionManager`:

Table 16.11. 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.
`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 backend

Return type	Method	Description
		permission store. Returns true if the operation was successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	Persists (grants) the specified list of `Permissions` to the backend permission store. Returns true if the operation was successful.
`boolean`	`revokePermission(Permission permission)`	Removes (revokes) the specified `Permission` from the backend permission store. Returns true if the operation was successful.
`boolean`	`revokePermissions(List<Permission> permissions)`	Removes (revokes) the specified list of `Permissions` from the backend permission store. Returns true if the operation was successful.
`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.

16.7.2. Permission checks for PermissionManager operations

Invoking the methods of `PermissionManager` requires that the currently-authenticated user has the appropriate authorization to perform that management operation. The following table lists the required permissions that the current user must have.

Table 16.12. 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 `Permissions` (depending on which method is called).	`seam.grant-permission`
`grantPermission()`	The target of the specified `Permission`.	`seam.grant-permission`

Method	Permission Target	Permission Action
`grantPermissions()`	Each of the targets of the specified list of `Permissions`.	`seam.grant-permission`
`revokePermission()`	The target of the specified `Permission`.	`seam.revoke-permission`
`revokePermissions()`	Each of the targets of the specified list of `Permissions`.	`seam.revoke-permission`

16.8. SSL Security

Seam includes basic support for serving sensitive pages via the HTTPS protocol. This is easily configured by specifying a `scheme` for the page in `pages.xml`. The following example shows how the view `/login.xhtml` is configured to use HTTPS:

```
<page view-id="/login.xhtml" scheme="https"/>
```

This configuration is automatically extended to both `s:link` and `s:button` JSF controls, which (when specifying the `view`) will also render the link using 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"/>
```

Browsing directly to a view when using the *incorrect* protocol will cause a redirect to the same view using the *correct* protocol. For example, browsing to a page that has `scheme="https"` using HTTP will cause a redirect to the same page using HTTPS.

It is also possible to configure a *default scheme* for all pages. This is useful if you wish to use HTTPS for a only few pages. If no default scheme is specified then the normal behavior is to continue use the current scheme. So once the user accessed a page that required HTTPS, then HTTPS would continue to be used after the user navigated away to other non-HTTPS pages. (While this is good for security, it is not so great for performance!). To define HTTP as the default `scheme`, add this line to `pages.xml`:

```
<page view-id="*" scheme="http" />
```

Of course, if *none* of the pages in your application use HTTPS then it is not required to specify a default scheme.

You may configure Seam to automatically invalidate the current HTTP session each time the scheme changes. Just add this line to `components.xml`:

```
<web:session invalidate-on-scheme-change="true"/>
```

This option helps make your system less vulnerable to sniffing of the session id or leakage of sensitive data from pages using HTTPS to other pages using HTTP.

16.8.1. Overriding the default ports

If you wish to configure the HTTP and HTTPS ports manually, they may be configured in `pages.xml` by specifying the `http-port` and `https-port` attributes on the `pages` element:

```
<pages xmlns="http://jboss.org/schema/seam/pages"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.org/schema/seam/pages http://jboss.org/schema/seam/
pages-2.3.xsd"
      no-conversation-view-id="/home.xhtml"
      login-view-id="/login.xhtml"
      http-port="8080"
      https-port="8443">
```

16.9. CAPTCHA

Though strictly not part of the security API, Seam provides a built-in CAPTCHA (*C*ompletely *A*utomated *P*ublic *T*uring test to tell *C*omputers and *H*umans *A*part) algorithm to prevent automated processes from interacting with your application.

16.9.1. Configuring the CAPTCHA Servlet

To get up and running, it is necessary to configure the Seam Resource Servlet, which will provide the Captcha challenge images to your pages. This 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>
```

16.9.2. Adding a CAPTCHA to a form

Adding a CAPTCHA challenge to a form is extremely easy. Here's an example:

```
<h:graphicImage value="/seam/resource/captcha"/>
<h:inputText id="verifyCaptcha" value="#{captcha.response}" required="true">
  <s:validate />
</h:inputText>
<h:message for="verifyCaptcha"/>
```

That's all there is to it. 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.

16.9.3. Customising the CAPTCHA algorithm

You may 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;
  }
}
```

16.10. 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 16.13. Security Events

Event Key	Description
`org.jboss.seam.security.loginSuccessful`	Raised when a login attempt is successful.
`org.jboss.seam.security.loginFailed`	Raised when a login 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.
`org.jboss.seam.security.notAuthorized`	Raised when a security check fails when the user is logged in however doesn't 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.

16.11. Run As

Sometimes it may be necessary to perform certain operations with elevated privileges, such as creating a new user account as an unauthenticated user. Seam Security supports such a mechanism via 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 example demonstrates how `RunAsOperation` is used, by calling its `addRole()` method to provide a set of roles to masquerade as 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();
```

In a similar way, the `getPrincipal()` or `getSubject()` methods can also be overriden 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`.

16.12. Extending the Identity component

Sometimes it might be necessary to extend the Identity component if your application has special security requirements. The following example (contrived, as credentials would normally be handled by the `Credentials` component instead) shows an extended Identity component with an additional `companyCode` field. The install precedence of `APPLICATION` ensures that this extended Identity gets installed in preference to 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

Note that 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.

16.13. OpenID

OpenID is a community standard for external web-based authentication. The basic idea is that any web application can supplement (or replace) its local handling of authentication by delegating responsibility to an external OpenID server of the user's choose. This benefits the user, who no longer has to remember a name and password for every web application he uses, and the developer, who is relieved of some of the burden of maintaining a complex authentication system.

When using OpenID, the user selects an OpenID provider, and the provider assigns the user an OpenID. The id will take the form of a URL, for example `http://maximoburrito.myopenid.com` however, it's acceptable to leave off the `http://` part of the identifier when logging into a site. The web application (known as a relying party in OpenID-speak) determines which OpenID server to contact and redirects the user to the remote site for authentication. Upon successful authentication 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 be sure the user accessing the application controls the OpenID he presented.

It's important to realize at this point that authentication does not imply authorization. The web application still needs to make a determination of how to use that information. The web application could treat the user as instantly logged in and give full access to the system or it could try and map the presented OpenID to a local user account, prompting the user to register if he hasn't already. The choice of how to handle the OpenID is left as a design decision for the local application.

16.13.1. Configuring OpenID

Seam uses the openid4java package and requires four additional JARs to make use of the Seam integration. These are: `htmlparser.jar`, `openid4java.jar`, `openxri-client.jar` and `openxri-syntax.jar`.

OpenID processing requires the use of 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>
```

With this configuration, OpenID support is 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.

16.13.2. Presenting an OpenIdDLogin form

To initiate an OpenID login, you can present a simply form to the user asking for 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, he will be redirected to his OpenID provider. The user will eventually return to your application through the Seam pseudo-view `/openid.xhtml`, which is provided by the `OpenIdPhaseListener`. Your application can handle the OpenID response by means of a `pages.xml` navigation from that view, just as if the user had never left your application.

16.13.3. Logging in immediately

The simplest strategy is to simply login the user 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>
```

This `loginImmediately()` action checks to see if the OpenID is valid. If it is valid, it adds an OpenIDPrincipal to the identity component, marks the user as logged in (i.e. `#{identity.loggedIn}` will be true) and returns true. If the OpenID was 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.

16.13.4. Deferring login

You may not want the user to be immediately logged in to your application. In that case, your navigation should check the `#{openid.valid}` property and redirect the user to a local registration or processing page. Actions you might take would be asking for more information and creating a local user account or presenting a captcha to avoid programmatic registrations. When you are done processing, if you want to log the user in, you can call the `loginImmediately` method, either through EL as shown previously or by directly interaction with the `org.jboss.seam.security.openid.OpenId` component. Of course, nothing prevents you from writing custom code to interact with the Seam identity component on your own for even more customized behaviour.

16.13.5. Logging out

Logging out (forgetting an OpenID association) is done by calling `#{openid.logout}`. If you are not using Seam security, you can call this method directly. If you are using Seam security, you should continue to use `#{identity.logout}` and install an event handler to capture the logout event, calling the OpenID logout method.

```
<event type="org.jboss.seam.security.loggedOut">
   <action execute="#{openid.logout}" />
</event>
```

It's important that you do not leave this out or the user will not be able to login again in the same session.

Internationalization, localization and themes

Seam makes it easy to build internationalized applications. First, let's walk through all the stages needed to internationalize and localize your app. Then we'll take a look at the components Seam bundles.

17.1. Internationalizing your app

A JEE application consists of many components and all of them must be configured properly for your application to be localized.

Note

Note that all i18n features in Seam work only in JSF context.

Starting at the bottom, the first step is to ensure that your database server and client is using the correct character encoding for your locale. Normally you'll want to use UTF-8. How to do this is outside the scope of this tutorial.

17.1.1. Application server configuration

To ensure that the application server receives the request parameters in the correct encoding from client requests you have to configure the tomcat connector. If you use JBoss AS, add the system properties org.apache.catalina.connector.URI_ENCODING and org.apache.catalina.connector.USE_BODY_ENCODING_FOR_QUERY_STRING to the server configuration. For JBoss AS 7.1.1 change `${JBOSS_HOME}/standalone/configuration/standalone.xml`:

```
<system-properties>
    <property name="org.apache.catalina.connector.URI_ENCODING" value="UTF-8"/>
    <property name="org.apache.catalina.connector.USE_BODY_ENCODING_FOR_QUERY_STRING" value="true"/>
</system-properties>
```

17.1.2. Translated application strings

You'll also need localized strings for all the *messages* in your application (for example field labels on your views). First you need to ensure that your resource bundle is encoded using the desired character encoding. By default ASCII is used. Although ASCII is enough for many languages, it doesn't provide characters for all languages.

Resource bundles must be created in ASCII, or use Unicode escape codes to represent Unicode characters. Since you don't compile a property file to byte code, there is no way to tell the JVM which character set to use. So 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 using \uXXXX, where XXXX is the hexadecimal representation of the character.

You can write your translation of labels (*Section 17.3, "Labels"*) to your messages resource bundle in the native encoding and then convert the content of the file into the escaped format through the tool `native2ascii` provided in the JDK. This tool will convert a file written in your native encoding to one that represents non-ASCII characters as Unicode escape sequences.

Usage of this tool is described *here for Java 5* [http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#intl] or *here for Java 6* [http://java.sun.com/javase/6/docs/technotes/tools/#intl]. For example, to convert a file from UTF-8:

```
$ native2ascii -encoding UTF-8 messages_cs.properties >
 messages_cs_escaped.properties
```

17.1.3. Other encoding settings

We need to make sure that the view displays your localized data and messages using the correct character set and also any data submitted uses the correct encoding.

To set the display character encoding, you need to use the `<f:view locale="cs_CZ"/>` tag (here we tell JSF to use the Czech locale). You may want to change the encoding of the xml document itself if you want to embed localized strings in the xml. To do this alter the encoding attribute in xml declaration `<?xml version="1.0" encoding="UTF-8"?>` as required.

Also JSF/Facelets should submit any requests using the specified character encoding, but to make sure any requests that don't specify an encoding 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" />
```

17.2. Locales

Each user login session has an associated instance of `java.util.Locale` (available to the application as a component named `locale`). Under normal circumstances, you won't need to do any special configuration to set the locale. Seam just delegates to JSF to determine the active locale:

- If there is a locale 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.

It is *possible* to set the locale manually via the Seam configuration properties `org.jboss.seam.international.localeSelector.language`, `org.jboss.seam.international.localeSelector.country` and `org.jboss.seam.international.localeSelector.variant`, but we can't think of any good reason to ever do this.

It is, however, useful to allow the user to set the locale manually via the application user interface. Seam provides built-in functionality for overriding the locale determined by the algorithm above. All you have to do is add 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`, just 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.

The brings us to the question of where the supported locales are defined. Typically, you provide a list of locales for which you have matching resource bundles in the `<locale-config>` element of the JSF configuration file (/META-INF/faces-config.xml). However, you have learned to appreciate that Seam's component configuration mechanism is more powerful than what is provided in Java EE. For that reason, you can configure the supported locales, and the default locale of the server, using the built-in component named `org.jboss.seam.international.localeConfig`. To use it, you first declare an XML namespace for Seam's international package in the Seam component descriptor. You then define the default locale and supported locales as follows:

```
<international:locale-config default-locale="fr_CA" supported-locales="en fr_CA fr_FR"/>
```

Naturally, if you pronounce that you support a locale, you better provide a resource bundle to match it! Up next, you'll learn how to define the language-specific labels.

17.3. Labels

JSF supports internationalization of user interface labels and descriptive text via the use of `<f:loadBundle />`. You can use this approach in Seam applications. Alternatively, you can take advantage of the Seam `messages` component to display templated labels with embedded EL expressions.

17.3.1. Defining labels

Seam provides a `java.util.ResourceBundle` (available to the application as a `org.jboss.seam.core.resourceBundle`). You'll need to make your internationalized labels available via this special resource bundle. By default, the resource bundle used by Seam is named `messages` and so you'll 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>
```

If you want to define a message just for a particular page, you can 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`.

17.3.2. Displaying labels

If you define your labels using the Seam resource bundle, you'll be able to use them without having to type `<f:loadBundle ... />` on every page. Instead, you can simply 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;
```

17.3.3. Faces messages

The `facesMessages` component is a super-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.

17.4. 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 says that all dates and times should be assumed to be UTC, and displayed as UTC, unless a timezone is explicitly specified using `<f:convertDateTime>`. This is an extremely inconvenient default behavior.

Note

You can use application parameter to set up different default time zone for JSF 2 in `web.xml`.

```
<context-param>
                                        <param-
name>javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE</
param-name>
```

```
    <param-value>true</param-value>
</context-param>
```

Seam overrides this behavior, and defaults all dates and times to 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 are simply capturing a date. The pattern is selected according the the user's locale and the time zone is selected as described above.

17.5. 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>
```

Note that 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`. For example, `default.properties` might define:

```
css ../screen.css
template /template.xhtml
```

Usually the entries in a theme resource bundle will be 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, when 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"/>
```

17.6. 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"/>
```

Seam Text

Collaboration-oriented websites require a human-friendly markup language for easy entry of formatted text in forum posts, wiki pages, blogs, comments, etc. Seam provides the `<s:formattedText/>` control for display of formatted text that conforms to the *Seam Text* language. Seam Text is implemented using an ANTLR-based parser. You don't need to know anything about ANTLR to use it, however.

18.1. Basic fomatting

Here is a simple example:

```
It's easy to make *emphasis*, |monospace|,
~deleted text~, super^scripts^ or _underlines_.
```

If we display this using `<s:formattedText/>`, we will get the following HTML produced:

```
<p>
It's easy to make <i>emphasis</i>, <tt>monospace</tt>
<del>deleted text</del>, super<sup>scripts</sup> or <u>underlines</u>.
</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.
```

(Note that a simple newline 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>
```

Ordered lists are created using the # character. 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:

```
The other guy said:

"Nyeah nyeah-nee
/nyeah/ nyeah!"

But what do you think he means by "nyeah-nee"?
```

```
<p>
The other guy said:
</p>

<q>Nyeah nyeah-nee
<i>nyeah</i> nyeah!</q>

<p>
But what do you think he means by <q>nyeah-nee</q>?
</p>
```

18.2. Entering code and text with special characters

Special characters such as `*`, `|` and `#`, along with HTML characters such as `<`, `>` and `&` may be escaped using `\`:

```
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 using backticks:

```
My code doesn't work:

`for (int i=0; i<100; i--)
{
   doSomething();
}`

Any ideas?
```

```
<p>
My code doesn't work:
</p>

<pre>for (int i=0; i&lt;100; i--)
{
   doSomething();
}</pre>

<p>
Any ideas?
</p>
```

Note that inline monospace formatting always escapes (most monospace formatted text is in fact code or tags with many special characters). So you can, for example, write:

```
This is a |<tag attribute="value"/>| example.
```

without escaping any of the characters inside the monospace bars. The downside is that you can't format inline monospace text in any other way (italics, underscore, and so on).

18.3. Links

A link may be created using the following syntax:

```
Go to the Seam website at [=>http://jboss.org/schema/seam].
```

Or, if you want to specify the text of the link:

```
Go to [the Seam website=>http://jboss.org/schema/seam].
```

For advanced users, it is even possible to customize the Seam Text parser to understand wikiword links written using this syntax.

18.4. Entering HTML

Text may even include a certain limited subset of HTML (don't worry, the subset is chosen to be safe from cross-site scripting attacks). This is useful for creating links:

```
You might want to link to <a href="http://jboss.org/schema/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>
```

But you can do much more if you want!

18.5. Using the SeamTextParser

The `<s:formattedText/>` JSF component internally uses the `org.jboss.seam.text.SeamTextParser`. You can use that class directly and implement your own text parsing, rendering, or HTML sanitation procedure. This is especially useful if you have a custom frontend for entering rich text, such as a Javascript-based HTML editor, and you want to validate user input to protect your website against Cross-Site Scripting (XSS) attacks. Another usecase are custom wiki text parsing and rendering engines.

The following example defines a custom text parser that 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=>http://example.com]
@Override
protected String linkTag(String descriptionText, String linkText) {
    return "<a href=\"" + linkText + "\">My Custom Link: " + descriptionText + "</a>";
}

// Renders a <p> or equivalent tag
@Override
protected String paragraphOpenTag() {
    return "<p class=\"myCustomStyle\">";
}

public void parse() throws ANTLRException {
    startRule();
}

}
```

The `linkTag()` and `paragraphOpenTag()` methods are just some of many you can override to customize rendered output. These methods generally return `String`. See the Javadoc for more details.

Also consult the Javadoc of `org.jboss.seam.text.SeamTextParser.DefaultSanitizer` for more information on what HTML elements, attributes, and attribute values or filtered by default.

iText PDF generation

Seam now includes a component set for generating documents using 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.

19.1. Using PDF Support

iText support is provided by `jboss-seam-pdf.jar`. This JAR contains the iText JSF controls, which are used to 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, put `jboss-seam-pdf.jar` in your `WEB-INF/lib` directory along with the iText JAR file. There is no further configuration needed to use Seam's iText support.

The Seam iText module requires the use of Facelets as the view technology. Future versions of the library may also support the use of JSP. Additionally, it 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 it contains a number examples that demonstrate the key PDF generation features current supported.

19.1.1. Creating a document

`<p:document>`	*Description* Documents are generated by facelet XHTML files using tags in the `http://jboss.org/schema/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` modes. Seam defaults to PDF generation, and many of the features only work correctly when generating PDF documents. • `pageSize` — The size of the page to be generate. The most commonly used values would be `LETTER` and `A4`. A full list of supported pages sizes can be found in `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`. In landscape mode, the height and width page size values are reversed.
- `margins` — The left, right, top and bottom margin values.
- `marginMirroring` — Indicates that margin settings should be reversed an 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.org/schema/seam/pdf">

  The document goes here.

</p:document>
```

19.1.2. Basic Text Elements

Useful documents will need to contain more than just text; however, the standard UI components are geared towards HTML generation and are not useful for generating PDF content. Instead, Seam provides a special UI components for generating suitable PDF content. Tags like `<p:image>` and `<p:paragraph>` are the basic foundations of simple documents. Tags like `<p:font>` provide style information to all the content surrounding them.

`<p:paragraph>`	*Description*
	
	Most uses of text should be sectioned into paragraphs so that text fragments can be flowed, formatted and styled in logical groups.
	
	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 isn't very fancy.`
	`</p:paragraph>`

`<p:text>`	*Description*
	
	The `text` tag allows text fragments to be produced from application data 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 will typically be 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`

	• `color` — The font color. (see *Section 19.1.7.1, "Color Values"* for color values) • `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 binding 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 19.1.7.2, "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.
- `scaleToFit` — Specifies the X any Y size to scale the image to. The image will be scale to fit those dimensions as closely as possible while preserving the XY ratio of the image.
- `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`. Links may also be a full URL to point to a resource outside of the document.
	
	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>`

19.1.3. Headers and Footers

`<p:header>`
 	
`<p:footer>`
	Description
	
	The `p:header` and `p:footer` components provide the ability to 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 19.1.7.2, "Alignment Values"* for alignment values)
- `backgroundColor` — The background color of the header/footer box. (see *Section 19.1.7.1, "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 19.1.7.1, "Color Values"* for color values)
- `borderWidth` — The width of the border. Individual border sides can be specified using `borderWidthLeft`, `borderWidthRight`, `borderWidthTop` and `borderWidthBottom`.

Usage

```
<f: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 of a header or footer using 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>
```

19.1.4. 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 the necessary

structure. Sections can only be used inside of chapters, but they may be nested arbitrarily deep. Most PDF viewers provide easy navigation between chapters and sections in a document.

Note

You cannot include a chapter into another chapter, this can be done only with section(s).

Attributes

- `alignment` — The alignment of the header/footer box section. (see *Section 19.1.7.2, "Alignment Values"* for alignment values)
- `number` — The chapter/section number. Every chapter/section should be assigned a number.
- `numberDepth` — The depth of numbering for chapter/section. All sections are numbered relative to their surrounding chapter/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 three. To omit the chapter number, a number depth of 2 should be used. In that case, the section number would be displayed as 1.4.

Note

Chapter(s) can have a number or without it by setting numberDepth to 0.

Usage

```
<p:document xmlns:p="http://jboss.org/schema/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/section number. The body of the title may contain raw text or may be a `p:paragraph`.

19.1.5. Lists

List structures can be displayed using 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.org/schema/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 list. One of: `NUMBERED`, `LETTERED`, `GREEK`, `ROMAN`, `ZAPFDINGBATS`, `ZAPFDINGBATS_NUMBER`. If no style is given, the list items are bulleted. • `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 list item. (See *Section 19.1.7.2, "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* ...

19.1.6. 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 3 columns, then every 3 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 3 columns and the first column should be twice the size of the second and third column.

- `headerRows` — The initial number of rows which are considered to be headers or footer rows and should be repeated if the table spans multiple pages.
- `footerRows` — The number of rows that are considered to be footer rows. This value is subtracted from the `headerRows` value. If document has 2 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 that the table spans.
- `horizontalAlignment` — The horizontal alignment of the table. (See *Section 19.1.7.2, "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 1. Tables do not have the ability to span across multiple rows.
- `horizontalAlignment` — The horizontal alignment of the cell. (see *Section 19.1.7.2, "Alignment Values"* for possible values)
- `verticalAlignment` — The vertical alignment of the cell. (see *Section 19.1.7.2, "Alignment Values"* for possible values)
- `padding` — Padding on a given side can also be specified 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>`

19.1.7. Document Constants

This section documents some of the constants shared by attributes on multiple tags.

19.1.7.1. Color Values

Several ways of specifying colors are provided. A limited number of colors are supported by name. They are: `white`, `gray`, `lightgray`, `darkgray`, `black`, `red`, `pink`, `yellow`, `green`, `magenta`, `cyan` and `blue`. Colors can be specified as an integer value, as defined by `java.awt.Color`. Finally a color value may be specified as `rgb(r,g,b)` or `rgb(r,g,b,a)` with the red, green, blue alpha values specified as an integer between 0 and 255 or as a floating point percentages followed by a '%' sign.

19.1.7.2. Alignment Values

Where alignment values are used, the Seam PDF supports the following horizontal alignment values: `left`, `right`, `center`, `justify` and `justifyall`. The vertical alignment values are `top`, `middle`, `bottom`, and `baseline`.

19.2. Charting

Charting support is also provided with `jboss-seam-pdf.jar`. Charts can be used in PDF documents or can be used as images in an HTML page. Charting requires the JFreeChart library (`jfreechart.jar` and `jcommon.jar`) to be added to the `WEB-INF/lib` directory. Four types of charts are currently supported: pie charts, bar charts and line charts. Where greater variety or control is needed, it is possible to construct charts using Java code.

`<p:chart>`	*Description* Displays a chart created in Java by a Seam component. *Attributes* • `chart` — The chart object to display. • `height` — The height of 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* • `chart` — The chart object to display, if programmatic chart creation is being used. • `dataset` — The dataset to be displayed, if programmatic dataset is being used. • `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`. Alternatively, the value can the 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: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* • `chart` — The chart object to display, if programmatic chart creation is being used. • `dataset` — The dataset to be displayed, if programmatic dataset is being used. • `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`. Alternatively, the value can the 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. • `chart` — The chart object to display, if programmatic chart creation is being used. • `dataset` — The dataset to be displayed, if programmatic dataset is being used. • `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` — he 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 the pie section 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 set of data 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 if 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 a 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 than can be referenced when drawing filled shapes. *Attributes* • `color` — The color value. For gradient colors, this the starting color. *Section 19.1.7.1, "Color Values"* • `color2` — For gradient colors, this is the color that ends the gradient. • `point` — The co-ordinates where the gradient color begins. • `point2` — The co-ordinates where the gradient color ends. *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 to be used to draw the line. The space separated integers indicate the length of each alternating drawn and undrawn segments. • `dashPhase` — The dash phase indicates the offset into the dash pattern that the line should be drawn with. *Usage* `<p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" />`

19.3. 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. Note that when used with HTML images, barcodes can not currently display barcode text in the barcode.

`<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 y 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 to draw the bars.
- `textColor` — The color of any text on the barcode.
- `textSize` — The size of the barcode text, if any.
- `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" />
```

19.4. Fill-in-forms

If you have a complex, pre-generated PDF with named fields, you can easily fill in the values from your application and present it to the user.

`<p:form>`

Description

Defines a form template to populate

Attributes

- `URL` — An URL pointing to the PDF file to use as a template. If the value has no protocol part (://), the file is read locally.
- `filename` — The filename to use for the generated PDF file.

	• `exportKey` — Place the generated PDF file in a DocumentData object under the specified key in the event context. If set, no redirect will occur.

`<p:field>`	*Description* Connects a field name to its value *Attributes* • `name` — The name of the field • `value` — The value of the field • `readOnly` — Should the field be read-only? Defaults to true.

```
<p:form
   xmlns:p="http://jboss.org/schema/seam/pdf"
   URL="http://localhost/Concept/form.pdf">
   <p:field name="person.name" value="Me, myself and I"/>
</p:form>
```

19.5. Rendering Swing/AWT components

Seam now provides experimental support for rendering Swing components into a PDF image. Some Swing look and feels supports, notably ones 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}" />
```

19.6. Configuring iText

Document generation works out of the box with no additional configuration needed. However, there are a few points of configuration that are needed for more serious applications.

The default implementation serves PDF documents from a generic URL, `/seam-doc.seam`. Many browsers (and users) would prefer to see URLs that contain the actual PDF name like `/myDocument.pdf`. This capability requires some configuration. To serve PDF files, all `*.pdf` resources should be mapped to the DocumentStoreServlet:

```
<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>*.pdf</url-pattern>
</servlet-mapping>
```

The `use-extensions` option on the document store component completes the functionality by instructing the document store to generate URLs with the correct filename extension for the document type being generated.

```
<components xmlns="http://jboss.org/schema/seam/components"
   xmlns:document="http://jboss.org/schema/seam/document"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://jboss.org/schema/seam/document http://jboss.org/schema/seam/document-2.3.xsd
                          http://jboss.org/schema/seam/components    http://jboss.org/schema/seam/
components-2.3.xsd">
   <document:document-store use-extensions="true"/>
</components>
```

The document store stores documents in conversation scope, and documents will expire when the conversation ends. At that point, references to the document will be invalid. You can specify a default view to be shown when a document does not exist using the `error-page` property of the `documentStore`.

```
<document:document-store use-extensions="true" error-page="/documentMissing.seam" />
```

19.7. Further documentation

For further information on iText, see:

- *iText Home Page* [http://www.lowagie.com/iText/]
- *iText in Action* [http://www.manning.com/lowagie/]

The Microsoft® Excel® spreadsheet application

Seam also supports generation of the Microsoft® Excel® spreadsheet application spreadsheets through the excellent *JExcelAPI* [http://jexcelapi.sourceforge.net/] library. The generated document is compatible with the Microsoft® Excel® spreadsheet application versions 95, 97, 2000, XP and 2003. Currently a limited subset of the library functionality is exposed but the ultimate goal is to be able to do everything the library allows for. Please refer to the JExcelAPI documentation for more information on capabilities and limitations.

20.1. The Microsoft® Excel® spreadsheet application support

The Microsoft® Excel® spreadsheet application `jboss-seam-excel.jar`. This JAR contains the the Microsoft® Excel® spreadsheet application JSF controls, which are used to construct views that can render the document, and the DocumentStore component, which serves the rendered document to the user. To include the Microsoft® Excel® spreadsheet application support in your application, include `jboss-seam-excel.jar` in your `WEB-INF/lib` directory along with the `jxl.jar` JAR file. Furthermore, you need to configure the DocumentStore servlet in your web.xml

The Microsoft® Excel® spreadsheet application Seam module requires the use of Facelets as the view technology. Additionally, it requires the use of the seam-ui package.

The `examples/excel` project contains an example of the Microsoft® Excel® spreadsheet application support in action. It demonstrates proper deployment packaging, and it shows the exposed functionality.

Customizing the module to support other kinds of the Microsoft® Excel® spreadsheet application spreadsheet API's has been made very easy. Implement the `ExcelWorkbook` interface, and register in components.xml.

```
<excel:excelFactory>
  <property name="implementations">
    <key>myExcelExporter</key>
    <value>my.excel.exporter.ExcelExport</value>
  </property>
</excel:excelFactory>
```

and register the excel namespace in the components tag with

```
xmlns:excel="http://jboss.org/schema/seam/excel"
```

Then set the UIWorkbook type to `myExcelExporter` and your own exporter will be used. Default is "jxl", but support for CSV has also been added, using the type "csv".

See *Section 19.6, "Configuring iText"* for information on how to configure the document servlet for serving the documents with an .xls extension.

If you are having problems accessing the generated file under IE (especially with https), make sure you are not using too strict restrictions in the browser, too strict security constraint in web.xml or a combination of both.

20.2. Creating a simple workbook

Basic usage of the worksheet support is simple; it is used like a familiar `<h:dataTable>` and you can bind to a `List`, `Set`, `Map`, `Array` or `DataModel`.

```
<e:workbook xmlns:e="http://jboss.org/schema/seam/excel">
  <e:worksheet>
    <e:cell column="0" row="0" value="Hello world!"/>
  </e:worksheet>
</e:workbook>
```

That's not terribly useful, so lets have a look at a more common case:

```
<e:workbook xmlns:e="http://jboss.org/schema/seam/excel">
  <e:worksheet value="#{data}" var="item">
    <e:column>
      <e:cell value="#{item.value}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

First we have the top-level workbook element which serves as the container and it doesn't have any 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. Nested inside the worksheet is a single column and within it you see the cell which is the final bind to the data within the currently iterated item

This is all you know to get started dumping your data to worksheets!

20.3. Workbooks

Workbooks are the top-level parents of worksheets and stylesheet links.

`<e:workbook>`	*Attributes* • `type` — Defines which export module to be used. The value is a string and can be either "jxl" or "csv". The default is "jxl". • `templateURI` — A template that should be used as a basis for the workbook. The value is a string (URI). • `arrayGrowSize` — The amount of memory by which to increase the amount of memory allocated to storing the workbook data. For processes reading many small workbooks inside a WAS it might be necessary to reduce the default size. Default value is 1 megabyte. The value is a number (bytes). • `autoFilterDisabled` — Should autofiltering be disabled?. The value is a boolean. • `cellValidationDisabled` — Should cell validation be ignored? The value is a boolean. • `characterSet` — The character set. This is only used when the spreadsheet is read, and has no effect when the spreadsheet is written. The value is a string (character set encoding). • `drawingsDisabled` — Should drawings be disabled? The value is a boolean. • `excelDisplayLanguage` — The language in which the generated file will display. The value is a string (two character ISO 3166 country code). • `excelRegionalSettings` — The regional settings for the generated excel file. The value is a string (two character ISO 3166 country code). • `formulaAdjust` — Should formulas be adjusted? The value is a boolean.

- `gcDisabled` — Should garbage collection be disabled? The value is a boolean.
- `ignoreBlanks` — Should blanks be ignored? The value is a boolean.
- `initialFileSize` — The initial amount of memory allocated to store the workbook data when reading a worksheet. For processes reading many small workbooks inside a WAS it might be necessary to reduce the default size. Default value is 5 megabytes. The value is a number (bytes).
- `locale` — The locale used by JExcelApi to generate the spreadsheet. Setting this value has no effect on the language or region of the generated excel file. The value is a string.
- `mergedCellCheckingDisabled` — Should merged cell checking be disabled? The value is a boolean.
- `namesDisabled` — Should handling of names be disabled? The value is a boolean.
- `propertySets` — Should any property sets be enabled (such as macros) to be copied along with the workbook? Leaving this feature enabled will result in the JXL process using more memory. The value is a boolean.
- `rationalization` — Should the cell formats be rationalized before writing out the sheet? The value is a boolean. Default is true.
- `supressWarnings` — Should warnings be suppressed?. Due to the change in logging in version 2.4, this will now set the warning behaviour across the JVM (depending on the type of logger used). The value is a boolean.
- `temporaryFileDuringWriteDirectory` — Used in conjunction with the `useTemporaryFileDuringWrite` setting to set the target directory for the temporary files. This value can be NULL, in which case the normal system default temporary directory is used instead. The value is a string (the directory to which temporary files should be written).
- `useTemporaryFileDuringWrite` — Should a temporary file is used during the generation of the workbook. If not set, the workbook will take place entirely in memory. Setting this flag involves an assessment of the trade-offs between memory usage and performance. The value is a boolean.

	• `workbookProtected` — Should the workbook be protected? The value is a boolean. • `filename` — The filename to use for the download. The value is a string. Please note that if you map the DocumentServlet to some pattern, this file extension must also match. • `exportKey` — A key under which to store the resulting data in a DocumentData object under the event scope. If used, there is no redirection. *Child elements* • `<e:link/>` — Zero or more stylesheet links (see *Section 20.14.1, "Stylesheet links"*). • `<e:worksheet/>` — Zero or more worksheets (see *Section 20.4, "Worksheets"*). *Facets* • `none`

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

defines a workbook with a worksheet and a greeting at A1

20.4. 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 to the backing data. The value is a string. The target of this expression is examined for an Iterable. Note that if the target is a Map, the iteration is done over the Map.Entry

entrySet(), so you should use a .key or .value to target in your references.

- `var` — The current row iterator variable name that can later 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# where # is the worksheet index. If the given worksheet name exists, that sheet is selected. This can be used for merging several data sets into a single worksheet, just define the same name for them (using `startRow` and `startCol` to make sure that they don't occupy the same space).
- `startRow` — Defines the starting row for the data. The value is a number. Used for placing the data in other places than the upper-left corner (especially useful if having multiple data sets for a single worksheet). The defaults is 0.
- `startColumn` — Defines the starting column for the data. The value is a number. Used for placing the data in other places than the upper-left corner (especially useful if having multiple data sets for a single worksheet). The default is 0.
- `automaticFormulaCalculation` — Should formulas be automatically calculated? The value is a boolean.
- `bottomMargin` — The bottom margin. The value is a number (inches).
- `copies` — The number of copies. The value is a number.
- `defaultColumnWidth` — The default column width. The value is a number (characters * 256).
- `defaultRowHeight` — The default row height. The value is a number (1/20 of a point).
- `displayZeroValues` — Should zero-values be displayed? The value is a boolean.
- `fitHeight` — The number of pages vertically that this sheet will be printed into. The value is a number.
- `fitToPages` — Should printing be fit to pages? The value is a boolean.
- `fitWidth` — The number of pages widthwise which this sheet should be printed into. The value is a number.

- `footerMargin` — The margin for any page footer. The value is a number (inches).
- `headerMargin` — The margin for any page headers. The value is a number (inches).
- `hidden` — Should the worksheet be hidden? The value is a boolean.
- `horizontalCentre` — Should the worksheet be centered horizontally? The value is a boolean.
- `horizontalFreeze` — The row at which the pane is frozen vertically. The value is a number.
- `horizontalPrintResolution` — The horizontal print resolution. The value is a number.
- `leftMargin` — The left margin. The value is a number (inches).
- `normalMagnification` — The normal magnification factor (not zoom or scale factor). The value is a number (percentage).
- `orientation` — The paper orientation for printing this sheet. The value is a string that can be either "landscape" or "portrait".
- `pageBreakPreviewMagnification` — The page break preview magnification factor (not zoom or scale factors). The value is a number (percentage).
- `pageBreakPreviewMode` — Show page in preview mode? The value is a boolean.
- `pageStart` — The page number at which to commence printing. The value is a number.
- `paperSize` — The paper size to be used when printing this sheet. The value is a string that can be one of "a4", "a3", "letter", "legal" etc (see *jxl.format.PaperSize* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/PaperSize.html]).
- `password` — The password for this sheet. The value is a string.
- `passwordHash` — The password hash - used only when copying sheets. The value is a string.
- `printGridLines` — Should grid lines be printed? The value is a boolean.
- `printHeaders` — Should headers be printed? The value is a boolean.

- `sheetProtected` — Should the sheet be protected (read-only)? The value is a boolean.
- `recalculateFormulasBeforeSave` — Should the formulas be re-calculated when the sheet is saved? The value is a boolean. Default value is false.
- `rightMargin` — The right margin. The value is a number (inches).
- `scaleFactor` — The scale factor for this sheet to be used when printing. The value is a number (percent).
- `selected` — Should the sheet be selected when the workbook opens? The value is a boolean.
- `showGridLines` — Should gridlines be shown? The value is a boolean.
- `topMargin` — The top margin. The value is a number (inches).
- `verticalCentre` — Center vertically? The value is a boolean.
- `verticalFreeze` — The row at which the pane is frozen vertically. The value is a number.
- `verticalPrintResolution` — The vertical print resolution. The value is a number.
- `zoomFactor` — The zoom factor. Do not confuse zoom factor (which relates to the on screen view) with scale factor (which refers to the scale factor when printing). The value is a number (percentage).

Child elemenents

- `<e:printArea/>` — Zero or more print area definitions (see *Section 20.11, “Print areas and titles”*).
- `<e:printTitle/>` — Zero or more print title definitions (see *Section 20.11, “Print areas and titles”*).
- `<e:headerFooter/>` — Zero or more header/footer definitions (see *Section 20.10, “Headers and footers”*).
- Zero or more worksheet commands (see *Section 20.12, “Worksheet Commands”*).

Facets

	• `header`— Contents that will be placed at the top of the data block, above the column headers (if any). • `footer`— Contents that will be 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>
```

defines a worksheet with the name "foo", starting at B2.

20.5. Columns

Columns are the children of worksheets and the parents of cells, images, formulas and hyperlinks. They are the structure that control the iteration of the worksheet data. See *Section 20.14.5, "Column settings"* for formatting.

`<e:column>`	*Attributes* • `none` *Child elemenents* • `<e:cell/>` — Zero or more cells (see *Section 20.6, "Cells"*). • `<e:formula/>` — Zero or more formulas (see *Section 20.7, "Formulas"*). • `<e:image/>` — Zero or more images (see *Section 20.8, "Images"*). • `<e:hyperLink/>` — Zero or more hyperlinks (see *Section 20.9, "Hyperlinks"*).

	Facets • `header` — This facet can/will contain one `<e:cell>` , `<e:formula>` , `<e:image>` or `<e:hyperLink>` that will be used as header for the column. • `footer` — This facet can/will contain one `<e:cell>` , `<e:formula>` , `<e:image>` or `<e:hyperLink>` that will be used as footer for the column.

```
<e:workbook>
  <e:worksheet value="#{personList}" var="person">
    <e:column>
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
<e:workbook>
```

defines a column with a header and an iterated output

20.6. 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 *???*

`<e:cell>`	*Attributes* • `column` — The column where to place the cell. The default is the internal counter. The value is a number. Note that the value is 0-based. • `row` — The row where to place the cell. The default is the internal counter. The value is number. Note that the value is 0-based. • `value` — The value to display. Usually an EL-expression referencing the var-attribute of the containing datatable. The value is a string.

	• `comment` — A comment to add to the cell. The value is a string. • `commentHeight` — The height of the comment. The value is a number (in pixels). • `commentWidth` — A width of the comment. The value is a number (in pixels). *Child elemenents* • Zero or more validation conditions (see *Section 20.6.1, "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>
```

defines a column with a header and an iterated output

20.6.1. Validation

Validations are nested inside cells or formulas. They add constrains for the cell data.

`<e:numericValidation>`	*Attributes* • `value` — The limit (or lower limit where applicable) of the validation. The value is a number. • `value2` — The upper limit (where applicable) of the validation. The value is a number.

- `condition` — The validation condition. The value is a string.
  - "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 elemenents

- 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>
```

adds numeric validation to a cell specifying that the value must be between 4 and 18.

`<e:rangeValidation>`	*Attributes* • `startColumn` — The starting column of the range of values to validate against. The value is a number. • `startRow` — The starting row of the range of values to validate against. The value is a number. • `endColumn` — The ending column of the range of values to validate against. The value is a number. • `endRow` — The ending row of the range of values to validate against. The value is a number. *Child elemenents* • `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>
```

adds validation to a cell specifying that the value must be in the values specified in range A1:A10.

`<e:listValidation>`	*Attributes* • `none`

	Child elemenents • `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 values to validate against. *Child elemenents* • `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>
```

adds validation to a cell specifying that the value must be "manager" or "employee".

20.6.2. Format masks

Format masks are defined in the mask attribute in cells or formulas. There are two types of format masks, one for numbers and one for dates

20.6.2.1. Number masks

When encountering a format mask, first it is checked if it is in internal form, e.g "format1", "accounting_float" and so on (see *jxl.write.NumberFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/NumberFormats.html]).

if the mask is not in the list, it is treated as a custom mask (see *java.text.DecimalFormat* [http://java.sun.com/javase/6/docs/api/java/text/DecimalFormat.html]). e.g "0.00" and automatically converted to the closest match.

20.6.2.2. Date masks

When encountering a format mask, first it is checked if it is in internal form, e.g "format1", "format2" and so on (see *jxl.write.DateFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/DateFormats.html]).

if the mask is not in the list, it is treated as a custom mask (see *java.text.DateFormat* [http://java.sun.com/javase/6/docs/api/java/text/DateFormat.html])., e.g "dd.MM.yyyy" and automatically converted to the closest match.

20.7. 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 20.6, "Cells"* for available attributes. Note that they can apply templates and have own font definitions etc just as normal cells.

The formula of the cell is placed in the `value` -attribute as a normal the Microsoft® Excel® spreadsheet application notation. Note that 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>
```

defines an formula in B2 summing cells A1 in worksheets FooSheet and BarSheet

20.8. Images

Images are nested within columns (for iteration) or inside worksheets (for direct placement using the `startColumn/startRow` and `rowSpan/columnSpan` attributes). The spans are optional and if omitted, the image will be inserted without resizing.

`<e:image>`	*Attributes* • `startColumn` — The starting column of the image. The default is the internal counter. The value is a number. Note that the value is 0-based. • `startRow` — The starting row of the image. The default is the internal counter. The value is a number. Note that the value is 0-based. • `columnSpan` — The column span of the image. The default is one resulting in the default width of the image. The value is a float. • `rowSpan` — The row span of the image. The default is the one resulting in the default height of the image. The value is a float. • `URI` — The URI to the image. The value is a string. *Child elemenents* • `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>
```

defines an image in A1:E5 based on the given data

20.9. 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` — The starting column of the hyperlink. The default is the internal counter. The value is a number. Note that the value is 0-based. • `startRow` — The starting row of the hyperlink. The default is the internal counter. The value is a number. Note that the value is 0-based. • `endColumn` — The ending column of the hyperlink. The default is the internal counter. The value is a number. Note that the value is 0-based. • `endRow` — The ending row of the hyperlink. The default is the internal counter. The value is a number. Note that the value is 0-based. • `URL` — The URL to link. The value is a string. • `description` — The description of the link. The value is a string. *Child elemenents* • `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>
```

defines a described hyperlink pointing to SFWK in the area A1:E5

20.10. Headers and footers

Headers and footers are children of worksheets and contain facets which in turn contains a string with commands that are parsed.

`<e:header>`	*Attributes* • `none` *Child elemenents* • `none` *Facets* • `left` — The contents of the left header/footer part. • `center` — The contents of the center header/footer part. • `right` — The contents of the right header/footer part.
`<e:footer>`	*Attributes* • `none` *Child elemenents* • `none` *Facets* • `left` — The contents of the left header/footer part.

	• `center` — The contents of the center header/footer part. • `right` — The contents of the right header/footer part.

The content of the facets is a string that can contain various #-delimited commands as follows:

#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, use another #bold# to turn it off
#italics#	Toggles italic font, use another #italic# to turn it off
#underline#	Toggles underlining, use another #underline# to turn it off
#double_underline#	Toggles double underlining, use another #double_underline# to turn it off
#outline#	Toggles outlined font, use another #outline# to turn it off
#shadow#	Toggles shadowed font, use another #shadow# to turn it off
#strikethrough#	Toggles strikethrough font, use another #strikethrough# to turn it off
#subscript#	Toggles subscripted font, use another #subscript# to turn it off
#superscript#	Toggles superscript font, use another #superscript# to turn it off
#font_name#	Sets font name, used like #font_name=Verdana"
#font_size#	Sets font size, use like #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>
```

20.11. Print areas and titles

Print areas and titles childrens of worksheets and worksheet templates and provide... print areas and titles.

`<e:printArea>`	*Attributes* • `firstColumn` — The column of the top-left corner of the area. The parameter is a number. Note that the value is 0-based. • `firstRow` — The row of the top-left corner of the area. The parameter is a number. Note that the value is 0-based. • `lastColumn` — The column of the bottom-right corner of the area. The parameter is a number. Note that the value is 0-based. • `lastRow` — The row of the bottom-right corner of the area. The parameter is a number. Note that the value is 0-based. *Child elemenents* • `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>
```

defines a print title between A1:A10 and a print area between B2:J10.

20.12. Worksheet Commands

Worksheet commands are children of workbooks and are usually executed only once.

20.12.1. Grouping

Provides grouping of columns and rows.

`<e:groupRows>`	*Attributes* • `startRow` — The row to start the grouping at. The value is a number. Note that the value is 0-based. • `endRow` — The row to end the grouping at. The value is a number. Note that the value is 0-based. • `collapse` — Should the grouping be collapsed initially? The value is a boolean. *Child elements* • `none` *Facets* • `none`
`<e:groupColumns>`	*Attributes* • `startColumn` — The column to start the grouping at. The value is a number. Note that the value is 0-based. • `endColumn` — The column to end the grouping at. The value is a number. Note that the value is 0-based. • `collapse` — Should the grouping be collapsed initially? The value is a boolean. *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>
```

groups rows 5 through 10 and columns 5 through 10 so that the rows are initially collapsed (but not the columns).

20.12.2. Page breaks

Provides page breaks

`<e:rowPageBreak>`	*Attributes* • `row` — The row to break at. The value is a number. Note that the value is 0-based. *Child elements* • `none` *Facets* • `none`

```
<e:workbook>
  <e:worksheet>
    <e:rowPageBreak row="4"/>
  </e:worksheet>
</e:workbook>
```

breaks page at row 5.

20.12.3. Merging

Provides cell merging

`<e:mergeCells>`	*Attributes* • `startRow` — The row to start the merging from. The value is a number. Note that the value is 0-based. • `startColumn` — The column to start the merging from. The value is a number. Note that the value is 0-based. • `endRow` — The row to end the merging at. The value is a number. Note that the value is 0-based. • `endColumn` — The column to end the merging at. The value is a number. Note that 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>
```

merges the cells in the range A1:J10

20.13. Datatable exporter

If you prefer to export an existing JSF datatable instead of writing a dedicated XHTML document, this can also be achieved easily by executing the `org.jboss.seam.excel.excelExporter.export` component, passing in the id of the datatable as an Seam EL parameter. Consider you have a data table

```
<h:form id="theForm">
  <h:dataTable id="theDataTable" value="#{personList.personList}"
    var="person">
    ...
  </h:dataTable>
</h:form>
```

that you want to view as an Microsoft® Excel® spreadsheet. Place a

```
<h:commandLink
  value="Export"
  action="#{excelExporter.export('theForm:theDataTable')}"
/>
```

in the form and you're done. You can of course execute the exporter with a button, s:link or other preferred method. There are also plans for a dedicated export tag that can be placed inside the datatable tag so you won't have to refer to the datatable by ID.

See *Section 20.14, "Fonts and layout"* for formatting.

20.14. Fonts and layout

Controlling how the output look is done with a combination of CSSish style attributes and tag attributes. The most common ones (fonts, borders, backgrounds etc) are CSS and some more general settings are in tag attributes.

The CSS attributes cascade down from parent to children and within one tag cascades over the CSS classes referenced in the `styleClass` attributes and finally over the CSS attributes defined in the `style` attribute. You can place them pretty much anywhere but e.g. placing a column width setting in a cell nested within that column makes little sense.

If you have format masks or fonts that use special characters, such as spaces and semicolons, you can escape the css string with '' characters like xls-format-mask:'$;$'

20.14.1. Stylesheet links

External stylesheets are references with the e:link tag. They are placed as children of the workbook.

`<e:link>`	*Attributes* • `URL` — The URL to the stylesheet *Child elemenents* • `none` *Facets* • `none`

```
<e:workbook>
  <e:link URL="/css/excel.css"/>
</e:workbook>
```

References a stylesheet that can be found at /css/excel.css

20.14.2. Fonts

This group of XLS-CSS attributes define a font and its attributes

xls-font-family	The name of the font. Make sure that it's one that is supported by your system.
xls-font-size	The font size. Use a plain number
xls-font-color	The color of the font (see *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-font-bold	Should the font be bold? Valid values are "true" and "false"
xls-font-italic	Should the font be italic? Valid values are "true" and "false"
xls-font-script-style	The script style of the font (see *jxl.format.ScriptStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/ScriptStyle.html]).

xls-font-underline-style	The underline style of the font (see *jxl.format.UnderlineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/UnderlineStyle.html]).
xls-font-struck-out	Should the font be struck out? Valid values are "true" and "false"
xls-font	A shorthand notation for setting all the values. Place the font name last and use tick marks for fonts with spaces in them, e.g. 'Times New Roman'. Use "italic", "bold" and "struckout". Example style="xls-font: red bold italic 22 Verdana"

20.14.3. 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* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-left-line-style	The border line style of the left edge of the cell (see *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-left	A shorthand for setting line style and color of the left edge of the cell, e.g style="xls-border-left: thick red"
xls-border-top-color	The border color of the top edge of the cell (see *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-top-line-style	The border line style of the top edge of the cell (see *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-top	A shorthand for setting line style and color of the top edge of the cell, e.g style="xls-border-top: red thick"
xls-border-right-color	The border color of the right edge of the cell (see *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-right-line-style	The border line style of the right edge of the cell (see *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-right	A shorthand for setting line style and color of the right edge of the cell, e.g style="xls-border-right: thick red"
xls-border-bottom-color	The border color of the bottom edge of the cell (see *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).

xls-border-bottom-line-style	The border line style of the bottom edge of the cell (see *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-bottom	A shorthand for setting line style and color of the bottom edge of the cell, e.g style="xls-border-bottom: thick red"
xls-border	A shorthand for setting line style and color for all edges of the cell, e.g style="xls-border: thick red"

20.14.4. Background

This group of XLS-CSS attributes defines the background of the cell

xls-background-color	The color of the background (see *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-background-pattern	The pattern of the background (see *jxl.format.Pattern* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Pattern.html]).
xls-background	A shorthand for setting the background color and pattern. See above for rules.

20.14.5. Column settings

This group of XLS-CSS attributes defines the column widths etc.

xls-column-width	The width of the column. Use largeish values (~5000) to start with. Used by the e:column in xhtml mode.
xls-column-widths	The width of the column. Use largeish values (~5000) to start with. Used by the excel exporter, placed in the datatable style attribute. Use numerical values or * to bypass a column. Example style="xls-column-widths: 5000, 5000, *, 10000"
xls-column-autosize	Should an attempt be made to autosize the column? Valid values are "true" and "false".
xls-column-hidden	Should the column be hidden? Valid values are "true" and "false".
xls-column-export	Should the column be shown in export? Valid values are "true" and "false". Default is "true".

20.14.6. Cell settings

This group of XLS-CSS attributes defines the cell properties

xls-alignment	The alignment of the cell value (see *jxl.format.Alignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Alignment.html]).

xls-force-type	The forced type of the cell data. The value is a string that can be one of "general", "number", "text", "date", "formula" or "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 20.6.2, "Format masks"*
xls-indentation	The indentation of the cell value. The value is numeric.
xls-locked	Should the cell be locked. Use with workbook level locked. Valid values are "true" and "false".
xls-orientation	The orientation of the cell value (see *jxl.format.Orientation* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Orientation.html]).
xls-vertical-alignment	The vertical alignment of the cell value (see *jxl.format.VerticalAlignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/VerticalAlignment.html]).
xls-shrink-to-fit	Should the cell values shrink to fit? Valid values are "true" and "false".
xls-wrap	Should the cell wrap with newlines? Valid values are "true" and "false".

20.14.7. 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).

20.14.8. Layout examples

TODO

20.14.9. Limitations

In the current version there are some known limitations regarding CSS support

- 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

20.15. Internationalization

There are only two resources bundle keys used, both for invalid data format and both take a parameter (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

20.16. Links and further documentation

The core of the the Microsoft® Excel® spreadsheet application functionality is based on the excellent JExcelAPI library which can be found on *http://jexcelapi.sourceforge.net/* [http://jexcelapi.sourceforge.net] and most features and possible limitations are inherited from here.

If you use the forum or mailing list, please remember that they don't know anything about Seam and the usage of their library, any issues are best reported in the JBoss Seam JIRA under the "excel" module.

RSS support

It is now easy to integrate RSS feeds in Seam through the *YARFRAW* [http://yarfraw.sourceforge.net/] library. The RSS support is currently in the state of "tech preview" in the current release.

21.1. Installation

To enable RSS support, include the `jboss-seam-rss.jar` in your applications `WEB-INF/lib` directory. The RSS library also has some dependent libraries that should be placed in the same directory. See *Section 39.2.6, "Seam RSS support"* for a list of libraries to include.

The Seam RSS support requires the use of Facelets as the view technology.

21.2. Generating feeds

The `examples/rss` project contains an example of RSS support in action. It demonstrates proper deployment packaging, and it shows the exposed functionality.

A feed is a xhtml-page that consist of a feed and a list of nested entry items.

```
<r:feed
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:r="http://jboss.org/schema/seam/rss"
  title="#{rss.feed.title}"
  uid="#{rss.feed.uid}"
  subtitle="#{rss.feed.subtitle}"
  updated="#{rss.feed.updated}"
  link="#{rss.feed.link}">
  <ui:repeat value="#{rss.feed.entries}" var="entry">
    <r:entry
      uid="#{entry.uid}"
      title="#{entry.title}"
      link="#{entry.link}"
      author="#{entry.author}"
      summary="#{entry.summary}"
      published="#{entry.published}"
      updated="#{entry.updated}"
    />
  </ui:repeat>
</r:feed>
```

21.3. Feeds

Feeds are the top-level entities that describe the properties of the information source. It contains zero or more nested entries.

`<r:feed>`	*Attributes* • `uid` — An optional unique feed id. The value is a string. • `title` — The title of the feed. The value is a string. • `subtitle` — The subtitle of the feed. The value is a string. • `updated` — When was the feed updated? The value is a date. • `link` — The link to the source of the information. The value is a string. • `feedFormat` — The feed format. The value is a string and defaults to ATOM1. Valid values are RSS10, RSS20, ATOM03 and ATOM10. *Child elements* • `Zero or more feed entries` *Facets* • `none`

21.4. Entries

Entries are the "headlines" in the feed.

`<r:feed>`	*Attributes* • `uid` — An optional unique entry id. The value is a string. • `title` — The title of the entry. The value is a string. • `link` — A link to the item. The value is a string. • `author` — The author of the story. The value is a string. • `summary` — The body of the story. The value is a string.

	• `textFormat` — The format of the body and title of the story. The value is a string and valid values are "text" and "html". Defaults to "html". • `published` — When was the story first published? The value is a date. • `updated` — When was the story updated? The value is a date. *Child elements* • `none` *Facets* • `none`

21.5. Links and further documentation

The core of the RSs functionality is based on the YARFRAW library which can be found on *http://yarfraw.sourceforge.net/* and most features and possible limitations are inherited from here.

For details on the ATOM 1.0 format, have a look at *the specs* [http://atompub.org/2005/07/11/draft-ietf-atompub-format-10.html]

For details on the RSS 2.0 format, have a look at *the specs* [http://cyber.law.harvard.edu/rss/rss.html]

Email

Seam now includes an optional components for templating and sending emails.

Email support is provided by `jboss-seam-mail.jar`. This JAR contains the mail JSF controls, which are used to construct emails, and the `mailSession` manager component.

The examples/mail project contains an example of the email support in action. It demonstrates proper packaging, and it contains a number of example that demonstrate the key features currently supported.

You can also test your mail's using Seam's integration testing environment. See *Section 38.2.3.4, "Integration Testing Seam Mail"*.

22.1. Creating a message

You don't need to learn a whole new templating language to use Seam Mail — an email is just facelet!

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
   xmlns:m="http://jboss.org/schema/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 set who the message is from, a `<m:to>` tag to specify a sender (notice how we use EL as we would in a normal facelet), and a `<m:subject>` tag.

The `<m:body>` tag wraps the body of the email. You can use regular HTML tags inside the body as well as JSF components.

So, now you have your email template, how do you go about sending it? Well, at the end of rendering the `m:message` the `mailSession` is called to send the email, so all you have to do is ask Seam to 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 would be thrown, which is caught and then displayed to the user.

22.1.1. Attachments

Seam makes it easy to attach files to an email. It supports most of the standard java types used when working with files.

If you wanted to email the `jboss-seam-mail.jar`:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam will load the file from the classpath, and attach it to the email. By default it would be attached as `jboss-seam-mail.jar`; if you wanted it to have another name you would just add the `fileName` attribute:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar" fileName="this-is-so-cool.jar"/>
```

You could 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"/>
```

You'll notice that for a `byte[]` and a `java.io.InputStream` you need to specify the MIME type of the attachment (as that information is not carried as part of the file).

And it gets even better, you can attach a Seam generated PDF, or any standard JSF view, just by wrapping a `<m:attachment>` around the normal tags you would use:

```
<m:attachment fileName="tiny.pdf">
   <p:document>
      A very tiny PDF
   </p:document>
</m:attachment>
```

If you had a set of files you wanted to attach (for example a set of pictures loaded from a database) you can just 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>
```

And if you want 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}" />
```

You may be wondering what `cid:#{...}` does. Well, the IETF specified that by putting this as the src for your image, the attachments will be looked at when trying to locate the image (the `Content-ID`'s must match) — magic!

You must declare the attachment before trying to access the status object.

22.1.2. HTML/Text alternative part

Whilst most mail readers nowadays support HTML, some don't, so you can add a plain text alternative to your email body:

```
<m:body>
   <f:facet name="alternative">Sorry, your email reader can't show our fancy email,
please go to http://labs.jboss.com/jbossseam to explore Seam.</f:facet>
</m:body>
```

22.1.3. Multiple recipients

Often you'll want to send an email to a group of recipients (for example your users). All of the 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>
```

22.1.4. Multiple messages

Sometimes, however, you need to send a slightly different message to each recipient (e.g. a password reset). 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>
```

22.1.5. Templating

The mail templating example shows that facelets templating just 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, but you must place them in a jar in `WEB-INF/lib` - referencing the `.taglib.xml` from `web.xml` isn't reliable when using Seam Mail (if you send your mail asynchronously Seam Mail doesn't have access to the full JSF or Servlet context, and so doesn't know about `web.xml` configuration parameters).

If you do need more configure Facelets or JSF when sending mail, you'll need to override the Renderer component and do the configuration programmatically - only for advanced users!

22.1.6. Internationalisation

Seam supports sending internationalised messages. By default, the encoding provided by JSF is used, but this can be overridden on the template:

```
<m:message charset="UTF-8">
   ...
</m:message>
```

The body, subject and recipient (and from) name will be encoded. You'll need to make sure facelets uses the correct charset for parsing your pages by setting encoding of the template:

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

22.1.7. Other Headers

Sometimes you'll want to add other headers to your email. Seam provides support for some (see *Section 22.4, "Tags"*). For example, we can set the importance of the email, and ask for a read receipt:

```
<m:message xmlns:m="http://jboss.org/schema/seam/mail"
   importance="low"
   requestReadReceipt="true"/>
```

Otherwise you can add any header to the message using the `<m:header>` tag:

```
<m:header name="X-Sent-From" value="JBoss Seam"/>
```

22.2. Receiving emails

Warning

Please be reminded that this section is not updated for JBoss AS 7.x!

If you are using EJB then you can use a MDB (Message Driven Bean) to receive email. JBoss provides a JCA adaptor — `mail-ra.rar` — but the version distributed with JBoss AS 4.x has a number of limitations (and isn't bundled in some versions) therefore we recommend using the `mail-ra.rar` distributed with Seam (it's in the `extras/` directory in the Seam bundle). `mail-ra.rar` should be placed in `$JBOSS_HOME/server/default/deploy`; if the version of JBoss AS you use already has this file, replace it.

Note

JBoss AS 5.x and newer has mail-ra.rar applied the patches, so there is no need to copy the mail-ra.rar from Seam distribution.

You can configure it 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 will cause `onMessage(Message message)` to be called. Most Seam annotations will work inside a MDB but you must not access the persistence context.

You can find more information on `mail-ra.rar` at *http://www.jboss.org/community/wiki/InboundJavaMail*.

If you aren't using JBoss AS you can still use `mail-ra.rar` or you may find your application server includes a similar adapter.

22.3. Configuration

To include Email support in your application, include `jboss-seam-mail.jar` in your `WEB-INF/lib` directory. If you are using JBoss AS there is no further configuration needed to use Seam's email support. Otherwise you need to make sure you have the JavaMail API, an implementation of the JavaMail API present (the API and impl used in JBoss AS are distributed with seam as `lib/mail.jar`), and a copy of the Java Activation Framework (distributed with Seam as `lib/activation.jar`.

Note

The Seam Mail module requires the use of Facelets as the view technology. This is the default View technology in JSF 2. Additionally, it requires the use of the jboss-seam-ui module.

The `mailSession` component uses JavaMail to talk to a 'real' SMTP server.

22.3.1. `mailSession`

A JavaMail Session may be available via a JNDI lookup if you are working in an JEE environment or you can use a Seam configured Session.

The mailSession component's properties are described in more detail in *Section 33.9, "Mail-related components"*.

22.3.1.1. JNDI lookup in JBoss AS

The JBoss AS 7 Mail service is defined in `standalone/configuration/standalone.xml` file. It configures a JavaMail session binding into JNDI. The default service configuration will need altering for your network. *Full article how to configure Mail system in JBoss AS 7* [http://www.mastertheboss.com/jboss-application-server/379-jboss-mail-service-configuration.html] describes the service in more detail.

```
<components xmlns="http://jboss.org/schema/seam/components"
   xmlns:core="http://jboss.org/schema/seam/core"
   xmlns:mail="http://jboss.org/schema/seam/mail">

   <mail:mail-session session-jndi-name="java:jboss/mail/Default"/>

</components>
```

Here we tell Seam to get the mail session bound to `java:jboss/mail/Default` from JNDI.

22.3.1.2. 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.org/schema/seam/components"
   xmlns:core="http://jboss.org/schema/seam/core"
   xmlns:mail="http://jboss.org/schema/seam/mail">

   <mail:mail-session host="smtp.example.com"/>

</components>
```

22.4. Tags

Emails are generated using tags in the `http://jboss.org/schema/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.

The standard templating tags of facelets can be used as normal. Inside the body you can use any JSF tag; if it requires access to external resources (stylesheets, javascript) then be sure to set the `urlBase`.

<m:message>
Root tag of a mail message

- `importance` — low, normal or high. By default normal, this sets the importance of the mail message.
- `precedence` — sets the precedence of the message (e.g. bulk).
- `requestReadReceipt` — by default false, if set, a read receipt request will be will be added, with the read receipt being sent to the `From:` address.
- `urlBase` — If set, the value is prepended to the `requestContextPath` allowing you to use components such as `<h:graphicImage>` in your emails.
- `messageId` — Sets the Message-ID explicitly

<m:from>
Set's the From: address for the email. You can only have one of these per email.

- `name` — the name the email should come from.
- `address` — the email address the email should come from.

<m:replyTo>
Set's the Reply-to: address for the email. You can only have one of these per email.

- `address` — the email address the email should come from.

<m:to>
Add 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>
Add 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>
Add 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>
Add a header to the email (e.g. `X-Sent-From: JBoss Seam`)

- `name` — The name of the header to add (e.g. `X-Sent-From`).
- `value` — The value of the header to add (e.g. `JBoss Seam`).

<m:attachment>
Add 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 an `byte[]`. In this case both a `fileName` and a `contentType` must be specified.

  If the value attribute is ommitted:

  - 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` — Specify the file name to use for the attached file.
- `contentType` — Specify the MIME type of the attached file

<m:subject>
Set's the subject for the email.

<m:body>
Set's the body for the email. Supports an `alternative` facet which, if an HTML email is generated can contain alternative text for a mail reader which doesn't support html.

- `type` — If set to `plain` then a plain text email will be generated otherwise an HTML email is generated.

Asynchronicity and messaging

Seam makes it very easy to perform work asynchronously from a web request. When most people think of asynchronicity in Java EE, they think of using JMS. This is certainly one way to approach the problem in Seam, and is the right way when you have strict and well-defined quality of service requirements. Seam makes it easy to send and receive JMS messages using Seam components.

But for cases when you are simply want to use a worker thread, JMS is overkill. 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

This chapter first covers how to leverage Seam to simplify JMS and then explains how to use the simpler asynchronous method and event facility.

23.1. Messaging in Seam

Seam makes it easy to send and receive JMS messages to and from Seam components. Both the message publisher and the message receiver can be Seam components.

You'll first learn to setup a queue and topic message publisher and then look at an example that illustrates how to perform the message exchange.

23.1.1. Configuration

To configure Seam's infrastructure for sending JMS messages, you need to tell Seam about any topics and queues you want to send messages to, and also tell Seam where to find the `QueueConnectionFactory` and/or `TopicConnectionFactory`.

Seam defaults to using `UIL2ConnectionFactory` which is the usual connection factory for use with JBossMQ. If you are using some other JMS provider, you need to set one or both of `queueConnection.queueConnectionFactoryJndiName` and `topicConnection.topicConnectionFactoryJndiName` in `seam.properties`, `web.xml` or `components.xml`.

You also need to list topics and queues in `components.xml` to install Seam managed `TopicPublishers` and `QueueSenders`:

```
<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"/>
```

23.1.2. Sending messages

Now, you can inject a JMS `TopicPublisher` and `TopicSession` into any Seam component to publish an object to a topic:

```
@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 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);
    }
  }
}
```

23.1.3. Receiving messages using a message-driven bean

You can process messages using any EJB 3 message-driven bean. The MDB can even be a Seam component, in which case it's possible to inject other event- and application- scoped Seam components. Here's an example of the payment receiver, which delegates to a payment processor.

Note

You'll likely need to set the create attribute on the `@In` annotation to true (i.e. create = true) to have Seam create an instance of the component being injected. This isn't necessary if the component supports auto-creation (e.g., it's annotated with `@Autocreate`).

First, create an MDB 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);
    }
  }
}
```

Then, implement the Seam component to which the receiver delegates processing of the payment.

```
@Name("paymentProcessor")
public class PaymentProcessor
{
  @In private EntityManager entityManager;

  public void processPayment(Payment payment)
  {
    // perhaps do something more fancy
    entityManager.persist(payment);
  }
}
```

If you are going to be performing transaction operations in your MDB, you should ensure that you are working with an XA datasource. Otherwise, it won't be possible to rollback database changes if the database transaction commits and a subsequent operation being performed by the message fails.

23.1.4. Receiving messages in the client

Seam Remoting lets you subscribe to a JMS topic from client-side JavaScript. This is described in *Chapter 26, Remoting*.

23.2. 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're working in an environment that supports EJB 3.0, and add the following line to `components.xml`:

```
<async:timer-service-dispatcher/>
```

then your asynchronous tasks will be processed by the container's EJB timer service. If you're not familiar with the Timer service, don't worry, you don't need to interact with it directly if you want to use asynchronous methods in Seam. The important thing to know is that any good EJB 3.0 implementation will have the option of using persistent timers, which gives some guarantee that the tasks will eventually be processed.

Another alternative is to use the open source Quartz library to manage asynchronous method. You need to bundle the Quartz library JAR (found in the `lib` directory) in your EAR and declare it as a Java module in `application.xml`. The Quartz dispatcher may be configured by adding a Quartz property file to the classpath. It must be named `seam.quartz.properties`. In addition, you need to add the following line to `components.xml` to install the Quartz dispatcher.

```
<async:quartz-dispatcher/>
```

The Seam API for the default `ScheduledThreadPoolExecutor`, the EJB3 `Timer`, and the Quartz `Scheduler` are largely the same. They can just "plug and play" by adding a line to `components.xml`.

23.2.1. Asynchronous methods

In simplest form, an asynchronous call just lets a method call be processed asynchronously (in a different thread) from the caller. We usually use an asynchronous call when we want to return an immediate response to the client, and let some expensive work be processed in the background. This pattern works very well in applications which use AJAX, where the client can automatically poll the server for the result of the work.

For EJB components, we annotate the local interface to specify that a method is processed asynchronously.

```
@Local
public interface PaymentHandler
{
   @Asynchronous
   public void processPayment(Payment payment);
}
```

(For JavaBean components we have to annotate the component implementation class.)

The use of asynchronicity is transparent to the bean class:

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
   public void processPayment(Payment payment)
   {
      //do some work!
   }
}
```

And 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";
   }
}
```

Note

Please distinguish between *EJB 3.1 annotation* `javax.ejb.Asynchronous` [http://docs.oracle.com/javaee/6/api/javax/ejb/Asynchronous.html] and Seam annotation `org.jboss.seam.annotations.async.Asynchronous`. While first is designated for session beans only, the latter works in non-EJB environment too.

The asynchronous method is processed in a completely new event context and does not have access to the session or conversation context state of the caller. However, the business process context *is* propagated.

Asynchronous method calls may be scheduled for later execution using 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 may access the `Timer` object associated with the invocation. The `Timer` object shown below is the EJB3 timer when you use the EJB3 dispatcher. For the default `ScheduledThreadPoolExecutor`, the returned object is `Future` from the JDK. For the Quartz dispatcher, it returns `QuartzTriggerHandle`, which we will discuss 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.

23.2.2. Asynchronous methods with the Quartz Dispatcher

The Quartz dispatcher (see earlier on how to install it) allows you to use the `@Asynchronous`, `@Duration`, `@Expiration`, and `@IntervalDuration` annotations as above. But it has some powerful additional features. The Quartz dispatcher supports three new annotations.

The `@FinalExpiration` annotation specifies an end date for the 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 the method returns the `QuartzTriggerHandle` object, which you can use later to stop, pause, and resume the scheduler. The `QuartzTriggerHandle` object is serializable, so you can save it into the database if you need to keep it around for 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 instance, 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 on the "nth Business Day" scenario. For instance, the following asynchronous method runs at 14:00 on the 2nd business day of each month. By default, it excludes all weekends and US federal holidays until 2010 from the business days.

```
// 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 (e.g., company holidays, non-US holidays etc.) via 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. If they are used in the same method, a `RuntimeException` will be thrown.

23.2.3. Asynchronous events

Component-driven events may also be asynchronous. To raise an event for asynchronous processing, simply call the `raiseAsynchronousEvent()` method of the `Events` class. To schedule a timed event, call the `raiseTimedEvent()` method, passing a *schedule* object (for the default dispatcher or timer service dispatcher, use `TimerSchedule`). Components may observe asynchronous events in the usual way, but remember that only the business process context is propagated to the asynchronous thread.

23.2.4. Handling exceptions from asynchronous calls

Each asynchronous dispatcher behaves differently when an exception propagates through it. For example, the `java.util.concurrent` dispatcher will suspend further executions of a call which

repeats, and the EJB3 timer service will swallow the exception. Seam therefore catches any exception which propagates out of the asynchronous call before it reaches the dispatcher.

By default, any exception which propagates out 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, for example, using `java.util.concurrent` dispatcher, we inject its control object and cancel all future invocations when an exception is encountered

You can also alter this behavior for an individual component by implementing the method `public void handleAsynchronousException(Exception exception);` on the component. For example:

```
  public void handleAsynchronousException(Exception exception) {
    log.fatal(exception);
  }
```

Caching

In almost all enterprise applications, the database is the primary bottleneck, and the least scalable tier of the runtime environment. People from a PHP/Ruby environment will try to tell you that so-called "shared nothing" architectures scale well. While that may be literally true, I don't know of many interesting multi-user applications which can be implemented with no sharing of resources between different nodes of the cluster. What these silly people are really thinking of is a "share nothing except for the database" architecture. Of course, sharing the database is the primary problem with scaling a multi-user application — so the claim that this architecture is highly scalable is absurd, and tells you a lot about the kind of applications that these folks spend most of their time working on.

Almost anything we can possibly do to share the database *less often* is worth doing.

This calls for a cache. Well, not just one cache. A well designed Seam application will feature a rich, multi-layered caching strategy that impacts every layer of the application:

- The database, of course, has its own cache. This is super-important, but can't scale like a cache in the application tier.
- Your ORM solution (Hibernate, or some other JPA implementation) has a second-level cache of data from the database. This is a very powerful capability, but is often misused. In a clustered environment, keeping the data in the cache transactionally consistent across the whole cluster, and with the database, is quite expensive. It makes most sense for data which is shared between many users, and is updated rarely. In traditional stateless architectures, people often try to use the second-level cache for conversational state. This is always bad, and is especially wrong in Seam.
- The Seam conversation context is a cache of conversational state. Components you put into the conversation context can hold and cache state relating to the current user interaction.
- In particular, the Seam-managed persistence context (or an extended EJB container-managed persistence context associated with a conversation-scoped stateful session bean) acts as a cache of data that has been read in the current conversation. This cache tends to have a pretty high hitrate! Seam optimizes the replication of Seam-managed persistence contexts in a clustered environment, and there is no requirement for transactional consistency with the database (optimistic locking is sufficient) so you don't need to worry too much about the performance implications of this cache, unless you read thousands of objects into a single persistence context.
- The application can cache non-transactional state in the Seam application context. State kept in the application context is of course not visible to other nodes in the cluster.
- The application can cache transactional state using the Seam `cacheProvider` component, which integrates JBossCache, JBoss POJO Cache, Infinispan or EHCache into the Seam

environment. This state will be visible to other nodes if your cache supports running in a clustered mode.

- Finally, Seam lets you cache rendered fragments of a JSF page. Unlike the ORM second-level cache, this cache is not automatically invalidated when data changes, so you need to write application code to perform explicit invalidation, or set appropriate expiration policies.

For more information about the second-level cache, you'll need to refer to the documentation of your ORM solution, since this is an extremely complex topic. In this section we'll discuss the use of caching directly, via the `cacheProvider` component, or as the page fragment cache, via the `<s:cache>` control.

24.1. Using Caching in Seam

The built-in `cacheProvider` component manages an instance of:

Infinispan 5.x (suitable for use in JBoss AS 7.1.x or later and other containers)
`org.infninispan.tree.TreeCache`

JBoss Cache 1.x (suitable for use in JBoss 4.2.x or later and other containers)
`org.jboss.cache.TreeCache`

JBoss Cache 2.x (suitable for use in JBoss 5.x and other containers)
`org.jboss.cache.Cache`

JBoss POJO Cache 1.x (suitable for use in JBoss 4.2.x or later and other containers)
`org.jboss.cache.aop.PojoCache`

EHCache (suitable for use in any container)
`net.sf.ehcache.CacheManager`

You can safely put any immutable Java object in the cache, and it will be stored in the cache and replicated across the cluster (assuming that replication is supported and enabled). If you want to keep mutable objects in the cache read the documentation of the underling caching project documentation to discover how to notify the cache of changes to the cache.

To use `cacheProvider`, you need to include the jars of the cache implementation in your project:

Infinispan 5.x

- `infinispan-core.jar` - Infinispan Core 5.1.x.Final

  `infinispan-tree.jar` - Infinispan TreeCache 5.1.x.Final

- `jgroups.jar` - JGroups 3.0

JBoss Cache 1.x

- `jboss-cache.jar` - JBoss Cache 1.4.1
- `jgroups.jar` - JGroups 2.4.1

JBoss Cache 2.x

- `jboss-cache.jar` - JBoss Cache 2.2.0
- `jgroups.jar` - JGroups 2.6.2

JBoss POJO Cache 1.x

- `jboss-cache.jar` - JBoss Cache 1.4.1
- `jgroups.jar` - JGroups 2.4.1
- `jboss-aop.jar` - JBoss AOP 1.5.0

EHCache

- `ehcache.jar` - EHCache 1.2.3

Tip

If you would like to know more details about Infinispan, look at the Infinispan *Documentation* [https://docs.jboss.org/author/display/ISPN/Home] page.

For an EAR deployment of Seam, we recommend that the infinispan jars and configuration go directly into the EAR.

Note

JBoss AS7 already provides Infinispan and JGroups jars, so you need to turn on that dependencies in your JBoss AS 7 deployment file or modify `META-INF/Manifest.mf` to have this dependencies. Check the Blog example or JBoss AS7 documentation how to do that.

You'll also need to provide a configuration file for Infinispan. Place `infinispan.xml` with an appropriate cache configuration into the Web applicaiton classpath (e.g. the ejb jar or `WEB-INF/classes`). Infinispan has many configuration settings, so we won't discuss them here. Please refer to the Infinispan documentation for more information.

You can find a sample configuration file `infinispan.xml` in `examples/blog/blog-web/src/main/resources/infinispan.xml`.

EHCache will run in it's default configuration without a configuration file

To alter the configuration file in use, configure your cache in `components.xml`:

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:cache="http://jboss.org/schema/seam/cache">
```

```
  <cache:infinispan-cache-provider configuration="infinispan.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 to have multiple cache configurations in your application, use `components.xml` to configure multiple cache providers:

```
<components xmlns="http://jboss.org/schema/seam/components"
          xmlns:cache="http://jboss.org/schema/seam/cache">
  <cache:infinispan-cache-provider name="myCache" configuration="myown/cache.xml"/>
          <cache:infinispan-cache-provider    name="myOtherCache"    configuration="myother/
cache.xml"/>
</components>
```

24.2. Page fragment caching

The most interesting use of caching in Seam is the `<s:cache>` tag, Seam's solution to the problem of page fragment caching in JSF. `<s:cache>` uses `pojoCache` internally, so you need to follow the steps listed above before you can use it. (Put the jars in the EAR, wade through the scary configuration options, etc.)

`<s:cache>` is used for caching some rendered content which changes rarely. For example, the welcome page of our blog displays the 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` let's you have multiple cached versions of each page fragment. In this case, there is one cached version per blog. The `region` determines the cache or region node that all version will be stored in. Different nodes may have different expiry policies. (That's the stuff you set up using the aforementioned scary configuration options.)

Of course, the big problem with `<s:cache>` is that it is too stupid to know when the underlying data changes (for example, when the blogger posts a new entry). So you need to evict the cached fragment manually:

```
public void post() {
   ...
   entityManager.persist(blogEntry);
   cacheProvider.remove("welcomePageFragments", "recentEntries-" + blog.getId() );
}
```

Alternatively, if it is not critical that changes are immediately visible to the user, you could set a short expiry time on the cache node.

Web Services

Seam integrates with JBossWS to allow standard Java EE web services to take full advantage of Seam's contextual framework, including support for conversational web services. This chapter walks through the steps required to allow web services to run within a Seam environment.

25.1. Configuration and Packaging

To allow Seam to intercept web service requests so that the necessary Seam contexts can be created for the request, a special SOAP handler must be configured; `org.jboss.seam.webservice.SOAPRequestHandler` is a `SOAPHandler` implementation that does the work of managing Seam's lifecycle during the scope of a web service request.

A special configuration file, `soap-handlers.xml` should be placed into the `META-INF` directory of the `jar` file that contains the web service classes. This file contains the following SOAP handler configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-name>SOAP Request Handler</handler-name>
      <handler-class>org.jboss.seam.webservice.SOAPRequestHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

25.2. Conversational Web Services

So how are conversations propagated between web service requests? Seam uses a SOAP header element present in both the SOAP request and response messages to carry the conversation ID from the consumer to the service, and back again. Here's an example of a web service request that contains a conversation ID:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:sb="http://seambay.example.seam.jboss.org/">
  <soapenv:Header>
                <seam:conversationId     xmlns:seam='http://www.jboss.org/seam/webservice'>4</
seam:conversationId>
  </soapenv:Header>
  <soapenv:Body>
    <sb:setAuctionPrice>
```

```
    <arg0>100</arg0>
   </sb:setAuctionPrice>
  </soapenv:Body>
</soapenv:Envelope>
```

As you can see in the above SOAP message, there is a `conversationId` element within the SOAP header that contains the conversation ID for the request, in this case `4`. Unfortunately, because web services may be consumed by a variety of web service clients written in a variety of languages, it is up to the developer to implement conversation ID propagation between individual web services that are intended to be used within the scope of a single conversation.

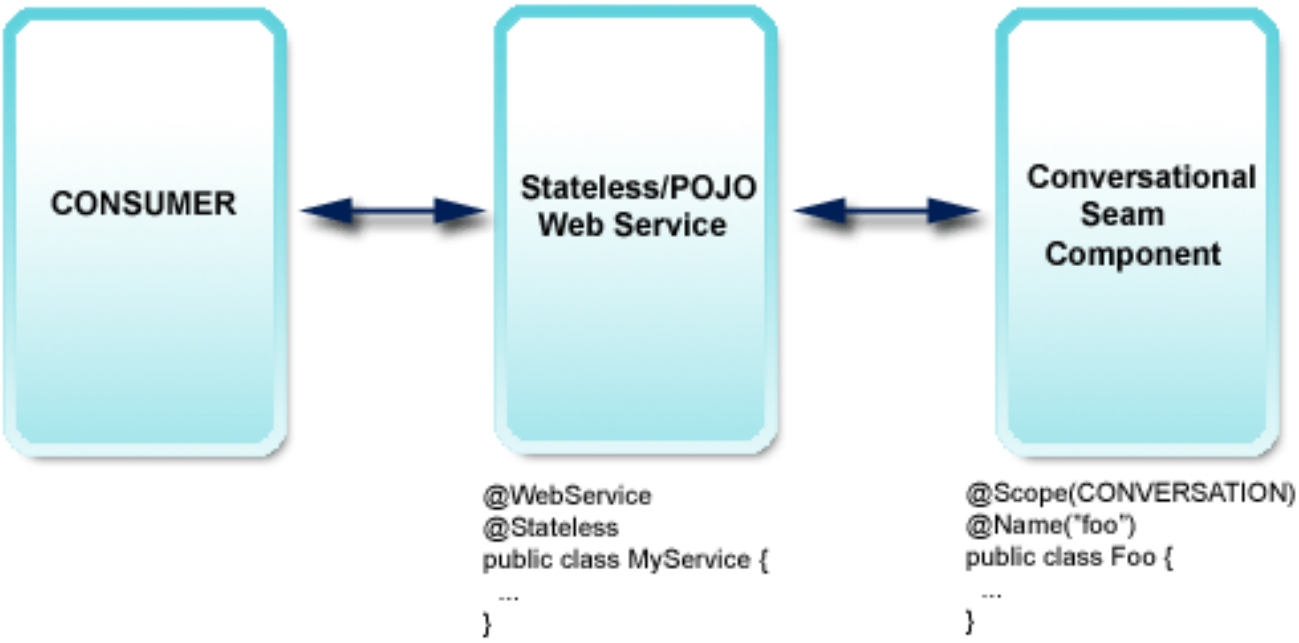
An important thing to note is that the `conversationId` header element must be qualified with a namespace of `http://www.jboss.org/seam/webservice`, otherwise Seam will not be able to read the conversation ID from the request. Here's an example of a response to the above request message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header>
            <seam:conversationId  xmlns:seam="http://www.jboss.org/seam/webservice">4</
seam:conversationId>
   </soap:Header>
   <soap:Body>
      <ns2:setAuctionPriceResponse xmlns:ns2="http://seambay.example.seam.jboss.org/"/>
   </soap:Body>
 </soap:Envelope>
```

As you can see, the response message contains the same `conversationId` element as the request.

25.2.1. A Recommended Strategy

As web services must be implemented as either a stateless session bean or POJO, it is recommended that for conversational web services, the web service acts as a facade to a conversational Seam component.



If the web service is written as a stateless session bean, then it is also possible to make it a Seam component by giving it a `@Name`. Doing this allows Seam's bijection (and other) features to be used in the web service class itself.

25.3. An example web service

Let's walk through an example web service. The code in this section all comes from the seamBay example application in Seam's `/examples` directory, and follows the recommended strategy as described in the previous section. Let's first take a look at the web service class and one of its web service methods:

```
@Stateless
@Name("auctionService")
@WebService(name = "AuctionService")
@HandlerChain(file = "soap-handlers.xml")
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
```

```
}
```

As you can see, our web service is a stateless session bean, and is annotated using 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, and 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.

As is required by the specification, each method that is to be exposed as a web service method must also be declared in the remote interface of the web service class (when the web service is a stateless session bean). In the above example, the `AuctionServiceRemote` interface must declare the `login()` method as it is annotated as a `@WebMethod`.

As you can see in the above code, the web service implements a `login()` method that delegates to Seam's built-in `Identity` component. In keeping with our recommended strategy, the web service is written as a simple facade, passing off the real work to a Seam component. This allows for the greatest reuse of business logic between web services and other clients.

Let's look at another example. This 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);
}
```

And here's the code from `AuctionAction`:

```
@Begin
public void createAuction()
{
   auction = new Auction();
   auction.setAccount(authenticatedAccount);
   auction.setStatus(Auction.STATUS_UNLISTED);
   durationDays = DEFAULT_AUCTION_DURATION;
}
```

From this we can 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.

25.4. RESTful HTTP webservices with RESTEasy

Seam integrates the RESTEasy implementation of the JAX-RS specification (JSR 311). You can decide how "deep" the integration into your Seam application is going to be:

- Seamless integration of RESTEasy bootstrap and configuration, automatic detection of resources and providers.
- Serving HTTP/REST requests with the SeamResourceServlet, no external servlet or configuration in web.xml required.
- Writing resources as Seam components, with full Seam lifecycle management and interception (bijection).

25.4.1. RESTEasy configuration and request serving

First, get the RESTEasy libraries and the `jaxrs-api.jar`, deploy them with the other libraries of your application. Also deploy the integration library, `jboss-seam-resteasy.jar`.

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 JBoss Tools 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.

On startup, all classes annotated `@javax.ws.rs.Path` will be discovered automatically and registered as HTTP resources. Seam automatically accepts and serves HTTP requests with its built-in `SeamResourceServlet`. The URI of a resource is build as follows:

- The URI starts with the host and context path of your application, e.g. `http://your.hostname/myapp`.
- Then the pattern mapped in `web.xml` for the `SeamResourceServlet`, e.g `/seam/resource` if you follow the common examples, is appended. Change this setting to expose your RESTful resources under a different base. Note that this is a global change and other Seam resources (e.g. `s:graphicImage` and `s:captcha`) are then also served under that base path.
- The RESTEasy integration for Seam then appends a configurable string to the base path, by default this is `/rest`. Hence, the full base path of your resources would e.g. be `/myapp/seam/resource/rest`. We recommend that you change this string in your application (details below). You could for example add a version number to prepare for a future REST API upgrade of your services (old clients would keep the old URI base): `/myapp/seam/resource/restv1`.
- Finally, the actual resource is available under the defined `@Path`, e.g. a resource mapped with `@Path("/customer")` would be available under `/myapp/seam/resource/rest/customer`.

As an example, the following resource definition would return a plaintext representation for any GET requests using the URI `http://your.hostname/myapp/seam/resource/rest/customer/123`:

```
@Path("/customer")
public class MyCustomerResource {

   @GET
   @Path("/{customerId}")
   @Produces("text/plain")
   public String getCustomer(@PathParam("customerId") int id) {
       return ...;
   }

}
```

No additional configuration is required; you do not have to edit `web.xml` or any other setting if these defaults are acceptable. However, you can configure RESTEasy in your Seam application. First import the `resteasy` namespace into your XML configuration (`components.xml`) file header:

```
<components
  xmlns="http://jboss.org/schema/seam/components"
  xmlns:resteasy="http://jboss.org/schema/seam/resteasy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
   "http://jboss.org/schema/seam/resteasy
       http://jboss.org/schema/seam/resteasy-2.3.xsd
    http://jboss.org/schema/seam/components
       http://jboss.org/schema/seam/components-2.3.xsd">
```

You can then change the `/rest` prefix as mentioned earlier:

```
<resteasy:application resource-path-prefix="/restv1"/>
```

The full base path to your resources is now `/myapp/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 interface.

Seam will scan your classpath for any deployed `@javax.ws.rs.Path` resources and any `@javax.ws.rs.ext.Provider` classes. You can disable scanning and configure these classes manually:

```
<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. We recommend you leave them enabled, as they provide plaintext, JSON, and JAXB marshalling out of the box.

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 additional `Accept` and `Accept-Language` header values `text/plain` and `de-DE`.

25.4.2. Resources as Seam components

Any resource and provider instances are managed by RESTEasy by default. That means 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 lifecycle. Providers are instantiated once for the whole application and are effectively singletons and supposed to be stateless.

You can write resources as Seam components and benefit from the richer lifecycle management of Seam, and interception for bijection, security, and so on. Simply make your resource class a Seam component:

```
@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();
   }

}
```

An instance of `customerResource` is now handled by Seam when a request hits the server. This is a Seam JavaBean component that is `EVENT`-scoped, hence no different than the default JAX-RS lifecycle. You get full Seam injection and interception support, and all other Seam components and contexts are available to you. Currently also supported are `APPLICATION` and `STATELESS` resource Seam components. 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 mapping of conversations to temporary HTTP resources and paths is planned but currently not supported.

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`.

Sub-resources as defined in the JAX RS specification, section 3.4.1, can also be Seam component instances:

```
@Path("/garage")
@Name("garage")
public class GarageService
{
  ...

  @Path("/vehicles")
  public VehicleService getVehicles() {
    return (VehicleService) Component.getInstance(VehicleService.class);
  }
}
```

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

Provider classes can currently not be Seam components. Although you can configure an `@Provider` annotated class as a Seam component, it will at runtime be managed by RESTEasy as a singleton with no Seam interception, bijection, etc. The instance will not be a Seam component instance. We plan to support Seam component lifecycle for JAX-RS providers in the future.

25.4.3. 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.

25.4.4. Mapping exceptions to HTTP responses

Section 3.3.4 of the JAX-RS specification defines how checked or unchecked exceptions are handled by the JAX RS implementation. In addition to using an exception mapping provider as defined by JAX-RS, the integration of RESTEasy with Seam allows you to map exceptions to HTTP response codes within Seam's `pages.xml` facility. If you are already using `pages.xml` declarations, this is easier to maintain than potentially many JAX RS exception mapper classes.

Exception handling within Seam requires that the Seam filter is executed for your HTTP request. Ensure that you do filter *all* requests in your `web.xml`, not - as some Seam examples might show - a request URI pattern that doesn't cover your REST request paths. 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 the same:

```
<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 an HTTP error to the client if an exception occurs. Seam allows you to map the exception as a redirect to a view of your Seam application. As this feature is typically used for human clients (web browsers) and not for REST API remote clients, you should pay extra attention to conflicting exception mappings in `pages.xml`.

Note that the HTTP response still passes through the servlet container, so an additional mapping might 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`!

25.4.5. 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 14, The Seam Application Framework*). These components allow you to bind domain model entity classes to an HTTP API.

25.4.5.1. 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.

25.4.5.2. 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 25.1.

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.

25.4.6. 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`.

Remoting

Seam provides a convenient method of remotely accessing components from a web page, using AJAX (Asynchronous Javascript and XML). The framework for this functionality is provided with almost no up-front development effort - your components only require simple annotating to become accessible via AJAX. This chapter describes the steps required to build an AJAX-enabled web page, then goes on to explain the features of the Seam Remoting framework in more detail.

26.1. Configuration

To use remoting, the Seam Resource servlet must first be configured 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>
```

The next step is to import the necessary Javascript into your web page. There are a minimum of two scripts that must be imported. The first one contains all the client-side framework code that enables remoting functionality:

```
<script type="text/javascript" src="seam/resource/remoting/resource/remote.js"></script>
```

The second script contains the stubs and type definitions for the components you wish to call. It is generated dynamically based on the local interface of your components, and includes type definitions for all of the classes that can be used to call the remotable methods of the interface. The name of the script reflects the name of your component. For example, if you have a stateless session bean annotated with `@Name("customerAction")`, then your script tag should look like this:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction"></script>
```

If you wish to access more than one component from the same page, then include them all as parameters of your script tag:

```
<script type="text/javascript"
      src="seam/resource/remoting/interface.js?customerAction&accountAction"></script>
```

Alternatively, you may use the `s:remote` tag to import the required Javascript. Separate each component or class name you wish to import with a comma:

```
<s:remote include="customerAction,accountAction"/>
```

26.2. The "Seam" object

Client-side interaction with your components is all performed via the `Seam` Javascript object. This object is defined in `remote.js`, and you'll be using it 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.

26.2.1. A Hello World example

Let's step through a simple example to see how the `Seam` object works. First of all, let's create a new Seam component called `helloAction`.

```
@Stateless
@Name("helloAction")
public class HelloAction {
   @WebRemote
   public String sayHello(String name) {
      return "Hello, " + name;
   }
}
```

Take special note of the `@WebRemote` annotation, as it's required to make our method accessible via remoting:

That's all the server-side code we need to write.

Note

If you are performing a persistence operation in the method marked `@WebRemote` you will also need to add a `@Transactional` annotation to the method. Otherwise,

your method would execute outside of a transaction without this extra hint.That's because unlike a JSF request, Seam does not wrap the remoting request in a transaction automatically.

Now for our web page - create a new page and import the `helloAction` component:

```
<s:remote include="helloAction"/>
```

To make this a fully interactive user experience, let's add a button to our page:

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

We'll also need to add some more script to make our button actually do something when it's clicked:

```
<script type="text/javascript">
  //<![CDATA[

  function sayHello() {
    var name = prompt("What is your name?");
    Seam.Component.getInstance("helloAction").sayHello(name, sayHelloCallback);
  }

  function sayHelloCallback(result) {
    alert(result);
  }

   // ]]>
</script>
```

We're done! 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 that the call was successful. If you want to save some time, you'll find the full source code for this Hello World example in Seam's `/examples/remoting/helloworld` directory.

So what does the code of our script actually do? Let's break it down into smaller pieces. To start with, you can see from the Javascript code listing that we have implemented two methods - the first method is responsible for prompting the user for their name and then making a remote request. Take a 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. We can invoke the methods of our component against this stub, which is exactly what happens with the remainder of the line: `sayHello(name, sayHelloCallback);`.

What this line of code in its completeness does, is invoke the `sayHello` method of our component, passing in `name` as a parameter. The second parameter, `sayHelloCallback` isn't a parameter of our component's `sayHello` method, instead it tells the Seam Remoting framework that once it receives the response to our request, it should pass it to the `sayHelloCallback` Javascript method. This callback parameter is entirely optional, so feel free to leave it out if you're calling a method with a `void` return type or if you don't care about the result.

The `sayHelloCallback` method, once receiving the response to our remote request then pops up an alert message displaying the result of our method call.

26.2.2. 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 in the following sections however their main difference is that `newInstance()` will always create a new instance of a component type, and `getInstance()` will return a singleton instance.

26.2.2.1. Seam.Component.newInstance()

Use this method to create a new instance of an entity or Javabean component. The object returned by this method will have the same getter/setter methods as its server-side counterpart, or alternatively if you wish you can access its fields directly. Take the following Seam entity component 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");
```

Then from here you can set the fields of the customer object:

```
customer.setFirstName("John");
// Or you can set the fields directly
customer.lastName = "Smith";
```

26.2.2.2. Seam.Component.getInstance()

The `getInstance()` method is used to get a reference 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 our example from before, if we have created a new `customer` and we now wish to save it, we would pass it to the `saveCustomer()` method of our `customerAction` component:

```
Seam.Component.getInstance("customerAction").saveCustomer(customer);
```

26.2.2.3. Seam.Component.getComponentName()

Passing an object into this method will return its 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");
```

26.2.3. Seam.Remoting

Most of the client side functionality for Seam Remoting is contained within the `Seam.Remoting` object. While you shouldn't need to directly call most of its methods, there are a couple of important ones worth mentioning.

26.2.3.1. Seam.Remoting.createType()

If your application contains or uses Javabean classes that aren't 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");
```

26.2.3.2. Seam.Remoting.getTypeName()

This method is the equivalent of `Seam.Component.getComponentName()` but for non-component types. It will return 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.

26.3. Client Interfaces

In the configuration section above, the interface, or "stub" for our component is imported into our page either via `seam/resource/remoting/interface.js`: or using the `s:remote` tag:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction"></script>
```

```
<s:remote include="customerAction"/>
```

By including this script in our page, the interface definitions for our component, plus any other components or types that are required to execute the methods of our component are generated and made available for the remoting framework to use.

There are two types of client stub that can be generated, "executable" stubs and "type" stubs. Executable stubs are behavioural, and are used to execute methods against your session bean components, while type stubs contain state and represent the types that can be passed in as parameters or returned as a result.

The type of client stub that is generated depends on the type of your Seam component. If the component is a session bean, then an executable stub will be generated, otherwise if it's an entity or JavaBean, then a type stub will be generated. There is one exception to this rule; if your component is a JavaBean (ie it is not a session bean nor an entity bean) and any of its methods are annotated with @WebRemote, then an executable stub will be generated for it instead of a type stub. This allows you to use remoting to call methods of your JavaBean components in a non-EJB environment where you don't have access to session beans.

26.4. The Context

The Seam Remoting Context contains additional information which is sent and received as part of a remoting request/response cycle. At this stage it only contains the conversation ID but may be expanded in the future.

26.4.1. Setting and reading the Conversation ID

If you intend on using remote calls within the scope of a conversation then you need to 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 hasn't been explicitly set with `Seam.Remoting.getContext().setConversationId()`, then it will be automatically assigned the first valid conversation ID that is returned by any remoting call. If you are working with multiple conversations within your page, then you may need to explicitly set the conversation ID before each call. If you are working with just a single conversation, then you don't need to do anything special.

26.4.2. Remote calls within the current conversation scope

In some circumstances it may be required to make a remote call within the scope of the current view's conversation. To do this, you must explicitly set the conversation ID to that of the view

before making the remote call. This small snippet of JavaScript will set the conversation ID that is used for remoting calls to the current view's conversation ID:

```
Seam.Remoting.getContext().setConversationId( #{conversation.id} );
```

26.5. Batch Requests

Seam Remoting allows multiple component calls to be executed within a single request. It is recommended that this feature is used wherever it is appropriate to reduce network traffic.

The method `Seam.Remoting.startBatch()` will start a new batch, and 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 will send 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 containning all return values will be returned to the client and the callback functions (if provided) triggered in the same order as execution.

If you start a new batch via the `startBatch()` method but then decide you don't want to send it, the `Seam.Remoting.cancelBatch()` method will discard any calls that were queued and exit the batch mode.

To see an example of a batch being used, take a look at `/examples/remoting/chatroom`.

26.6. Working with Data types

26.6.1. 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.

26.6.1.1. String

Simply use Javascript String objects when setting String parameter values.

26.6.1.2. Number

There is support for all number types supported by Java. On the client side, number values are always serialized as their String representation and then on the server side they are converted to the correct destination type. Conversion into either a primitive or wrapper type is supported for `Byte`, `Double`, `Float`, `Integer`, `Long` and `Short` types.

26.6.1.3. Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

26.6.2. JavaBeans

In general these will be either Seam entity or JavaBean components, or some other non-component class. Use the appropriate method (either `Seam.Component.newInstance()` for Seam components or `Seam.Remoting.createType()` for everything else) to create a new instance of the object.

It is important to note that only objects that are created by either of these two methods should be used as parameter values, where the parameter is not one of the other valid types mentioned anywhere else in this section. In some situations you may have a component method 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 you might want to pass in an instance of your `myWidget` component, however the interface for `myAction` won't include `myWidget` as it is not directly referenced by any of its methods. To get around this, `MyWidget` needs to be explicitly imported:

```
<s:remote include="myAction,myWidget"/>
```

This will then allow a `myWidget` object to be created with `Seam.Component.newInstance("myWidget")`, which can then be passed to `myAction.doSomethingWithObject()`.

26.6.3. 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` (or descendent, such as `java.sql.Date` or `java.sql.Timestamp` class.

26.6.4. Enums

On the client side, enums are treated the same as Strings. When setting the value for an enum parameter, simply 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.

26.6.5. Collections

26.6.5.1. Bags

Bags cover all collection types including arrays, collections, lists, sets, (but excluding Maps - see the next section for those), and are implemented client-side as a Javascript array. When calling a component method that accepts one of these types as a parameter, your parameter should be a Javascript array. If a component method returns one of these types, then the return value will also be a Javascript array. The remoting framework is clever enough on the server side to convert the bag to an appropriate type for the component method call.

26.6.5.2. Maps

As there is no native support for Maps within Javascript, a simple Map implementation is provided with the Seam Remoting framework. To create a Map which 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 are equivalent to their Java counterpart. Where the method returns a collection, such as `keySet()` and `values()`, a Javascript Array object will be returned that contains the key or value objects (respectively).

26.7. Debugging

To aid in tracking down bugs, it is possible to enable a debug mode which will display the contents of all the packets send back and forth between the client and server in a popup window. To enable debug mode, either execute the `setDebug()` method in Javascript:

```
Seam.Remoting.setDebug(true);
```

Or configure it via components.xml:

```
<remoting:remoting debug="true"/>
```

To turn off debugging, call `setDebug(false)`. If you want to write your own messages to the debug log, call `Seam.Remoting.log(message)`.

26.8. Handling Exceptions

When invoking a remote component method, it is possible to specify an exception handler which will 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()` that returns the exception message which is produced by the exception thrown by the `@WebRemote` method.

26.9. The Loading Message

The default loading message that appears in the top right corner of the screen can be modified, its rendering customised or even turned off completely.

26.9.1. Changing the message

To change the message from the default "Please Wait..." to something different, set the value of `Seam.Remoting.loadingMessage`:

```
Seam.Remoting.loadingMessage = "Loading...";
```

26.9.2. Hiding the loading message

To completely suppress the display of the loading message, override the implementation of `displayLoadingMessage()` and `hideLoadingMessage()` with functions that instead do nothing:

```
// don't display the loading indicator
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

26.9.3. 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 this override the `displayLoadingMessage()` and `hideLoadingMessage()` messages with your own implementation:

```
Seam.Remoting.displayLoadingMessage = function() {
  // Write code here to display the indicator
};

Seam.Remoting.hideLoadingMessage = function() {
  // Write code here to hide the indicator
};
```

26.10. Controlling what data is returned

When a remote method is executed, the result is serialized into an XML response that is returned to the client. This response is then unmarshaled by the client into a Javascript object. For complex types (i.e. Javabeans) that include references to other objects, all of these referenced objects are also serialized as part of the response. These objects may reference other objects, which may reference other objects, and so forth. If left unchecked, this object "graph" could potentially be enormous, depending on what relationships exist between your objects. And as a side issue (besides the potential verbosity of the response), you might also wish to prevent sensitive information from being exposed to the client.

Seam Remoting provides a simple means to "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 using 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.

For all our examples, we'll use the following `Widget` class:

```
@Name("widget")
public class Widget
{
  private String value;
  private String secret;
  private Widget child;
  private Map<String,Widget> widgetMap;
  private List<Widget> widgetList;

  // getters and setters for all fields
}
```

26.10.1. Constraining normal fields

If your remote method returns an instance of `Widget`, but you don't want to expose the `secret` field because it contains sensitive information, you would constrain it like this:

```
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

The value "secret" refers to the `secret` field of the returned object. Now, suppose that we don't care about exposing this particular field to the client. Instead, notice that the `Widget` value that is returned has a field `child` that is also a `Widget`. What if we want to hide the `child`'s `secret` value instead? We can do this by using dot notation to specify this field's path within the result's object graph:

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

26.10.2. Constraining Maps and Collections

The other place that objects can exist within an object graph are within a `Map` or some kind of collection (`List`, `Set`, `Array`, etc). Collections are easy, and are treated like any other field. For

example, if our `Widget` contained a list of other `Widgets` in its `widgetList` field, to constrain the `secret` field of the `Widgets` in this list the annotation would look like this:

```
@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 will constrain the `Map`'s key object values, while `[value]` will constrain 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();
```

26.10.3. Constraining objects of a specific type

There is one last notation that can be used to constrain the fields of a type of object no matter where in the result's object graph it appears. This notation uses either the name of the component (if the object is a Seam component) or the fully qualified class name (only if the object is not a Seam component) and is expressed using square brackets:

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

26.10.4. 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();
```

26.11. Transactional Requests

By default there is no active transaction during a remoting request, so if you wish to perform database updates during a remoting request, you need to annotate the `@WebRemote` method with `@Transactional`, like so:

```
@WebRemote @Transactional(TransactionPropagationType.REQUIRED)
```

```
public void updateOrder(Order order) {
  entityManager.merge(order);
}
```

26.12. JMS Messaging

Seam Remoting provides experimental support for JMS Messaging. This section describes the JMS support that is currently implemented, but please note that this may change in the future. It is currently not recommended that this feature is used within a production environment.

26.12.1. Configuration

Before you can subscribe to a JMS topic, you must first configure a list of the topics that can be subscribed to by Seam Remoting. 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"/>
```

26.12.2. 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 first being the name of the JMS Topic to subscribe to, the second being the callback function to invoke when a message is received.

There are two types of messages supported, Text messages and Object messages. If you need to test for the type of message that is passed to your callback function you can use the `instanceof` operator to test whether the message is a `Seam.Remoting.TextMessage` or `Seam.Remoting.ObjectMessage`. A `TextMessage` contains the text value in its `text` field (or alternatively call `getText()` on it), while an `ObjectMessage` contains its object value in its `value` field (or call its `getValue()` method).

26.12.3. Unsubscribing from a Topic

To unsubscribe from a topic, call `Seam.Remoting.unsubscribe()` and pass in the topic name:

```
Seam.Remoting.unsubscribe("topicName");
```

26.12.4. Tuning the Polling Process

There are two parameters which you can modify to control how polling occurs. The first one is `Seam.Remoting.pollInterval`, which controls how long to wait between subsequent polls for new messages. This parameter is expressed in seconds, and its default setting is 10.

The second parameter is `Seam.Remoting.pollTimeout`, and is also expressed as 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 then an empty response will be immediately returned.

Caution should be used when setting a high `pollTimeout` value; each request that has to wait for a message means that a server thread is tied up until a message is received, or until the request times out. If many such requests are being served simultaneously, it could mean a large number of threads become tied up because of this reason.

It is recommended that you set these options via components.xml, however they can be overridden via Javascript if desired. The following example demonstrates how to configure the polling to occur much more aggressively. You should set these parameters to suitable values for your application:

Via components.xml:

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

Via JavaScript:

```
// 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;
```

Seam and the Google Web Toolkit

For those that prefer to use the Google Web Toolkit (GWT) to develop dynamic AJAX applications, Seam provides an integration layer that allows GWT widgets to interact directly with Seam components.

To use GWT, we assume that you are already familiar with the GWT tools - more information can be found at *http://code.google.com/webtoolkit/*. This chapter does not attempt to explain how GWT works or how to use it.

27.1. Configuration

There is no special configuration required to use GWT in a Seam application, however the Seam resource servlet must be installed. See *Chapter 31, Configuring Seam and packaging Seam applications* for details.

27.2. Preparing your component

The first step in preparing a Seam component to be called via GWT, is to create both synchronous and asynchronous service interfaces for the methods you wish to call. Both of these 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 that it also contains 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 example `MyServiceAsync`, will be 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 shouldn't happen, I checked on the client, " +
      "but its 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 name of the seam component *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. The methods that are to be made accessible via GWT also need to be annotated with the `@WebRemote` annotation.

27.3. Hooking up a GWT widget to the Seam component

The next step, is to 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:

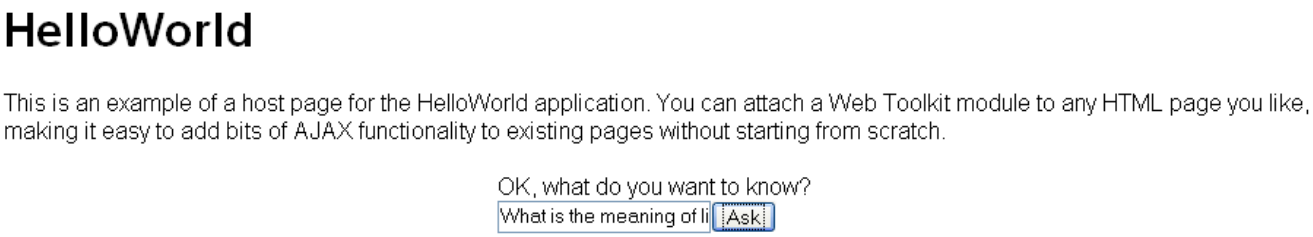
```
private MyServiceAsync getService() {
  String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

  MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
  ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
  return svc;
}
```

The final step is to 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, the button invokes the `askServer()` method passing the contents of the input text (in this example, validation is also performed to ensure 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.

27.4. GWT Ant Targets

For deployment of GWT apps, there is a compile-to-Javascript step (which compacts and obfuscates the code). There is an ant utility which can be used instead of the command line or GUI utility that GWT provides. To use this, you will need to have the ant task jar in your ant classpath, as well as GWT downloaded (which you will need for hosted mode anyway).

Then, in your ant file, place (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, which has the contents:

```
gwt.home=/gwt_home_dir
```

This of course should point to the directory where GWT is installed. Then to use it, create a target:

```
<!-- the following are 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>
```

This target when called will compile the GWT application, and copy it to the specified directory (which would be in the `webapp` part of your war - remember GWT generates HTML and Javascript artifacts). You never edit the resulting code that `gwt-compile` generates - you always edit in the GWT source directory.

Remember that GWT comes with a hosted mode browser - you should be using that if you are developing with GWT. If you aren't using that, and are just compiling it each time, you aren't getting the most out of the toolkit (in fact, if you can't or won't use the hosted mode browser, I would go far as to say you should NOT be using GWT at all - it's that valuable!).

27.5. GWT Maven plugin

For a deployment of GWT apps, there is a set of maven GWT goals which does everything what GWT supports. The maven-gwt-plugin usage is in more details at *GWT* [http://mojo.codehaus.org/gwt-maven-plugin/].

Basic set up is for instance here:

```
    <build>
      <plugins>
        [...]
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>gwt-maven-plugin</artifactId>
            <version>1.2</version> <!-- version 1.2 allows us to specify gwt version by gwt-user
 dependency -->
          <configuration>
                    <generateDirectory>${project.build.outoutDirectory}/seam-reference-guide</
generateDirectory>
            <inplace>false</inplace>
            <logLevel>TRACE</logLevel>
            <extraJvmArgs>-Xmx512m -DDEBUG</extraJvmArgs>
            <soyc>false</soyc>
          </configuration>
          <executions>
            <execution>
```

```
                <goals>
                    <goal>resources</goal>
                    <goal>compile</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    [...]
  </plugins>
    [...]
</build>
```

More can be seen here *http://mojo.codehaus.org/gwt-maven-plugin/user-guide/compile.html*

Spring Framework integration

The Spring integration (part of the Seam IoC module) allows easy migration of Spring-based projects to Seam and allows Spring applications to take advantage of key Seam features like conversations and Seam's more sophisticated persistence context management.

Note! The Spring integration code is included in the jboss-seam-ioc library. This dependency is required for all seam-spring integration techniques covered in this chapter.

Seam's support for Spring provides the ability to:

- inject Seam component instances into Spring beans
- inject Spring beans into Seam components
- turn Spring beans into Seam components
- allow Spring beans to live in any Seam context
- start a spring WebApplicationContext with a Seam component
- Support for Spring PlatformTransactionManagement
- provides a Seam managed replacement for Spring's `OpenEntityManagerInViewFilter` and `OpenSessionInViewFilter`
- Support for Spring `TaskExecutors` to back `@Asynchronous` calls

28.1. Injecting Seam components into Spring beans

Injecting Seam component instances into Spring beans is accomplished using the `<seam:instance/>` namespace handler. To enable the Seam namespace handler, the Seam namespace must be added to the Spring beans definition file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:seam="http://jboss.org/schema/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.org/schema/seam/spring-seam
                       http://jboss.org/schema/seam/spring-seam-2.3.xsd">
```

Now any Seam component may be injected into any Spring bean:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
```

```
  <property name="someProperty">
    <seam:instance name="someComponent"/>
  </property>
</bean>
```

An EL expression may be used instead of a component name:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="#{someExpression}"/>
  </property>
</bean>
```

Seam component instances may even be made available for injection into Spring beans by a Spring bean id.

```
<seam:instance name="someComponent" id="someSeamComponentInstance"/>

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty" ref="someSeamComponentInstance">
</bean>
```

Now for the caveat!

Seam was designed from the ground up to support a stateful component model with multiple contexts. Spring was not. Unlike Seam bijection, Spring injection does not occur at method invocation time. Instead, injection happens only when the Spring bean is instantiated. So the instance available when the bean is instantiated will be the same instance that the bean uses for the entire life of the bean. For example, if a Seam `CONVERSATION`-scoped component instance is directly injected into a singleton Spring bean, that singleton will hold a reference to the same instance long after the conversation is over! We call this problem *scope impedance*. Seam bijection ensures that scope impedance is maintained naturally as an invocation flows through the system. In Spring, we need to 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>
```

This example shows one way to use a Seam-managed persistence context from a Spring bean. (For a more robust way to use Seam-managed persistence contexts as a replacement for the Spring `OpenEntityManagerInView` filter see section on *Using a Seam Managed Persistence Context in Spring*)

28.2. Injecting Spring beans into Seam components

It is even easier to inject Spring beans into Seam component instances. Actually, there are two possible approaches:

- inject a Spring bean using an EL expression
- make the Spring bean a Seam component

We'll discuss the second option in the next section. The easiest approach is to access the Spring beans via EL.

The Spring `DelegatingVariableResolver` is an integration point Spring provides for integrating Spring with JSF. This `VariableResolver` makes all Spring beans available in EL by their bean id. You'll need to add the `DelegatingVariableResolver` to `faces-config.xml`:

```
<application>
   <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
   </variable-resolver>
</application>
```

Then you can inject Spring beans using `@In`:

```
@In("#{bookingService}")
private BookingService bookingService;
```

The use of Spring beans in EL is not limited to injection. Spring beans may be used anywhere that EL expressions are used in Seam: process and pageflow definitions, working memory assertions, etc...

28.3. Making a Spring bean into a Seam component

The `<seam:component/>` namespace handler can be used to make any Spring bean a Seam component. Just place the `<seam:component/>` tag within the declaration of the bean that you wish to be a Seam component:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
   <seam:component/>
</bean>
```

By default, `<seam:component/>` will create a `STATELESS` Seam component with class and name provided in the bean definition. Occasionally, such as when a `FactoryBean` is used, the class of the Spring bean may not be the class appearing in the bean definition. In such cases the `class` should be explicitly specified. A Seam component name may be explicitly specified in cases where there is potential for a naming conflict.

The `scope` attribute of `<seam:component/>` may be used if you wish the Spring bean to be managed in a particular Seam scope. The Spring bean must be scoped to `prototype` if the Seam scope specified is anything other than `STATELESS`. Pre-existing Spring beans usually have a fundamentally stateless character, so this attribute is not usually needed.

28.4. Seam-scoped Spring beans

The Seam integration package also lets you use Seam's contexts as Spring 2.0 style custom scopes. This lets you declare any Spring bean in any of Seam's contexts. However, note once again that Spring's component model was never architected to support statefulness, so please use this feature with great care. In particular, clustering of session or conversation scoped Spring beans is deeply problematic, and care must be taken when injecting a bean or component from a wider scope into a bean of a narrower scope.

By specifying `<seam:configure-scopes/>` once in a Spring bean factory configuration, all of the Seam scopes will be available to Spring beans as custom scopes. To associate a Spring bean with a particular Seam scope, specify the Seam 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"/
>
```

The prefix of the scope name may be changed by specifying the `prefix` attribute in the `configure-scopes` definition. (The default prefix is `seam.`)

By default an instance of a Spring Component registered in this way is not automatically created when referenced using `@In`. To have an instance auto-created you must either specify `@In(create=true)` at the injection point to identify a specific bean to be auto created or you can

use the `default-auto-create` attribute of `configure-scopes` to make all spring beans who use a seam scope auto created.

Seam-scoped Spring beans defined this way can be injected into other Spring beans without the use of `<seam:instance/>`. However, care must be taken to ensure scope impedance is maintained. The normal approach used in Spring is to specify `<aop:scoped-proxy/>` in the bean definition. However, Seam-scoped Spring beans are *not* compatible with `<aop:scoped-proxy/>`. So if you need to inject a Seam-scoped Spring bean into a singleton, `<seam:instance/>` must be used:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="seam.CONVERSATION"/
>

...

<bean id="someSingleton">
   <property name="someSeamScopedSpringBean">
      <seam:instance name="someSpringBean" proxy="true"/>
   </property>
</bean>
```

28.5. Using Spring PlatformTransactionManagement

Spring provides an extensible transaction management abstraction with support for many transaction APIs (JPA, Hibernate, JDO, and JTA) Spring also provides tight integrations with many application server TransactionManagers such as Websphere and Weblogic. Spring transaction management exposes support for many advanced features such as nested transactions and supports full Java EE transaction propagation rules like REQUIRES_NEW and NOT_SUPPORTED. For more information see the spring documentation *here* [http://static.springframework.org/spring/docs/2.0.x/reference/transaction.html].

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 utilize Springs transaction synchronization capabilities for synchronization callbacks.

28.6. Using a Seam Managed Persistence Context in Spring

One of the most powerful features of Seam is its conversation scope and the ability to have an EntityManager open for the life of a conversation. This eliminates many of the problems associated with the detachment and re-attachment of entities as well as mitigates occurrences of the dreaded `LazyInitializationException`. Spring does not provide a way to manage an persistence context beyond the scope of a single web request (`OpenEntityManagerInViewFilter`). So, it would be nice if Spring developers could have access to a Seam managed persistence context using all of the same tools Spring provides for integration with JPA(e.g. `PersistenceAnnotationBeanPostProcessor`, `JpaTemplate`, etc.)

Seam provides a way for Spring to access a Seam managed persistence context with Spring's provided JPA tools bringing conversation scoped persistence context capabilities to Spring applications.

This integration work provides the following functionality:

- transparent access to a Seam managed persistence context using Spring provided tools
- access to Seam conversation scoped persistence contexts in a non web request (e.g. asynchronous quartz job)
- allows for using Seam managed persistence contexts with Spring managed transactions (will need to flush the persistence context manually)

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.

```
<bean id="seamEntityManagerFactory" class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBea
   <property name="persistenceContextName" value="entityManager"/>
</bean>
```

Where 'persistenceContextName' is the name of the Seam managed persistence context component. By default this EntityManagerFactory has a unitName equal to the Seam component name or in this case 'entityManager'. If you wish to provide a different unitName you can do so by providing a persistenceUnitName like so:

```
<bean id="seamEntityManagerFactory" class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBea
   <property name="persistenceContextName" value="entityManager"/>
   <property name="persistenceUnitName" value="bookingDatabase:extended"/>
```

```
</bean>
```

This EntityManagerFactory can then be used in any Spring provided tools. For example, using Spring's `PersistenceAnnotationBeanPostProcessor` is the exact same as before.

```
<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` which persistenctUnitName you wish to use by default 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.SeamManagedEntityManagerFactoryBea
   <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` are configured the same way for a Seam managed persistence context as they would be fore a Seam managed persistence context.

```
<bean id="bookingService" class="org.jboss.seam.example.spring.BookingService">
   <property name="entityManagerFactory" ref="seamEntityManagerFactory"/>
</bean>
```

28.7. Using a Seam Managed Hibernate Session in Spring

The Seam Spring integration also provides support for complete access to a Seam managed Hibernate session using spring's tools. This integration is very similar to the *JPA integration*.

Like Spring's JPA integration spring's propagation model allows only one open EntityManager per EntityManagerFactory per transaction??? to be available to spring tools. So, the Seam Session integration works 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>
```

Where 'sessionName' is the name of the `persistence:managed-hibernate-session` component. This SessionFactory can then be used in any Spring provided tools. The integration also provides support for calls to `SessionFactory.getCurrentInstance()` as long as you call getCurrentInstance() on the `SeamManagedSessionFactory`.

28.8. 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 a couple of limitations.

- the Spring ApplicationContext must be started *after* the `SeamListener`
- it can be tricky starting a Spring ApplicationContext for use in Seam unit and integration tests

To overcome these two limitations the Spring integration includes a Seam component that will start a Spring ApplicationContext. To use this Seam component place the `<spring:context-loader/>` definition in the `components.xml`. Specify your Spring context file location in the `config-locations` attribute. If more than one config file is needed you can place them in the nested `<spring:config-locations/>` element following standard `components.xml` multi value practices.

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:spring="http://jboss.org/schema/seam/spring"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jboss.org/schema/seam/components
                    http://jboss.org/schema/seam/components-2.3.xsd
                    http://jboss.org/schema/seam/spring
                    http://jboss.org/schema/seam/spring-2.3.xsd">
```

```
    <spring:context-loader config-locations="/WEB-INF/applicationContext.xml"/>

</components>
```

28.9. Using a Spring TaskExecutor for @Asynchronous

Spring provides an abstraction for executing code asynchronously called a `TaskExecutor`. The Spring Seam integration allows for the use of a Spring `TaskExecutor` for executing 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 of an asynchronous event a fallback Seam `Dispatcher` can be provided to handle scheduled asynchronous event 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}"/>
```

Guice integration

Google Guice is a library that provides lightweight dependency injection through type-safe resolution. The Guice integration (part of the Seam IoC module) allows use of Guice injection for all Seam components annotated with the `@Guice` annotation. In addition to the regular bijection that Seam performs (which becomes optional), Seam also delegates to known Guice injectors to satisfy the dependencies of the component. Guice may be useful to tie non-Seam parts of large or legacy applications together with Seam.

Note

The Guice integration is bundled in the jboss-seam-ioc library. This dependency is required for all integration techniques covered in this chapter. You will also need the Guice JAR file on the classpath.

29.1. Creating a hybrid Seam-Guice component

The goal is to create a hybrid Seam-Guice component. The rule for how to do this is very simple. If you want to use Guice injection in your Seam component, annotate it with the `@Guice` annotation (after importing the type `org.jboss.seam.ioc.guice.Guice`).

```
@Name("myGuicyComponent")
@Guice public class MyGuicyComponent
{
   @Inject MyObject myObject;
   @Inject @Special MyObject mySpecialObject;
   ...
}
```

This Guice injection will happen on every method call, just like with bijection. Guice injects based on type and binding. To satisfy the dependencies in the previous example, you might have bound the following implementations in a Guice module, where `@Special` is an annotation you define in your application.

```
public class MyGuicyModule implements Module
{
   public void configure(Binder binder)
   {
      binder.bind(MyObject.class)
         .toInstance(new MyObject("regular"));
```

```
    binder.bind(MyObject.class).annotatedWith(Special.class)
      .toInstance(new MyObject("special"));
  }
}
```

Great, but which Guice injector will be used to inject the dependencies? Well, you need to perform some setup first.

29.2. Configuring an injector

You tell Seam which Guice injector to use by hooking it into the injection property of the Guice initialization component in the Seam component descriptor (components.xml):

```
<components xmlns="http://jboss.org/schema/seam/components"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:guice="http://jboss.org/schema/seam/guice"
  xsi:schemaLocation="
    http://jboss.org/schema/seam/guice
    http://jboss.org/schema/seam/guice-2.3.xsd
    http://jboss.org/schema/seam/components
    http://jboss.org/schema/seam/components-2.3.xsd">

  <guice:init injector="#{myGuiceInjector}"/>

</components>
```

`myGuiceInjector` must resolve to a Seam component that implements the Guice `Injector` interface.

Having to create an injector is boiler-plate code, though. What you really want to be able to do is simply hook up Seam to your Guice modules. Fortunately, there is a built-in Seam component that implements the `Injector` interface to do exactly that. You can configure it in the Seam component descriptor with this additional stanza.

```
<guice:injector name="myGuiceInjector">
  <guice:modules>
    <value>com.example.guice.GuiceModule1</value>
    <value>com.example.guice.GuiceModule2</value>
  </guice:modules>
</guice:injector>
```

Of course you can also use an injector that is already used in other, possibly non-Seam part of you application. That's one of the main motivations for creating this integration. Since the injector is defined with EL expression, you can obtain it in whatever way you like. For instance, you may use the Seam factory component pattern to provide injector.

```
@Name("myGuiceInjectorFactory")
public InjectorFactory
{
  @Factory(name = "myGuiceInjector", scope = APPLICATION, create = true)
  public Injector getInjector()
  {
    // Your code that returns injector
  }
}
```

29.3. Using multiple injectors

By default, an injector configured in the Seam component descriptor is used. If you really need to use multiple injectors (AFAIK, you should use multiple modules instead), you can specify different injector for every Seam component in the `@Guice` annotation.

```
@Name("myGuicyComponent")
@Guice("myGuiceInjector")
public class MyGuicyComponent
{
  @Inject MyObject myObject;
  ...
}
```

That's all there is to it! Check out the guice example in the Seam distribution to see the Seam Guice integration in action!

Hibernate Search

30.1. Introduction

Full text search engines like Apache Lucene™ are a very powerful technology that bring full text and efficient queries to applications. Hibernate Search, which uses Apache Lucene under the covers, indexes your domain model with the addition of a few annotations, takes care of the database / index synchronization and returns regular managed objects that are matched by full text queries. Keep in mind, thought, that there are mismatches that arise when dealing with an object domain model over a text index (keeping the index up to date, mismatch between the index structure and the domain model, and querying mismatch). But the benefits of speed and efficiency far outweigh these limitations.

Hibernate Search has been designed to integrate nicely and as naturally as possible with JPA and Hibernate. As a natural extension, JBoss Seam provides an Hibernate Search integration.

Please refer to the *Hibernate Search documentation* [] for information specific to the Hibernate Search project.

30.2. 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. Here is a minimal persistence unit configuration to get started.

```
<persistence-unit name="sample">
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
      value="filesystem"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
      value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit>
```

In addition to the configuration file, the following jars have to be deployed:

- hibernate-search.jar

- hibernate-search-orm.jar
- hibernate-search-engine.jar
- lucene-core.jar

Maven coordinates for using Hibernate Search:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search</artifactId>
  <version>4.1.1.Final</version>
</dependency>
```

Some Hibernate Search extensions require additional dependencies, a commonly used is `hibernate-search-analyzers.jar`. For details, see the *Hibernate Search documentation* [http://docs.jboss.org/hibernate/search/4.1/reference/en-US/html_single] for details.

Note

If you deploy those in a EAR, don't forget to update `application.xml`

30.3. Usage

Hibernate Search uses annotations to map entities to a Lucene index, check the *reference documentation* [http://docs.jboss.org/hibernate/search/4.1/reference/en-US/html_single] for more informations.

Hibernate Search is fully integrated with the API and semantic of JPA / Hibernate. Switching from a HQL or Criteria based query requires just a few lines of code. The main API the application interacts with is the `FullTextSession` API (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

`FullTextSession` extends `org.hibernate.Session` so that it can be used as a regular Hibernate Session

If the Java Persistence API is used, a smoother integration is proposed.

```
@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();
  }
  [...]
}
```

When Hibernate Search is present, a `FulltextEntityManager` is injected. `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 (i.e. via the @PersistenceContext annotation), it is not possible to replace the `EntityManager` interface by the `FullTextEntityManager` interface in the declaration statement. However, the implementation injected will be a `FullTextEntityManager` implementation: downcasting is then possible.

```
@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();
  }
  [...]
}
```

Caution

For people accustomed to Hibernate Search out of Seam, note that using `Search.getFullTextSession` is not necessary.

Check the DVDStore or the blog examples of the JBoss Seam distribution for a concrete use of Hibernate Search.

Configuring Seam and packaging Seam applications

Configuration is a very boring topic and an extremely tedious pastime. Unfortunately, several lines of XML are required to integrate Seam into your JSF implementation and servlet container. There's no need to be too put off by the following sections; you'll never need to type any of this stuff yourself, since you can just use seam-gen to start your application or you can copy and paste from the example applications!

31.1. Basic Seam configuration

First, let's look at the basic configuration that is needed whenever we use Seam with JSF.

31.1.1. Integrating Seam with JSF and your servlet container

Of course, you need 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 to suit your taste.)

In addition, Seam 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 have a broken implementation of server-side state saving that interferes with Seam's conversation propagation. If you have problems with conversation

propagation during form submissions, try switching to client-side state saving. You'll need this in `web.xml`:

```
<context-param>
   <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
   <param-value>client</param-value>
</context-param>
```

Warning

Setting of `javax.faces.STATE_SAVING_METHOD` to `client` can lead to security issues and it should be set environment entry `com.sun.faces.ClientStateSavingPassword` in `web.xml` like:

```
            <env-entry>
               <env-entry-name>com.sun.faces.ClientStateSavingPassword</env-
entry-name>
               <env-entry-type>java.lang.String</env-entry-type>
               <env-entry-value>INSERT_YOUR_PASSWORD</env-entry-value>
            </env-entry>
```

There is a minor gray area in the JSF specification regarding the mutability of view state values. Since Seam uses the JSF view state to back its PAGE scope this can become an issue in some cases. If you're using server side state saving with the JSF-RI and you want a PAGE scoped bean to keep its exact value for a given view of a page you will need to specify the following context-param. Otherwise if a user uses the "back" button a PAGE scoped component will have the latest value if it has changed not the value of the "back" page. (see *Spec Issue* [http://java.net/jira/browse/JAVASERVERFACES_SPEC_PUBLIC-1127]). This setting is not enabled by default because of the performance hit of serializing the JSF view with every request.

```
<context-param>
   <param-name>com.sun.faces.serializeServerState</param-name>
   <param-value>true</param-value>
</context-param>
```

31.1.2. 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>
```

31.1.3. Seam servlet filters

Seam doesn't need any servlet filters for basic operation. However, there are several features which depend upon the use of filters. To make things easier, Seam lets you add and configure servlet filters just like you would configure other built-in Seam components. To take advantage of this feature, we 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>
```

The Seam master filter *must* be the first filter specified in `web.xml`. This ensures it is run first.

The Seam filters share a number of common attributes, you can set these in `components.xml` in addition to any parameters discussed below:

- `url-pattern` — Used to specify which requests are filtered, the default is all requests. `url-pattern` is a Tomcat style pattern which allows a wildcard suffix.
- `regex-url-pattern` — Used to specify which requests are filtered, the default is all requests. `regex-url-pattern` is a true regular expression match for request path.

- `disabled` — Used to disable a built in filter.

Note that the patterns are matched against the URI path of the request (see `HttpServletRequest.getURIPath()`) and that the name of the servlet context is removed before matching.

Adding the master filter enables the following built-in filters.

31.1.3.1. Exception handling

This filter provides the exception mapping functionality in `pages.xml` (almost all applications will need this). It also takes care of rolling back uncommitted transactions when uncaught exceptions occur. (According to the Java EE specification, the web container should do this automatically, but we've found that this behavior cannot be relied upon in all application servers. And it is certainly not required of plain servlet engines like Tomcat.)

By default, the exception handling filter will process all requests, however this behavior may be adjusted by adding a `<web:exception-filter>` entry to `components.xml`, as shown in this example:

```
<components xmlns="http://jboss.org/schema/seam/components"
        xmlns:web="http://jboss.org/schema/seam/web">

   <web:exception-filter url-pattern="*.seam"/>

</components>
```

31.1.3.2. 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 will process all requests by default, but this behavior can also be adjusted in `components.xml`:

```
<web:redirect-filter url-pattern="*.seam"/>
```

31.1.3.3. URL rewriting

This filter allows Seam to apply URL rewriting for views based on configuration in the `pages.xml` file. This filter is not activate by default, but can be activated by adding the 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 ommitted, the rewrite filter assumes the pattern `*.seam`.

31.1.3.4. Multipart form submissions

This feature is necessary when using the Seam file upload JSF control. It detects multipart form requests and processes them according to the multipart/form-data specification (RFC-2388). To override the default settings, add the following entry to `components.xml`:

```
<web:multipart-filter create-temp-files="true"
                      max-request-size="1000000"
                      url-pattern="*.seam"/>
```

- `create-temp-files` — If set to `true`, uploaded files are written to a temporary file (instead of held in memory). This may be an important consideration if large file uploads are expected. The default setting is `false`.
- `max-request-size` — If the size of a file upload request (determined by reading the `Content-Length` header in the request) exceeds this value, the request will be aborted. The default setting is 0 (no size limit).

31.1.3.5. Character encoding

Sets the character encoding of submitted form data.

This filter 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 encoding to use.
- `override-client` — If this is set to `true`, the request encoding will be set to whatever is specified by `encoding` no matter whether the request already specifies an encoding or not. If set to `false`, the request encoding will only be set if the request doesn't already specify an encoding. The default setting is `false`.

31.1.3.6. RichFaces

If RichFaces is used in your project, Seam will install the RichFaces Ajax filter for you, making sure to install it before all other built-in filters. You don't need to install the RichFaces Ajax filter in `web.xml` yourself.

The RichFaces Ajax filter is only installed if the RichFaces jars 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 (e.g. javascript, CSS, images, etc). When developing custom javascript or CSS, setting to true prevents the browser from caching the resource.
- `log4j-init-file` — is used to setup per-application logging. A path, relative to web application context, to the log4j.xml configuration file should be provided.

31.1.3.7. Identity Logging

This filter adds the authenticated user name to the log4j mapped diagnostic context so that it can be included in formatted log output if desired, by adding %X{username} to the pattern.

By default, the logging filter will process all requests, however this behavior may be adjusted by adding a `<web:logging-filter>` entry to `components.xml`, as shown in this example:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:web="http://jboss.org/schema/seam/web">
   <web:logging-filter url-pattern="*.seam"/>
</components>
```

31.1.3.8. Context management for custom servlets

Requests sent direct to some servlet other than the JSF servlet are not processed through the JSF lifecycle, so Seam provides a servlet filter that can be applied to any other servlet that needs access to Seam components.

This filter allows custom servlets to interact with the Seam contexts. It sets up the Seam contexts at the beginning of each request, and tears them down at the end of the request. You should make sure that this filter is *never* applied to the JSF `FacesServlet`. Seam uses the phase listener for context management in a JSF request.

This filter is not installed by default and requires an entry in `components.xml` to enable it:

```
<web:context-filter url-pattern="/media/*"/>
```

The context filter expects to find the conversation id of any conversation context in a request parameter named `conversationId`. You are responsible for ensuring that it gets sent in the request.

You are also responsible for ensuring propagation of any new conversation id back to the client. Seam exposes the conversation id as a property of the built in component `conversation`.

31.1.3.9. Enabling HTTP cache-control headers

Seam does *not* automatically add `cache-control` HTTP headers to any resources served by the Seam resource servlet, or directly from your view directory by the servlet container. This means that your images, Javascript and CSS files, and resource representations from Seam resource servlet such as Seam Remoting Javascript interfaces are usually not cached by the browser. This is convenient in development but should be changed in production when optimizing the application.

You can configure a Seam filter to enable automatic addition of `cache-control` headers depending on the requested URI in `components.xml`:

```
<web:cache-control-filter name="commonTypesCacheControlFilter"
                          regex-url-pattern=".*(\.gif|\.png|\.jpg|\.jpeg|\.css|\.js)"
                          value="max-age=86400"/> <!-- 1 day -->

<web:cache-control-filter name="anotherCacheControlFilter"
                          url-pattern="/my/cachable/resources/*"
                          value="max-age=432000"/> <!-- 5 days -->
```

You do not have to name the filters unless you have more than one filter enabled.

31.1.3.10. Adding custom filters

Seam can install your filters for you, allowing you to specify *where* in the chain your filter is placed (the servlet specification doesn't provide a well defined order if you specify your filters in a `web.xml`). Just add the `@Filter` annotation to your Seam component (which 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 means that the component is available during Seam startup; bijection isn't available here (`@BypassInterceptors`); and the filter should be further down the chain than the RichFaces filter (`@Filter(within="org.jboss.seam.web.ajax4jsfFilter")`).

31.1.4. Integrating Seam with your EJB container

In a Seam application, EJB components have a certain duality, as they are managed by both the EJB container and Seam. Actually, it's more that Seam resolves EJB component references, manages the lifetime of stateful session bean components, and also participates in each method call via interceptors. Let's start with the configuration of the Seam interceptor chain.

We need to apply the `SeamInterceptor` to our Seam EJB components. This interceptor delegates to a set of built-in server-side interceptors that handle such concerns as 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>
```

Seam needs to know where to go to find session beans in JNDI. One way to do this is specify the `@JndiName` annotation on every session bean Seam component. However, this is quite tedious. A better approach is to specify a pattern that Seam can use to calculate the JNDI name from the EJB name. Fortunately, new portable JNDI Syntax was introduced in Java EE 6. There are three JNDI namespaces for portable JNDI lookups - `java:global`, `java:module` and `java:app`. More in *Java*

EE 6 tutorial [http://docs.oracle.com/javaee/6/tutorial/doc/gipjf.html#girgn] We usually specify this option in `components.xml`.

For JBoss AS 7, the following pattern is correct:

```
<core:init jndi-name="java:app/<ejb-module-name>/#{ejbName}" />
```

In this case, `<ejb-module-name>` is the name of the EJB module (by default it is filename of ejb jar) in which the bean is deployed, Seam replaces `#{ejbName}` with the name of the EJB.

How these JNDI names are resolved and somehow locate an EJB component might appear a bit like black magic at this point, so let's dig into the details. First, let's talk about how the EJB components get into JNDI.

EJB components would get assigned a global JNDI name automatically, using the pattern described in *Java EE 6 tutorial* [http://docs.oracle.com/javaee/6/tutorial/doc/gipjf.html#girgn]. The EJB name is the first non-empty value from the following list:

- The value of the `<ejb-name>` element in ejb-jar.xml
- The value of the `name` attribute in the @Stateless or @Stateful annotation
- The simple name of the bean class

Let's look at an example. Assume that you have the following EJB bean and interface defined.

```
package com.example.myapp;

import javax.ejb.Local;

@Local
public interface Authenticator
{
   boolean authenticate();
}

package com.example.myapp;

import javax.ejb.Stateless;

@Stateless
@Name("authenticator")
public class AuthenticatorBean implements Authenticator
{
```

```
    public boolean authenticate() { ... }
}
```

Assuming your EJB bean class is deployed in an EAR named myapp, the global JNDI name myapp/AuthenticatorBean/local will be assigned to it on JBoss AS. As you learned, you can reference this EJB component as a Seam component with the name `authenticator` and Seam will take care of finding it in JNDI according to the JNDI pattern (or `@JndiName` annotation).

So what about the rest of the application servers? Well, according to the Java EE spec, which most vendors try to adhere to religiously, you have to declare an EJB reference for your EJB in order for it to be assigned a JNDI name. That requires some XML. It also means that it is up to you to establish a JNDI naming convention so that you can leverage the Seam JNDI pattern. You might find the JBoss convention a good one to follow.

There are two places you have to define the EJB reference when using Seam on non-JBoss application servers. If you are going to be looking up the Seam EJB component through 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. Here is the EJB reference for the example component just shown:

```
<ejb-local-ref>
   <ejb-ref-name>myapp/AuthenticatorBean</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <local>org.example.vehicles.action.Authenticator</local>
</ejb-local-ref>
```

This reference will cover most uses of the component in a Seam application. However, if you want to be able to inject a Seam EJB component into another Seam EJB component using `@In`, you need to define this EJB reference in another location. This time, it must be defined in ejb-jar.xml, and it's a bit tricker.

Within the context of an EJB method call, you have to deal with a somewhat sheltered JNDI context. When Seam attempts to find another Seam EJB component to satisfy an injection point defined using `@In`, whether or not it finds it depends on whether an EJB reference exists in JNDI. Strictly speaking, you cannot simply resolve JNDI names as you please. You have to define the references explicitly. Fortunately, JBoss recognized how aggravating this would be for the developer and all versions of JBoss automatically register EJBs so they are always available in JNDI, both to the web container and the EJB container. So if you are using JBoss, you can skip the next few paragraphs. However, if you are deploying to GlassFish, pay close attention.

For application servers that stubbornly adhere to the EJB specification, EJB references must always be defined explicitly. But unlike with the web context, where a single resource reference covers all uses of the EJB from the web environment, you cannot declare EJB references globally in the EJB container. Instead, you have to specify the JNDI resources for a given EJB component one-by-one.

Let's assume that we have an EJB named RegisterAction (the name is resolved using the three steps mentioned previously). That EJB has the following Seam injection:

```
@In(create = true)
Authenticator authenticator;
```

In order for this injection to work, the link must be established in the ejb-jar.xml file as follows:

```
<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>
```

Notice that the contents of the `<ejb-local-ref>` are identical to what we defined in web.xml. What we are doing is bringing the reference into the EJB context where it can be used by the RegisterAction bean. You will need to add one of these references for any injection of a Seam EJB component into another Seam EJB component using `@In`. (You can see an example of this setup in the jee5/booking example).

But what about `@EJB`? It's true that you can inject one EJB into another using `@EJB`. However, by doing so, you are injecting the actual EJB reference rather than the Seam EJB component instance. In this case, some Seam features will work, while others won't. That's because Seam's interceptor is invoked on any method call to an EJB component. But that only invokes Seam's server-side interceptor chain. What you lose is Seam's state management and Seam's client-side interceptor chain. Client-side interceptors handle concerns such as security and concurrency. Also, when injecting a SFSB, there is no guarantee that you will get the SFSB bound to the active session or conversation, whatever the case may be. Thus, you definitely want to inject the Seam EJB component using `@In`.

Finally, let's talk about transactions. In an EJB environment, we recommend the use of a special built-in component for transaction management, that is fully aware of container transactions, and can correctly process transaction success events registered with the `Events` component. If

you don't add this line to your `components.xml` file, Seam won't know when container-managed transactions end:

```
<transaction:ejb-transaction/>
```

31.1.5. Don't forget!

There is one final item you need to know about. You must place a `seam.properties`, `META-INF/seam.properties` or `META-INF/components.xml` file in any archive in which your Seam components are deployed (even an empty properties file will do). At startup, Seam will scan any archives with `seam.properties` files for seam components.

In a web archive (WAR) file, you must place a `seam.properties` file in the `WEB-INF/classes` directory if you have any Seam components included here.

That's why all the Seam examples have an empty `seam.properties` file. You can't just delete this file and expect everything to still work!

You might think this is silly and what kind of idiot framework designers would make an empty file affect the behavior of their software?? Well, this is a workaround for a limitation of the JVM — if we didn't use this mechanism, our next best option would be to force you to list every component explicitly in `components.xml`, just like some other competing frameworks do! I think you'll like our way better.

31.2. Using Alternate JPA Providers

Seam comes packaged and configured with Hibernate as the default JPA provider. If you require using a different JPA provider you must tell `seam` about it.

This is a workaround

Configuration of the JPA provider will be easier in the future and will not require configuration changes, unless you are adding a custom persistence provider implementation.

Telling seam about a different JPA provider can be done in one of two ways:

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">
```

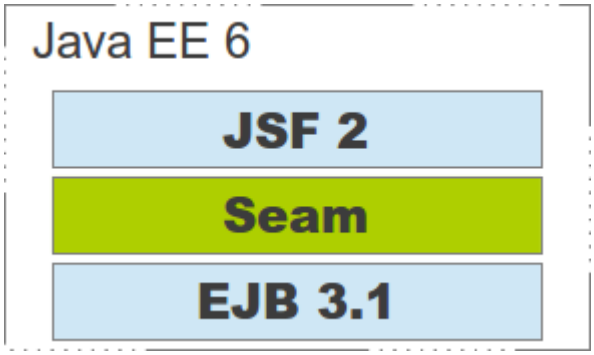
```
</component>
```

If you want to take advantage of your JPA provider's non-standard features you will need to write you own implementation of the `PersistenceProvider`. Use `HibernatePersistenceProvider` as a starting point (don't forget to give back to the community :). Then you will need to tell `seam` to use it as before.

```
<component name="org.jboss.seam.persistence.persistenceProvider"
        class="org.your.package.YourPersistenceProvider">
</component>
```

All that is left is updating the `persistence.xml` file with the correct provider class, and what ever properties your provider needs. Don't forget to package your new provider's jar files in the application if they are needed.

31.3. Configuring Seam in Java EE 6



If you're running in a Java EE environment, this is all the configuration required to start using Seam!

31.3.1. Packaging

Once you've packaged all this stuff together into an EAR, the archive structure will look something like this:

```
my-application.ear/
   jboss-seam.jar
   lib/
      jboss-el.jar
   META-INF/
      MANIFEST.MF
      application.xml
      jboss-deployment-structure.xml
   my-application.war/
```

```
    META-INF/
      MANIFEST.MF
    WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
        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
          ...
```

You should declare `jboss-seam.jar` as an ejb module in `META-INF/application.xml`; `jboss-el.jar` should be placed in the EAR's lib directory (putting it in the EAR classpath.

If you want to use jBPM or Drools, you must include the needed jars in the EAR's lib directory.

If you want to use the Seam tag library (most Seam applications do), you must include `jboss-seam-ui.jar` in the `WEB-INF/lib` directory of the WAR. If you want to use the PDF or email tag libraries, you need to put `jboss-seam-pdf.jar` or `jboss-seam-mail.jar` in `WEB-INF/lib`.

If you want to use the Seam debug page (only works for applications using facelets), you must include `jboss-seam-debug.jar` in the `WEB-INF/lib` directory of the WAR.

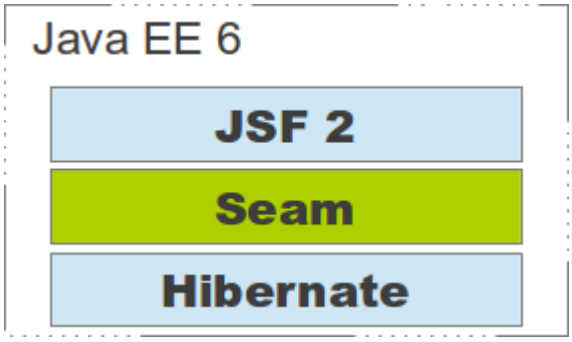
Seam ships with several example applications that are deployable in any Java EE container that supports EJB 3.1.

`faces-config.xml` is not required in JSF 2, but if you want to set up something non-default you need to place it in `WAR/WEB-INF`.

I really wish that was all there was to say on the topic of configuration but unfortunately we're only about a third of the way there. If you're too overwhelmed by all this tedious configuration stuff, feel free to skip over the rest of this section and come back to it later.

31.4. Configuring Seam without EJB

Seam is useful even if you're not yet ready to take the plunge into EJB 3.1. In this case you would use Hibernate 4 instead of EJB 3.1 persistence, and plain JavaBeans instead of session beans. You'll miss out on some of the nice features of session beans but it will be very easy to migrate to EJB 3.1 when you're ready and, in the meantime, you'll be able to take advantage of Seam's unique declarative state management architecture.



Seam JavaBean components do not provide declarative transaction demarcation like session beans do. You *could* manage your transactions manually using the JTA `UserTransaction` or declaratively using Seam's `@Transactional` annotation. But most applications will just use Seam managed transactions when using Hibernate with JavaBeans.

The Seam distribution includes a version of the booking example application that uses Hibernate and JavaBeans instead of EJB, and another version that uses JPA and JavaBeans. These example applications are ready to deploy into any Java EE application server.

31.4.1. Boostrapping Hibernate in Seam

Seam will bootstrap a Hibernate `SessionFactory` from your `hibernate.cfg.xml` file if you install a built-in component:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

You will also need to configure a *managed session* if you want a Seam managed Hibernate `Session` to be available via injection.

```
<persistence:managed-hibernate-session name="hibernateSession"
                        session-factory="#{hibernateSessionFactory}"/>
```

31.4.2. Boostrapping JPA in Seam

Seam will bootstrap a JPA `EntityManagerFactory` from your `persistence.xml` file if you install this built-in component:

```
<persistence:entity-manager-factory name="entityManagerFactory"/>
```

You will also need to configure a *managed persistence context* if you want a Seam managed JPA `EntityManager` to be available via injection.

```
<persistence:managed-persistence-context name="entityManager"
                    entity-manager-factory="#{entityManagerFactory}"/>
```

31.4.3. Packaging

We can package our application as a WAR, in the following structure:

```
my-application.war/
  META-INF/
    MANIFEST.MF
    jboss-deployment-structure.xml
  WEB-INF/
    web.xml
    components.xml
    faces-config.xml
    lib/
      jboss-seam.jar
      jboss-seam-ui.jar
      jboss-el.jar
      hibernate-core.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
  ...
```

If we want to deploy Hibernate in a non-EE environment like Tomcat or TestNG, we need to do a little bit more work.

31.5. Configuring Seam in Java SE

It is possible to use Seam completely outside of an EE environment. In this case, you need to tell Seam how to manage transactions, since there will be no JTA available. If you're using JPA, you can tell Seam to use JPA resource-local transactions, ie. `EntityTransaction`, like so:

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

If you're using Hibernate, you can tell Seam to use the Hibernate transaction API like this:

```
<transaction:hibernate-transaction session="#{session}"/>
```

Of course, you'll also need to define a datasource.

31.6. Configuring jBPM in Seam

Seam's jBPM integration is not installed by default, so you'll need to enable jBPM by installing a built-in component. You'll also need to 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>
```

No further special configuration is needed if you only have pageflows. If you do have business process definitions, you need to 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>
```

The most important thing to notice here is that jBPM transaction control is disabled. Seam or EJB3 should control the JTA transactions.

31.6.1. Packaging

There is not yet any well-defined packaging format for jBPM configuration and process/pageflow definition files. In the Seam examples we've decided to simply package all these files into the root of the EAR. In future, we will probably design some other standard packaging format. So the EAR looks something like this:

```
my-application.ear/
  jboss-seam.jar
  lib/
    jboss-el.jar
    jbpm-jpdl.jar
```

```
META-INF/
   MANIFEST.MF
   application.xml
   jboss-deployment-structure.xml
my-application.war/
   META-INF/
      MANIFEST.MF
   WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
         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
```

31.7. Deployment in JBoss AS 7

JBoss AS 7 is default deployment target for all examples in Seam 2.3 distribution.

Seam 2.3 requires to have setup special deployment metada file `jboss-deployment-structure.xml` for correct initialization. Minimal content for EAR is:

Example 31.1. jboss-deployment-structure.xml

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
 <deployment>
    <dependencies>
       <module name="org.dom4j" export="true"/>
       <module name="org.apache.commons.collections" export="true"/>
       <module name="javax.faces.api" export="true"/>
     </dependencies>
 </deployment>
</jboss-deployment-structure>
```

More details about new AS 7 classloading can be found at *https://docs.jboss.org/author/display/AS7/Developer+Guide#DeveloperGuide-ClassloadinginJBossAS7*

Deployment of multiple modules in one EAR

There is a significant enhancement for speed up the application deployment in AS 7. This unfortunatelly can cause some issues while you have multiple war/ejb modules in your application.

This situation requires to use and set up new Java EE 6 configuration parameter - *Module initialization order* - in `application.xml` - `initialize-in-order` to true. This causes that initialization will happen in defined order like it is in `application.xml`. Example of `application.xml`:

```
<?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"
    version="6" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/application_6.xsd">
  <application-name>test-app</application-name>
  <initialize-in-order>true</initialize-in-order>
  <module>
    <ejb>jboss-seam.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>test-web1.war</web-uri>
      <context-root>test</context-root>
    </web>
    <web>
```

```
    <web-uri>test-web2.war</web-uri>
    <context-root>test</context-root>
  </web>
 </module>
</application>
```

If you are using `maven-ear-plugin` for generation of your application, you can use this plugin configuration:

```
<plugin>
  <artifactId>maven-ear-plugin</artifactId>
  <!-- from version 2.6 the plugin supports Java EE 6 descriptor -->
  <version>2.7</version>
  <configuration>
    <version>6</version>
    <generateApplicationXml>true</generateApplicationXml>
    <defaultLibBundleDir>lib</defaultLibBundleDir>
    <initializeInOrder>true</initializeInOrder>
    <modules>
      <jarModule>
        <groupId>org.jboss.el</groupId>
        <artifactId>jboss-el</artifactId>
        <includeInApplicationXml>false</includeInApplicationXml>
        <bundleDir>lib</bundleDir>
      </jarModule>
      <ejbModule>
        <groupId>org.jboss.seam</groupId>
        <artifactId>jboss-seam</artifactId>
        <bundleFileName>jboss-seam.jar</bundleFileName>
      </ejbModule>
      <ejbModule>
        <groupId>some.user.module</groupId>
        <artifactId>hello-ejbs</artifactId>
        <bundleFileName>hello-ejbs.jar</bundleFileName>
      </ejbModule>
      <webModule>
        <groupId>some.user.module</groupId>
        <artifactId>hello-web1</artifactId>
        <contextRoot>/hello1</contextRoot>
        <bundleFileName>hello-web1.war</bundleFileName>
      </webModule>
      <webModule>
        <groupId>some.user.module</groupId>
```

```
                <artifactId>hello-web2</artifactId>
                <contextRoot>/hello2</contextRoot>
                <bundleFileName>hello-web2.war</bundleFileName>
            </webModule>
        </modules>
    </configuration>
</plugin>
```

31.8. Configuring SFSB and Session Timeouts in JBoss AS 7

It is very important that the timeout for Stateful Session Beans is set higher than the timeout for HTTP Sessions, otherwise SFSB's may time out before the user's HTTP session has ended. JBoss Application Server has a default session bean timeout of 30 minutes, which is configured in `standalone/configuration/standalone.xml` (replace *standalone.xml* with your standalone-full.xml if you use full profile).

The default SFSB timeout can be adjusted by modifying the value of `default-access-timeout` in the EJB subsystem `subsystem xmlns="urn:jboss:domain:ejb3:1.2"`:

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.2">
        <session-bean>
            <stateless>
                <bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>
            </stateless>
            <stateful default-access-timeout="5000" cache-ref="simple"/>
            <singleton default-access-timeout="5000"/>
        </session-bean>
        ...
</subsystem>
```

The default HTTP session timeout can't be modified in JBoss AS 7.

To override default value for your own application, simply include `session-timeout` entry in your application's own `web.xml`:

```
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
```

31.9. Running Seam in a Portlet

If you want to run your Seam application in a portlet, take a look at the JBoss Portlet Bridge, an implementation of JSR-301 that supports JSF within a portlet, with extensions for Seam and RichFaces. See *http://labs.jboss.com/portletbridge* for more.

31.10. Deploying custom resources

Seam scans all jars containing `/seam.properties`, `/META-INF/components.xml` or `/META-INF/seam.properties` on startup for resources. For example, all classes annotated with `@Name` are registered with Seam as Seam components.

You may also want Seam to handle custom resources. A common use case is to handle a specific annotation and Seam provides specific support for this. First, tell Seam which annotations to handle in `/META-INF/seam-deployment.properties`:

```
# A colon-separated list of annotation types to handle
org.jboss.seam.deployment.annotationTypes=com.acme.Foo:com.acme.Bar
```

Then, during application startup you can get hold of all classes annotated with `@Foo`:

```
@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 handle *any* resource. For example, you process any files with the extension `.foo.xml`. To do this, we need to 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;
   }
}
```

Here we are just building a list of any files with the suffix `.foo.xml`.

Then, we need to register the deployment handler with Seam in `/META-INF/seam-deployment.properties`. You can register multiple deployment handler using a comma separated list.

```
# For standard deployment
org.jboss.seam.deployment.deploymentHandlers=com.acme.FooDeploymentHandler
# For hot deployment
org.jboss.seam.deployment.hotDeploymentHandlers=com.acme.FooDeploymentHandler
```

Seam uses deployment handlers internally to install components and namespaces. You can easily access the deployment handler during an `APPLICATION` scoped component's startup:

```
@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) {
     // ...
  }
}
```

Seam annotations

When you write a Seam application, you'll use a lot of annotations. Seam lets you use annotations to achieve a declarative style of programming. Most of the annotations you'll use are defined by the EJB 3.0 specification. The annotations for data validation are defined by the Bean Validation standard. Finally, Seam defines its own set of annotations, which we'll describe in this chapter.

All of these annotations are defined in the package `org.jboss.seam.annotations`.

32.1. Annotations for component definition

The first group of annotations lets you 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, STATELESS`.

When no scope is explicitly specified, the default depends upon 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 contexts variables. The `@Name`/`@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 specification of multiple additional roles.

`@BypassInterceptors`

```
@BypassInterceptors
```

Disables Seam all interceptors on a particular component or method of a component.

`@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 just the parent conversation in which it was instantiated. The component instance will not be visible to nested child conversations, which will get their own instance.

Warning: this is ill-defined, since it implies that a component will be visible for some part of a request cycle, and invisible after that. It is not recommended that applications use this feature!

`@Startup`

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

Specifies that an application scope component is started immediately at initialization time. This is mainly used for certain built-in components that bootstrap critical infrastructure such as JNDI, datasources, etc.

```
@Scope(SESSION) @Startup
```

Specifies that a session scope component is started immediately at session creation time.

- `depends` — specifies that the named components must be started first, if they are installed.

`@Install`

```
@Install(false)
```

Specifies whether or not a component should be installed by default. The lack of an `@Install` annotation indicates a component should 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 the dependency doesn't have a single well-known name.

```
@Install(classDependencies="org.hibernate.Session")
```

Specifies that a component should only be installed if the named class is in 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`.

32.2. 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. Default 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. Default to false.
- `required` — specifies 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 may 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 is used (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 Seam should throw an exception if the component attribute is null during outjection.

Note that it is quite common for these annotations to occur together, for example:

```
@In(create=true) @Out private User currentUser;
```

The next annotation supports the *manager component* pattern; a Seam component manages the lifecycle 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 is the thing that is injected instead of the component instance itself.

The next annotation supports the *factory component* pattern; a Seam component is responsible for initializing the value of a context variable. This is especially useful for initializing any state needed for rendering the response to a non-faces request. It appears on a component method.

`@Factory`

```
@Factory("processInstance") public void createProcessInstance() { ... }
```

Specifies that the method of the component is 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 method returns a value that Seam should use to initialize the value of the named context variable, when 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, default to the JavaBeans property name.
- `scope` — specifies the scope that Seam should bind the returned value to. Only meaningful for factory methods which 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`.

This 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. Default to the name of the component class.

The last 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. Default to the name of the component attribute.

32.3. Annotations for component lifecycle methods

These annotations allow a component to react to its own lifecycle events. They occur on methods of the component. There may be only one of each per component class.

`@Create`

```
@Create
```

Specifies that the method should be called when an instance of the component is instantiated by Seam. Note that 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. Note that 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 one doesn't exist. If an instance does not exist and create is false, the event will not be observed. The default value for create is true.

32.4. Annotations for context demarcation

These annotations provide declarative conversation demarcation. They appear on methods of Seam components, usually action listener methods.

Every web request has a conversation context associated with it. Most of these conversations end at the end of the request. If you want a conversation that span multiple requests, you must "promote" the current 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 simply propagated.

```
@Begin(nested=true)
```

Specifies that if a long-running conversation is already in progress, a new *nested* conversation context begins. The nested conversation will end when the next `@End` is encountered, and the

outer conversation will resume. It is perfectly legal for multiple nested conversations to 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)
```

Specify 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).

- `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. Default 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` — set the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.
- `pageflow` — a process definition 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 effectively destroys the entire conversation stack. If the conversation is not nested, the current conversation is simply ended.

`@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` will be available in a request context variable named `taskInstance`. The jBPM `ProcessInstance` will be available in a request context variable named `processInstance`. (Of course, these objects are available for injection via `@In`.)
- `taskIdParameter` — the name of a request parameter which holds the id of the task. Default to `"taskId"`, which is also the default used by the Seam `taskList` JSF component.
- `flushMode` — set 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` will be available in a request context variable named `taskInstance`. The jBPM `org.jbpm.graph.exe.ProcessInstance` will be available in a request context variable named `processInstance`.
- `taskIdParameter` — the name of a request parameter which holds the id of the task. Default to `"taskId"`, which is also the default used by the Seam `taskList` JSF component.
- `flushMode` — set 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 given 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 a request parameter holding the process id. Default to `"processId"`.

`@Transition`

```
@Transition("cancel")
```

Marks a method as signaling a transition in the current jBPM process instance whenever the method returns a non-null result.

32.5. 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 a similar transactional behavior to the default behavior of a session bean component. ie. 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 may be applied at either class or method level.

Do not use this annotation on EJB 3.0 components, use `@TransactionAttribute`!

`@ApplicationException`

```
@ApplicationException
```

Synonym for javax.ejb.ApplicationException, for use in a pre Java EE 5 environment. Applied to an exception to denote that it is an application exception and should be reported to the client directly(i.e., unwrapped).

Do not use this annotation on EJB 3.0 components, use `@javax.ejb.ApplicationException` instead.

- `rollback` — by default `false`, if `true` this exception should set the transaction to rollback only
- `end` — by default `false`, if `true` this exception should end the current long-running conversation

`@Interceptors`

```
@Interceptors({DVDInterceptor, CDInterceptor})
```

Synonym for javax.interceptors.Interceptors, for use in a pre Java EE 5 environment. Note that this may only be used as a meta-annotation. Declares an ordered list of interceptors for a class or method.

Do not use this annotations on EJB 3.0 components, use `@javax.interceptor.Interceptors` instead.

These annotations are mostly useful for JavaBean Seam components. If you use EJB 3.0 components, you should use the standard Java EE5 annotation.

32.6. Annotations for exceptions

These annotations let you specify how Seam should handle an exception that propagates out of 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, default to the exception message.
- `end` — specifies that the long-running conversation should end, default to `false`.

`@HttpError`

```
@HttpError(errorCode=404)
```

Specifies that the annotated exception causes a HTTP error to be sent.

- `errorCode` — the HTTP error code, default to `500`.
- `message` — a message to be sent with the HTTP error, default to the exception message.
- `end` — specifies that the long-running conversation should end, default to `false`.

32.7. 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 26, Remoting* chapter for more details).

32.8. Annotations for Seam interceptors

The following annotations appear on Seam interceptor classes.

Please refer to the documentation for the EJB 3.0 specification for information about the annotations required for EJB interceptor definition.

`@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 that is called before 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.

32.9. Annotations for asynchronicity

The following annotations are used to declare an asynchronous method, for 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 that a parameter of the asynchronous call is the duration before the call is processed (or first processed for recurring calls).

`@Expiration`

```
@Expiration
```

Specifies that a parameter of the asynchronous call is the datetime at which the call is processed (or first processed for recurring calls).

`@IntervalDuration`

```
@IntervalDuration
```

Specifies that an asynchronous method call recurs, and that the annotation parameter is duration between recurrences.

32.10. Annotations for use with JSF

The following annotations make working with JSF easier.

`@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, register 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.

32.10.1. 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, in addition to the DataModel Selection being injected, the associated DataModel will also be injected. 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. Not needed if there is exactly one `@DataModel` in the component.

32.11. 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.

32.12. 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.org/schema/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.org/schema/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`.

32.13. Annotations for integrating with the servlet container

These annotations allow you to integrate your Seam components with the servlet container.

`@Filter`

Use the Seam component (which implements `javax.servlet.Filter`) annotated with `@Filter` as a servlet filter. It will 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.

Built-in Seam components

This chapter describes Seam's built-in components, and their configuration properties. The built-in components will be created even if they are not listed in your `components.xml` file, but if you need to override default properties or specify more than one component of a certain type, `components.xml` is used.

Note that you can replace any of the built in components with your own implementations simply by specifying the name of one of the built in components on your own class using `@Name`.

33.1. Context injection components

The first set of built in components exist purely to support 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, for example `org.jboss.seam.core.contexts.sessionContext['user']`.

`org.jboss.seam.faces.facesContext`
: Manager component for the `FacesContext` context object (not a true Seam context)

All of these components are always installed.

33.2. 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. It is provided to save a developer from having to 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 Locale of the user. For Locale.US, the input pattern is mm/DD/yy. However, to comply with Y2K, the year is changed from two digits to four (e.g., mm/DD/yyyy).

  It's possible to override the input pattern globally using component configuration. Consult the JavaDoc for this class to see examples.

`org.jboss.seam.faces.facesMessages`
Allows faces success messages to propagate across a browser redirect.

- `add(FacesMessage facesMessage)` — add a faces message, which will be displayed during the next render response phase that occurs in the current conversation.
- `add(String messageTemplate)` — add a faces message, rendered from the given message template which may contain EL expressions.
- `add(Severity severity, String messageTemplate)` — add a faces message, rendered from the given message template which may contain EL expressions.
- `addFromResourceBundle(String key)` — add a faces message, rendered from a message template defined in the Seam resource bundle which may contain EL expressions.
- `addFromResourceBundle(Severity severity, String key)` — add a faces message, rendered from a message template defined in the Seam resource bundle which may contain EL expressions.
- `clear()` — clear all messages.

`org.jboss.seam.faces.redirect`
A convenient API for performing redirects with parameters (this is especially 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()` — perform 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 (session-scoped by default) responsible for maintaining a collection of render stamps. A render stamp is an indicator as to whether a form which was rendered has been submitted. This store is useful when the client-side state saving method of JSF is being used because it puts the determination of whether a form has been posted in the control of the server rather than in the component tree which is maintained on the client.

To unbind this check from the session (which is one of the main design goals of client-side state saving) an implementation must be provided that stores the render stamps in the

application (valid as long as the application is running) or the database (valid across server restarts).

- `maxSize` — The maximum number of stamps to be kept in the store. Default: 100

These components are installed when the class `javax.faces.context.FacesContext` is available on the classpath.

33.3. Utility components

These components are merely useful.

`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)` — raise an event of a particular type and distribute to all observers.
- `raiseAsynchronousEvent(String type)` — raise an event to be processed asynchronously by the EJB3 timer service.
- `raiseTimedEvent(String type, ....)` — schedule an event to be processed asynchronously by the EJB3 timer service.
- `addListener(String type, String methodBinding)` — add 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)` — scan the template for JSF EL expressions of the form `#{...}` and replace them with their evaluated values.

`org.jboss.seam.core.expressions`
An API for creating value and method bindings.

- `createValueBinding(String expression)` — create a value binding object.
- `createMethodBinding(String expression)` — create 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. Default to `treecache.xml`.

All of these components are always installed.

33.4. Components for internationalization and themes

The next group of 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()` — select the specified locale.
- `localeSelector.locale` — the actual `java.util.Locale`.
- `localeSelector.localeString` — the stringified 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()` — select the specified locale.
- `timezoneSelector.timezone` — the actual `java.util.TimeZone`.
- `timezoneSelector.timeZoneId` — the stringified 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 `SelectItems` 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.

33.5. Components for controlling conversations

The next group of components allow control of conversations by the application or user interface.

`org.jboss.seam.core.conversation`
API for application control of attributes of the current Seam conversation.

- `getId()` — returns the current conversation id
- `isNested()` — is the current conversation a nested conversation?
- `isLongRunning()` — is the current conversation 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 be used 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()` — redirect to the last well-defined view id for this conversation (useful after login challenges).
- `leave()` — exit the scope of this conversation, without actually ending the conversation.
- `begin()` — begin a long-running conversation (equivalent to `@Begin`).
- `beginPageflow(String pageflowName)` — begin a long-running conversation with a pageflow (equivalent to `@Begin(pageflow="...")`).
- `end()` — end a long-running conversation (equivalent to `@End`).
- `pop()` — pop the conversation stack, returning to the parent conversation.
- `root()` — return to the root conversation of the conversation stack.
- `changeFlushMode(FlushModeType flushMode)` — change the flush mode of the conversation.

`org.jboss.seam.core.conversationList`
: Manager component for the conversation list.

`org.jboss.seam.core.conversationStack`
: Manager component for the conversation stack (breadcrumbs).

`org.jboss.seam.faces.switcher`
: The conversation switcher.

All of these components are always installed.

33.6. jBPM-related components

These components are for use with jBPM.

`org.jboss.seam.pageflow.pageflow`
: API control of Seam pageflows.

- `isInProcess()` — returns `true` if there is currently a pageflow in process
- `getProcessInstance()` — returns jBPM `ProcessInstance` for the current pageflow
- `begin(String pageflowName)` — begin a pageflow in the context of the current conversation
- `reposition(String nodeName)` — reposition the current pageflow to a particular node

`org.jboss.seam.bpm.actor`
API for application control of attributes of the jBPM actor associated with the current session.

- `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`
API for application control of the jBPM transition for the current task.

- `setName(String transitionName)` — sets the jBPM transition name to be used when the current task is ended via `@EndTask`.

`org.jboss.seam.bpm.businessProcess`
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()` — is a task instance associated with the current conversation?
- `businessProcess.hasCurrentProcess()` — is a process instance associated with the current conversation.
- `createProcess(String name)` — create an instance of the named process definition and associate it with the current conversation.
- `startTask()` — start the task associated with the current conversation.
- `endTask(String transitionName)` — end the task associated with the current conversation.
- `resumeTask(Long id)` — associate the task with the given id with the current conversation.
- `resumeProcess(Long id)` — associate the process with the given id with the current conversation.
- `transition(String transitionName)` — trigger the transition.

`org.jboss.seam.bpm.taskInstance`
Manager component for the jBPM `TaskInstance`.

`org.jboss.seam.bpm.processInstance`
Manager component for the jBPM `ProcessInstance`.

`org.jboss.seam.bpm.jbpmContext`
Manager component for an event-scoped `JbpmContext`.

`org.jboss.seam.bpm.taskInstanceList`
: Manager component for the jBPM task list.

`org.jboss.seam.bpm.pooledTaskInstanceList`
: Manager component for the jBPM pooled task list.

`org.jboss.seam.bpm.taskInstanceListForType`
: Manager component for the jBPM task lists.

`org.jboss.seam.bpm.pooledTask`
: Action handler for pooled task assignment.

`org.jboss.seam.bpm.processInstanceFinder`
: Manager 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.

33.7. Security-related components

These components relate to web-tier security.

`org.jboss.seam.web.userPrincipal`
: 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. `<h:commandButton value="edit" rendered="#{isUserInRole['admin']}"/>`.

33.8. JMS-related components

These components are for use with managed `TopicPublishers` and `QueueSenders` (see below).

`org.jboss.seam.jms.queueSession`
: Manager component for a JMS `QueueSession` .

`org.jboss.seam.jms.topicSession`
: Manager component for a JMS `TopicSession` .

33.9. Mail-related components

These components are for use with Seam's Email support

`org.jboss.seam.mail.mailSession`
: Manager component for a JavaMail `Session`. The session can be either looked up in the JNDI context (by setting the `sessionJndiName` property) or it can created from the configuration options in which 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` — enable JavaMail debugging (very verbose)
  - `org.jboss.seam.mail.mailSession.ssl` — enable SSL connection to SMTP (will default to port 465)

    `org.jboss.seam.mail.mailSession.tls` — by default true, enable TLS support in the mail session
  - `org.jboss.seam.mail.mailSession.sessionJndiName` — name under which a javax.mail.Session is bound to JNDI. If supplied, all other properties will be ignored.

33.10. Infrastructural components

These components provide critical platform infrastructure. You can install a component which isn't installed by default by setting `install="true"` on the component in `components.xml`.

`org.jboss.seam.core.init`
: 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` — enable Seam debug mode. This should be set to false when in production. You may see errors if the system is placed under any load and debug is enabled.
  - `org.jboss.seam.core.init.clientSideConversations` — if set to `true`, Seam will save conversation context variables in the client instead of in the `HttpSession`.

`org.jboss.seam.core.manager`
: 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` — 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, default to `conversationId`.
- `org.jboss.seam.core.manager.conversationIsLongRunningParameter` — the request parameter used to propagate information about whether the conversation is long-running, default to `conversationIsLongRunning`.
- `org.jboss.seam.core.manager.defaultFlushMode` — set the flush mode set by default on any Seam Managed Persistence Context. By default `AUTO`.

`org.jboss.seam.navigation.pages`
: Internal component for Seam workspace management. Always installed.

- `org.jboss.seam.navigation.pages.noConversationViewId` — global setting for the view id to redirect to when a conversation entry is not found on the server side.
- `org.jboss.seam.navigation.pages.loginViewId` — global setting for the view id to redirect to when an unauthenticated user tries to access a protected view.
- `org.jboss.seam.navigation.pages.httpPort` — global setting for the port to use when the http scheme is requested.
- `org.jboss.seam.navigation.pages.httpsPort` — global setting for the port to use when the https scheme is requested.
- `org.jboss.seam.navigation.pages.resources` — a list of resources to search for `pages.xml` style resources. Defaults to `WEB-INF/pages.xml`.

`org.jboss.seam.bpm.jbpm`
: Bootstraps a `JbpmConfiguration`. Install as class `org.jboss.seam.bpm.Jbpm`.

- `org.jboss.seam.bpm.jbpm.processDefinitions` — a list of resource names of jPDL files to be used for orchestration of business processes.
- `org.jboss.seam.bpm.jbpm.pageflowDefinitions` — a list of resource names of jPDL files to be used for orchestration of conversation page flows.

`org.jboss.seam.core.conversationEntries`
: Internal session-scoped component recording the active long-running conversations between requests.

`org.jboss.seam.faces.facesPage`
: Internal page-scoped component recording the conversation context associated with a page.

`org.jboss.seam.persistence.persistenceContexts`
: Internal component recording the persistence contexts which were used in the current conversation.

`org.jboss.seam.jms.queueConnection`
Manages a JMS `QueueConnection`. Installed whenever managed `QueueSender` is installed.

- `org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName` — the JNDI name of a JMS `QueueConnectionFactory`. Default to `UIL2ConnectionFactory`

`org.jboss.seam.jms.topicConnection`
Manages a JMS `TopicConnection`. Installed whenever managed `TopicPublisher` is installed.

- `org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName` — the JNDI name of a JMS `TopicConnectionFactory`. Default to `UIL2ConnectionFactory`

`org.jboss.seam.persistence.persistenceProvider`
Abstraction layer for non-standardized features of JPA provider.

`org.jboss.seam.core.validators`
Caches instances of Hibernate Validator `ClassValidator`.

`org.jboss.seam.faces.validation`
Allows the application to determine whether validation failed or was successful.

`org.jboss.seam.debug.introspector`
Support for the Seam Debug Page.

`org.jboss.seam.debug.contexts`
Support for the Seam Debug Page.

`org.jboss.seam.exception.exceptions`
Internal component for exception handling.

`org.jboss.seam.transaction.transaction`
API for controlling transactions and abstracting the underlying transaction management implementation behind a JTA-compatible interface.

`org.jboss.seam.faces.safeActions`
Decides if an action expression in an incoming URL is safe. This is done by checking that the action expression exists in the view.

33.11. Miscellaneous components

These components don't fit into

`org.jboss.seam.async.dispatcher`
Dispatcher stateless session bean for asynchronous methods.

`org.jboss.seam.core.image`
Image manipulation and interrogation.

`org.jboss.seam.core.pojoCache`
: Manager component for a PojoCache instance.

`org.jboss.seam.core.uiComponent`
: Manages a map of UIComponents keyed by component id.

33.12. Special components

Certain special Seam component classes are installable 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`
: 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, default to java:/<*managedPersistenceContext*>.

<*entityManagerFactory*>, `org.jboss.seam.persistence.EntityManagerFactory`
: Manages a JPA `EntityManagerFactory`. This is most useful when using JPA outside of an EJB 3.0 supporting environment.

  - `entityManagerFactory.persistenceUnitName` — the name of the persistence unit.

  See the API JavaDoc for further configuration properties.

<*session*>, `org.jboss.seam.persistence.ManagedSession`
: 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, default to java:/<*managedSession*>.

<*sessionFactory*>, `org.jboss.seam.persistence.HibernateSessionFactory`
Manages a Hibernate `SessionFactory`.

- `<sessionFactory>.cfgResourceName` — the path to the configuration file. Default to `hibernate.cfg.xml`.

See the API JavaDoc for further configuration properties.

<*managedQueueSender*>, `org.jboss.seam.jms.ManagedQueueSender`
Manager component for an event scoped managed JMS `QueueSender`.

- <*managedQueueSender*>.queueJndiName — the JNDI name of the JMS queue.

<*managedTopicPublisher*>, `org.jboss.seam.jms.ManagedTopicPublisher`
Manager component for an event scoped managed JMS `TopicPublisher`.

- <*managedTopicPublisher*>.topicJndiName — the JNDI name of the JMS topic.

<*managedWorkingMemory*>, `org.jboss.seam.drools.ManagedWorkingMemory`
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`
Manager component for an application scoped Drools `RuleBase`. *Note that this is not really intended for production usage, since it does not support dynamic installation of new rules.*

- <*ruleBase*>.ruleFiles — a list of files containing Drools rules.

  <*ruleBase*>.dslFile — a Drools DSL definition.

<*entityHome*>, `org.jboss.seam.framework.EntityHome`

<*hibernateEntityHome*>, `org.jboss.seam.framework.HibernateEntityHome`

<*entityQuery*>, `org.jboss.seam.framework.EntityQuery`

<*hibernateEntityQuery*>, `org.jboss.seam.framework.HibernateEntityQuery`

Seam JSF controls

Seam includes a number of JSF controls that are useful for working with Seam. These are intended to complement the built-in JSF controls, and controls from other third-party libraries. We recommend JBoss RichFaces, ICEsoft ICEfaces and Apache MyFaces Trinidad tag libraries for use with Seam. We do not recommend the use of the Tomahawk tag library.

34.1. 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.org/schema/seam/taglib">
```

The ui example demonstrates the use of a number of these tags.

34.1.1. Navigation Controls

34.1.1.1. `<s:button>`

Description

A button that supports invocation of an action with control over conversation propagation. *Does not submit the form.*

Attributes

- `value` — the label.
- `action` — a method binding that specified the action listener.
- `view` — the JSF view id to link to.
- `fragment` — the fragment identifier to link to.
- `disabled` — is the link disabled?
- `propagation` — determines the conversation propagation style: `begin`, `join`, `nested`, `none`, `end` or `endRoot`.
- `pageflow` — a pageflow definition to begin. (This is only useful when `propagation="begin"` or `propagation="join"` is used).
- `includePageParams` — when set to false, page parameters defined in `pages.xml` will be excluded from rendering.

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 />`.

34.1.1.2. `<s:conversationId>`

Description

Add the conversation id to JSF link or button (e.g. `<h:commandLink />` , `<s:button />`).

Attributes

None

34.1.1.3. `<s:taskId>`

Description

Add the task id to an output link (or similar JSF control), when the task is available via `#{task}`.

Attributes

None.

34.1.1.4. `<s:link>`

Description

A link that supports invocation of an action with control over conversation propagation. *Does not submit the form.*

The use of action listeners (including the default JSF action listener) is not supported with `<s:link />`.

Attributes

- `value` — the label.
- `action` — a method binding that specified the action listener.
- `view` — the JSF view id to link to.

- `fragment` — the fragment identifier to link to.
- `disabled` — is the link disabled?
- `propagation` — determines the conversation propagation style: `begin`, `join`, `nested`, `none`, `end` or `endRoot`.
- `pageflow` — a pageflow definition to begin. (This is only useful when using `propagation="begin"` or `propagation="join"`.)
- `includePageParams` — when set to false, page parameters defined in `pages.xml` will be excluded from rendering.

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 occured.

34.1.1.5. `<s:conversationPropagation>`

Description

Customize the conversation propagation for a command link or button (or similar JSF control). *Facelets only.*

Attributes

- `type` — determines the conversation propagation style: `begin`, `join`, `nested`, `none`, `end` or `endRoot`.
- `pageflow` — a pageflow definition to begin. (This is only useful when using `propagation="begin"` or `propagation="join"`.)

Usage

```
<h:commandButton value="Apply" action="#{personHome.update}">
  <s:conversationPropagation type="join" />
</h:commandButton>
```

34.1.1.6. `<s:defaultAction>`

Description

Specify the default action to run when the form is submitted using the enter key.

Currently you can only nest it inside buttons (e.g. `<h:commandButton />`, `<a:commandButton />` or `<tr:commandButton />`).

You must specify an id on the action source. You can only have one default action per form.

Attributes

None.

Usage

```
<h:commandButton id="foo" value="Foo" action="#{manager.foo}">
  <s:defaultAction />
</h:commandButton>
```

34.1.2. Converters and Validators

34.1.2.1. `<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>
```

34.1.2.2. `<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. The converter should be able to find the items declared in the JSF controls on form submission, otherwise you will receive a validation error.

Attributes

None.

Configuration

You must use *Seam managed transactions* (see *Section 10.2, "Seam managed transactions"*) with `<s:convertEntity />`.

If your *Managed Persistence Context* isn't called `entityManager`, then you need to set it in components.xml:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:ui="http://jboss.org/schema/seam/ui">

  <ui:jpa-entity-loader entity-manager="#{em}" />
```

If you are using a *Managed Hibernate Session* then you need to set it in components.xml:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:ui="http://jboss.org/schema/seam/ui">

  <ui:hibernate-entity-loader />
```

If your *Managed Hibernate Session* isn't called `session`, then you need to set it in components.xml:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:ui="http://jboss.org/schema/seam/ui">

  <ui:hibernate-entity-loader session="#{hibernateSession}" />
```

If you want to use more than one entity manager with the entity converter, you can create a copy of the entity converter for each entity manager in `components.xml` - note how the entity converter delegates to the entity loader to perform persistence operations:

```
<components xmlns="http://jboss.org/schema/seam/components"
            xmlns:ui="http://jboss.org/schema/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>
```

34.1.2.3. `<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>
```

34.1.2.4. `<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>
```

34.1.2.5. `<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>
```

34.1.2.6. `<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>
```

34.1.2.7. `<s:validateEquality>`

Description

Tag to nest inside an input control to validate that its 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` — What operator to use when comparing the 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>
```

34.1.2.8. `<s:validate>`

Description

A non-visual control, validates a JSF input field against the bound property using Hibernate Validator.

Attributes

None.

Usage

```
<h:inputText id="userName" required="true"
             value="#{customer.userName}">
  <s:validate />
</h:inputText>
<h:message for="userName" styleClass="error" />
```

34.1.2.9. `<s:validateAll>`

Description

A non-visual control, validates all child JSF input fields against their bound properties using 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>
```

34.1.3. Formatting

34.1.3.1. `<s:decorate>`

Description

"Decorate" a JSF input field when validation fails or when `required="true"` is set.

Attributes

- `template` — the facelets template to use 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 to enclose the template used to decorate 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 you have set the input component being decorated as required, and `#{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.org/schema/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>
```

34.1.3.2. `<s:div>`

Description

Render a HTML `<div>`.

Attributes

None.

Usage

```
<s:div rendered="#{selectedMember == null}">
  Sorry, but this member does not exist.
</s:div>
```

34.1.3.3. `<s:span>`

Description

Render a HTML `<span>`.

Attributes

- `title` — Title for a span.

Usage

```
<s:span styleClass="required" rendered="#{required}" title="Small tooltip">*</s:span>
```

34.1.3.4. `<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>
```

34.1.3.5. `<s:label>`

Description

"Decorate" 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"/>
```

34.1.3.6. `<s:message>`

Description

"Decorate" 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>
```

34.1.4. Seam Text

34.1.4.1. `<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.

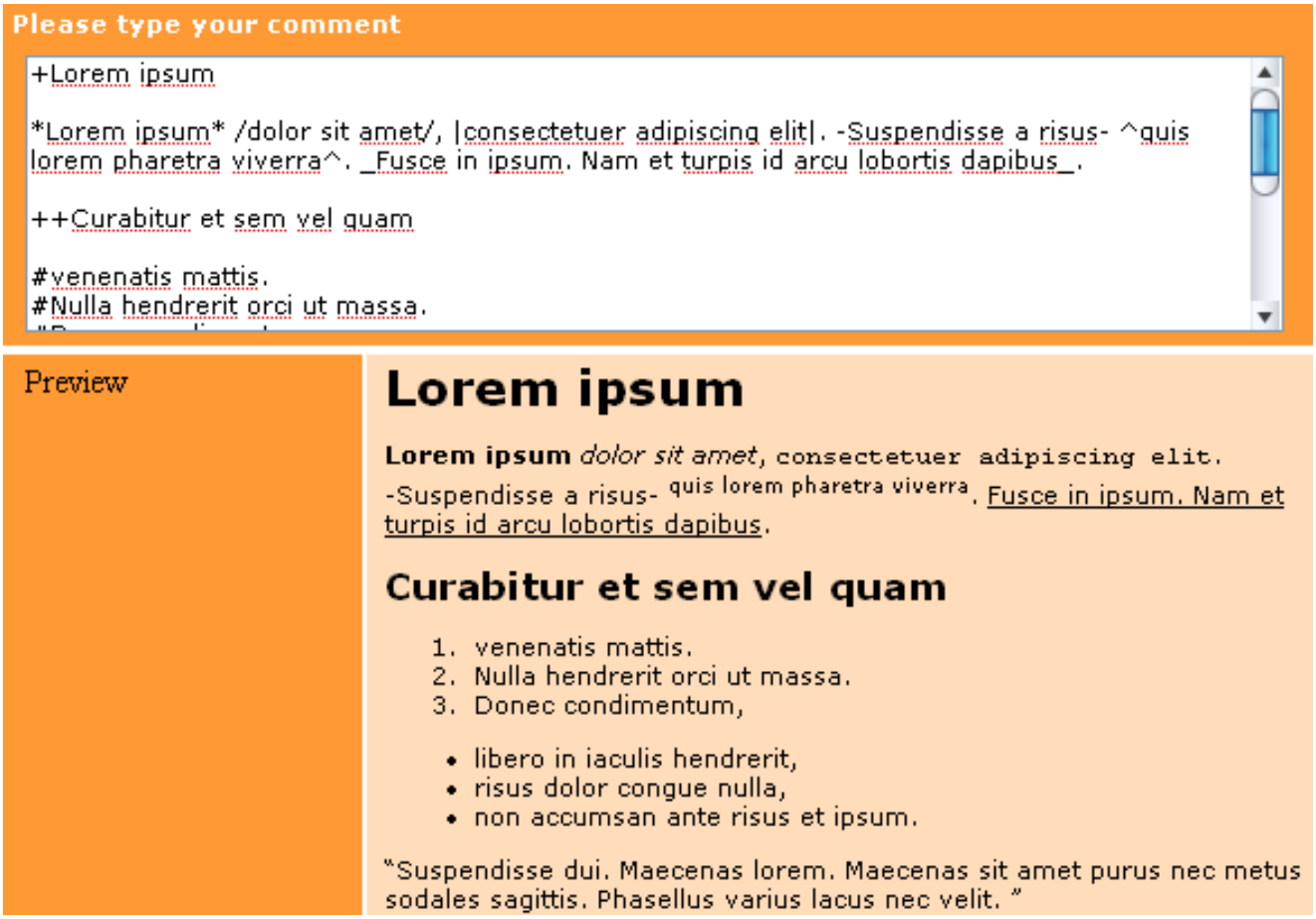
Attributes

- `value` — an EL expression specifying the rich text markup to render.

Usage

```
<s:formattedText value="#{blog.text}"/>
```

Example



34.1.5. Form support

34.1.5.1. `<s:token>`

Description

Produces a random token that is inserted into a hidden form field to help to secure JSF form posts against cross-site request forgery (XSRF) attacks. Note that 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, hence binding the token to the session. This value can be set to false if the "build before restore" mode of Facelets is activated (the default in JSF 2.0). (default: false)
- `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. (default: false)
- `allowMultiplePosts` — indicates whether to allow the same form to be submitted multiple times with the same signature (as long as the view does not change). This is a common need if the form is perform Ajax calls but not rerendering itself or, at the very least, the UIToken component. The preferred approach is to have the UIToken component rerendered on any Ajax call where the UIToken component would be processed. (default: false)

Usage

```
<h:form>
  <s:token enableCookieNotice="true" requireSession="false"/>
  ...
</h:form>
```

34.1.5.2. `<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}">
  <!-- JSF 2 way <f:converter converterId="org.jboss.seam.ui.EnumConverter" />-->
  <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>
```

34.1.5.3. `<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 `List<SelectItem>`
- `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` — Value to return to the server if this option is selected. Optional, by default the `var` object is used. Can reference the `var` variable.
- `disabled` — if 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>
```

34.1.5.4. `<s:fileUpload>`

Description

Renders a file upload control. This control must be used within a form with an encoding type of `multipart/form-data`, i.e:

```
<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 may be configured in components.xml:

- `createTempFiles` — if this option is set to true, uploaded files are streamed to a temporary file instead of 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` — this value binding receives the binary file data. The receiving field should be declared as a `byte[]` or `InputStream` (required).

- `contentType` — this value binding receives the file's content type (optional).
- `fileName` — this value binding receives the filename (optional).
- `fileSize` — this value binding receives the file size (optional).
- `accept` — a comma-separated list of content types to accept, may not be supported by the browser. E.g. `"images/png,images/jpg"`, `"images/*"`.
- `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}" />
```

34.1.6. Other

34.1.6.1. `<s:cache>`

Description

Cache 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 if the cache should be used.
- `region` — a 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>
```

34.1.6.2. `<s:resource>`

Description

A tag that acts a file download provider. It must be alone in the JSF page. To be able to use this control, web.xml must be set up 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` — Data that should be downloaded. May be a java.util.File, an InputStream or a byte array.
- `fileName` — Filename of the file to be served
- `contentType` — content type of the file to be downloaded
- `disposition` — disposition to use. Default is inline

Usage

Here is an example on how to use the tag:

```
<s:resource xmlns="http://www.w3.org/1999/xhtml"
 xmlns:s="http://jboss.org/schema/seam/taglib"
```

```
data="#{resources.data}"
contentType="#{resources.contentType}"
fileName="#{resources.fileName}" />
```

The bean named `resources` is some backing bean that given some request parameters servers a specific file, see `s:download`.

34.1.6.3. `<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>
```

Will produce something like: `http://localhost/resources.seam?fileId=1`

34.1.6.4. `<s:graphicImage>`

Description

An extended `<h:graphicImage>` that allows the image to be created in a Seam Component; further transforms can be applied to the image.

All attributes for `<h:graphicImage>` are supported, as well as:

Attributes

- `value` — 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/png`, `image/jpeg`, `image/gif` and `image/bmp`.
- `fileName` — if not specified the served image will have a generated file name. If you want to name your file, you should specify it here. This name should be unique
- `cache` — specify if this image should be cached. Boolean value.
- `invalidate` — specify if this image cache should be invalidated. Boolean value.
- `cacheKey` — specify unique string for a cached image.

Transformations

To apply a transform to the image, you would nest a tag specifying the transform to apply. Seam currently supports these transforms:

`<s:transformImageSize>`

- `width` — new width of the image
- `height` — new height of the image
- `maintainRatio` — if `true`, and *one* of `width`/`height` are specified, the image will be resized with the dimension not specified being calculated to maintain the aspect ratio.
- `factor` — scale the image by the given factor

`<s:transformImageBlur>`

- `radius` — perform a convolution blur with the given radius

`<s:transformImageType>`

- `contentType` — alter the type of the image to either `image/jpeg` or `image/png`

It's easy to create your own transform - create a `UIComponent` which 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>
```

34.1.6.5. `<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 26, Remoting* for more details.

Usage

```
<s:remote include="customerAction,accountAction,com.acme.MyBean"/>
```

34.2. Annotations

Seam also provides annotations to allow you to 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. Shown here is a converter which is able to access the JPA EntityManager inside a JTA transaction, when converting the value back to it's 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. Shown here is a validator which injects another Seam component; the injected component is used to validate the value.

JBoss EL

Seam uses JBoss EL which provides an extension to the standard Unified Expression Language (EL). JBoss EL provides a number of enhancements that increase the expressiveness and power of EL expressions.

35.1. Parameterized Expressions

Standard EL 2.1 does not allow you to use a method with user defined parameters — of course, JSF listener methods (e.g. a `valueChangeListener`) take parameters provided by JSF. *Standard EL 2.2* [http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html], which is in Java EE 6, allows it now. So you don't have to use JBoss EL enhancements.

You can still use JBoss EL instead of standard EL 2.2 from Java EE 6 by setting up `com.sun.faces.expressionFactory` in `web.xml`:

```
<context-param>
   <param-name>com.sun.faces.expressionFactory</param-name>
   <param-value>org.jboss.el.ExpressionFactoryImpl</param-value>
</context-param>
```

JBoss EL and EL 2.2 removed 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
  }
}
```

35.1.1. Usage

Just as in calls to method from Java, parameters are surrounded by parentheses, and separated by commas:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}" value="Book Hotel"/>
```

The parameters `hotel` and `user` will be evaluated as value expressions and passed to the `bookHotel()` method of the component.

Any value expression may be used as a parameter:

```
<h:commandButton
  action="#{hotelBooking.bookHotel(hotel.id, user.username)}"
  value="Book Hotel"/>
```

It's important to fully understand how this extension to EL works. When the page is rendered, the parameter *names* are stored (for example, `hotel.id` and `user.username`), and evaluated (as value expressions) when the page is submitted. You can't pass objects as parameters!

You must ensure that the parameters are available not only when the page is rendered, but also when it is submittedIf the arguments can not be resolved when the page is submitted 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, used to bind a field to a backing bean. Value expressions use JavaBean naming conventions and expect a getter/setter pair. Often JSF expects a value expression where only retrieval (get) is needed (e.g. the `rendered` attribute). Many objects, however, don't have appropriately named property accessors or 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')}
```

35.1.2. Limitations and Hints

When using JBoss EL you should keep the following points in mind:

- *Incompatibility with JSP 2.1* — JBoss EL can't currently be used with JSP 2.1 as the compiler rejects expressions with parameters in. So, if you want to use this extension with JSF 1.2, you will need to use Facelets. 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 works great if you are selecting a row using a `<h:commandButton />` or `<h:commandLink />`:

```
@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 (and therefore produce a bookmarkable link) so a "magic" parameter is needed to recreate the item when the action method is called. This magic 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. In the case of a method binding, JSF assumes that there are no parameters to pass. With this extension, we can't know the parameter types until after the expression has been evaluated. 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.

35.2. 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, your only option is to iterate over the list to retrieve the values. JBoss EL allows this with a projection expression:

```
#{company.departments.{d|d.name}}
```

The subexpression 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 it would be perfectly valid to write the following, assuming you had a use for the lengths of all the department names in a company:

```
#{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 looks like it returns a list of all the employees in all the 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}}}
```

It is important to note that this syntax cannot be parsed by Facelets or JSP and thus cannot be used in xhtml or JSP files. We anticipate that the projection syntax will change in future versions of JBoss EL.

Clustering and EJB Passivation

Please note that this chapter is still being reviewed. Tread carefully.

This chapter covers two distinct topics that happen share a common solution in Seam, (web) clustering and EJB passivation. Therefore, they are addressed together in this reference manual. Although performance tends to be grouped in this category as well, it's kept separate because the focus of this chapter is on the programming model and how it's affected by the use of the aforementioned features.

In this chapter you will learn how Seam manages the passivation of Seam components and entity instances, how to activate this feature, and how this feature is related to clustering. You will also learn how to deploy a Seam application into a cluster and verify that HTTP session replication is working properly. Let's start with a little background on clustering and see an example of how you deploy a Seam application to a JBoss AS cluster.

36.1. Clustering

Clustering (more formally web clustering) allows an application to run on two or more parallel servers (i.e., nodes) while providing a uniform view of the application to clients. Load is distributed across the servers in such a way that if one or more of the servers fails, the application is still accessible via any of the surviving nodes. This topology is crucial for building scalable enterprise applications as performance and availability can be improved simply by adding nodes. But it brings up an important question. *What happens to the state that was on the server that failed?*

Since day one, Seam has always provided support for stateful applications running in a cluster. Up to this point, you have learned that Seam provides state management in the form of additional scopes and by governing the life cycle of stateful (scoped) components. But state management in Seam goes beyond creating, storing and destroying instances. Seam tracks changes to JavaBean components and stores the changes at strategic points during the request so that the changes can be restored when the request shifts to a secondary node in the cluster. Fortunately, monitoring and replication of stateful EJB components is already handled by the EJB server, so this feature of Seam is intended to put stateful JavaBeans on par with their EJB cohorts.

But wait, there's more! Seam also offers an incredibly unique feature for clustered applications. In addition to monitoring JavaBean components, Seam ensures that managed entity instances (i.e. JPA and Hibernate entities) don't become detached during replication. Seam keeps a record of the entities that are loaded and automatically loads them on the secondary node. You must, however, be using a Seam-managed persistence context to get this feature. More in depth information about this feature is provided in the second half of this chapter.

Now that you understand what features Seam offers to support a clustered environment, let's look at how you program for clustering.

36.1.1. Programming for clustering

Any session- or conversation-scoped mutable JavaBean component that will 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 dirty flag that is reported and reset by the `clearDirty()` method. Seam calls this method to determine if it is necessary to replicate the component. This avoids having to use the more cumbersome Servlet API to add and remove the session attribute on every change of the object.

You also must ensure that all session- and conversation-scoped JavaBean components are Serializable. Additional, all fields of a stateful component (EJB or JavaBean) must Serializable unless the field is 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 where people often get bitten is by using `List.subList` to create a list. The resulting list is not Serializable. So watch out for situations like that. If hit a `java.io.NotSerializableException` and cannot locate the culprit at first glance, you can put a breakpoint on this exception, run the application server in debug mode and attach a debugger (such as Eclipse) to see what deserialization is choking on.

Note

Please note that clustering does not work with hot deployable components. But then again, you shouldn't be using hot deployable components in a non-development environment anyway.

36.1.2. Deploying a Seam application to a JBoss AS cluster with session replication

Note

The detailed clustering How-To on AS7 is described at *AS7 Cluster Howto* [https://docs.jboss.org/author/display/AS71/AS7+Cluster+Howto]

The procedure outlined in this tutorial has been validated with an seam-gen application and the Seam booking example.

In the tutorial, I assume that the IP addresses of the master and slave servers are 192.168.1.2 and 192.168.1.3, respectively. I am intentionally not using the mod_jk load balancer so that it's easier to validate that both nodes are responding to requests and can share sessions.

Note

JBoss AS clustering relies on UDP multicasting provided by jGroups. The SELinux configuration that ships with RHEL/Fedora blocks these packets by default. You can allow them to pass by modifying 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
```

Detailed information can be found on *this page* [http://www.jboss.org/community/docs/DOC-11935] on the JBoss Wiki.

- Create two instances of JBoss AS (just extract the zip twice)
- Deploy the JDBC driver to `$JBOSS_HOME/standalone/deployments/` on both instances if not using H2
- Add `<distributable/>` as the first child element in `WEB-INF/web.xml`
- Set the `distributable` property on `org.jboss.seam.core.init` to true to enable the ManagedEntityInterceptor (i.e., `<core:init distributable="true"/>`)
- Ensure you have two IP addresses available (two computers, two network cards, or two IP addresses bound to the same interface). I'll assume the two IP address are 192.168.1.2 and 192.168.1.3
- Change only on Node2 `node2/domain/configuration/host.xml`the host name from master to slave.

```
<host name="slave" xmlns="urn:jboss:domain:1.1">
```

- Edit on both nodes `domain/configuration/host.xml` and search `interfaces` element, e.g for node1:

```
  <interfaces>
     <interface name="management">
```

```
        <loopback-address value="192.168.1.2"/>
    </interface>
    <interface name="public">
        <loopback-address value="192.168.1.2"/>
    </interface>
    <interface name="unsecure">
        <loopback-address value="192.168.1.2"/>
    </interface>
</interfaces>
```

respectively for node2:

```
<interfaces>
    <interface name="management">
        <loopback-address value="192.168.1.3"/>
    </interface>
    <interface name="public">
        <loopback-address value="192.168.1.3"/>
    </interface>
    <interface name="unsecure">
        <loopback-address value="192.168.1.3"/>
    </interface>
</interfaces>
```

- Change the `domain-controller` element content on slave *node2* in `domain/configuration/host.xml` to be able to manage Node2 by Web Console UI, like:
  - remove

```
<local/>
```

  - uncomment and change `host` and `port` attributes in `remote` element

  The result should be as the following:

```
<domain-controller>
  <remote host="192.168.1.2" port="9999" security-realm="ManagementRealm"/> -->
</domain-controller>
```

- Start the master JBoss AS 7 instance on the first IP

```
cd node1/jboss-as-7.1.1.Final; ./bin/domain.sh
```

The log should report that there are 1 cluster members and 0 other members.

- Start the slave JBoss AS instance on the second IP

```
cd node2/jboss-as-7.1.1.Final; ./bin/domain.sh
```

The log should report that there are 2 cluster members and 1 other members. It should also show the state being retrieved from the master.

- Deploy the `*-ds.xml` by using admin console at *http://localhost:9990*

  In the log of the master you should see acknowledgement of the deployment. In the log of the slave you should see a corresponding message acknowledging the deployment to the slave.

- Deploy the application by using admin console at *http://localhost:9990*

  In the log of the master you should see acknowledgement of the deployment. In the log of the slave you should see a corresponding message acknowledging the deployment to the slave. Note that you may have to wait up to 3 minutes for the deployed archive to be transfered.

You're application is now running in a cluster with HTTP session replication!

36.2. EJB Passivation and the ManagedEntityInterceptor

The ManagedEntityInterceptor (MEI) is an optional interceptor in Seam that gets applied to conversation-scoped components when enabled. Enabling it is simple. You just set the distributable property on the org.jboss.seam.init.core component to true. More simply put, you add (or update) the following component declaration in the component descriptor (i.e., components.xml).

```
<core:init distributable="true"/>
```

Note that this doesn't enable replication of HTTP sessions, but it does prepare Seam to be able to deal with passivation of either EJB components or components in the HTTP session.

The MEI serves two distinct scenarios (EJB passivation and HTTP session passivation), although to accomplish the same overall goal. It ensures that throughout the life of a conversation using at least one extended persistence context, the entity instances loaded by the persistence context(s)

remain managed (they do not become detached prematurely by a passivation event). In short, it ensures the integrity of the extended persistence context (and therefore its guarantees).

The previous statement implies that there is a challenge that threatens this contract. In fact, there are two. One case is when a stateful session bean (SFSB) that hosts an extended persistence context is passivated (to save memory or to migrate it to another node in the cluster) and the second is when the HTTP session is passivated (to prepare it to be migrated to another node in the cluster).

I first want to discuss the general problem of passivation and then look at the two challenges cited individually.

36.2.1. The friction between passivation and persistence

The persistence context is where the persistence manager (i.e., JPA EntityManager or Hibernate Session) stores entity instances (i.e., objects) it has loaded from the database (via the object-relational mappings). Within a persistence context, there is no more than one object per unique database record. The persistence context is often referred to as the first-level cache because if the application asks for a record by its unique identifier that has already been loaded into the persistence context, a call to the database is avoided. But it's about more than just caching.

Objects held in the persistence context can be modified, which the persistence manager tracks. When an object is modified, it's considered "dirty". The persistence manager will migrate these changes to the database using a technique known as write-behind (which basically means only when necessary). Thus, the persistence context maintains a set of pending changes to the database.

Database-oriented applications do much more than just read from and write to the database. They capture transactional bits of information that need to be transferred into the database atomically (at once). It's not always possible to capture this information all on one screen. Additionally, the user might need to make a judgement call about whether to approve or reject the pending changes.

What we are getting at here is that the idea of a transaction from the user's perspective needs to be extended. And that is why the extended persistence context fits so perfectly with this requirement. It can hold such changes for as long as the application can keep it open and then use the built-in capabilities of the persistence manager to push these pending changes to the database without requiring the application developer to worry about the low-level details (a simple call to `EntityManager#flush()` does the trick).

The link between the persistence manager and the entity instances is maintained using object references. The entity instances are serializable, but the persistence manager (and in turn its persistence context) is not. Therefore, the process of serialization works against this design. Serialization can occur either when a SFSB or the HTTP session is passivated. In order to sustain the activity in the application, the persistence manager and the entity instances it manages must weather serialization without losing their relationship. That's the aid that the MEI provides.

36.2.2. Case #1: Surviving EJB passivation

Conversations were initially designed with stateful session beans (SFSBs) in mind, primarily because the EJB 3 specification designates SFSBs as hosts of the extended persistence context. Seam introduces a complement to the extended persistence context, known as a Seam-managed persistence context, which works around a number of limitations in the specification (complex propagation rules and lack of manual flushing). Both can be used with a SFSB.

A SFSB relies on a client to hold a reference to it in order to keep it active. Seam has provided an ideal place for this reference in the conversation context. Thus, for as long as the conversation context is active, the SFSB is active. If an EntityManager is injected into that SFSB using the annotation @PersistenceContext(EXTENDED), then that EntityManager will be bound to the SFSB and remain open throughout its lifetime, the lifetime of the conversation. If an EntityManager is injected using @In, then that EntityManager is maintained by Seam and stored directly in the conversation context, thus living for the lifetime of the conversation independent of the lifetime of the SFSB.

With all of that said, the Java EE container can passivate a SFSB, which means it will serialize the object to an area of storage external to the JVM. When this happens depends on the settings of the individual SFSB. This process can even be disabled. However, the persistence context is not serialized (is this only true of SMPC?). In fact, what happens depends highly on the Java EE container. The spec is not very clear about this situation. Many vendors just tell you not to let it happen if you need the guarantees of the extended persistence context. Seam's approach is more conservative. Seam basically doesn't trust the SFSB with the persistence context or the entity instances. After each invocation of the SFSB, Seam moves the reference to entity instance held by the SFSB into the current conversation (and therefore into the HTTP session), nullifying those fields on the SFSB. It then restores this references at the beginning of the next invocation. Of course, Seam is already storing the persistence manager in the conversation. Thus, when the SFSB passivates and later activates, it has absolutely no averse affect on the application.

Note

If you are using SFSBs in your application that hold references to extended persistence contexts, and those SFSBs can passivate, then you must use the MEI. This requirement holds even if you are using a single instance (not a cluster). However, if you are using clustered SFSB, then this requirement also applies.

It is possible to disable passivation on a SFSB. See the *Ejb3DisableSfsbPassivation* [http://www.jboss.org/community/docs/DOC-9656] page on the JBoss Wiki for details.

36.2.3. Case #2: Surviving HTTP session replication

Dealing with passivation of a SFSB works by leveraging the HTTP session. But what happens when the HTTP session passivates? This happens in a clustered environment with session replication enabled. This case is much tricker to deal with and is where a bulk of the MEI

infrastructure comes into play. In this case, the persistence manager is going to be destroyed because it cannot be serialized. Seam handles this deconstruction (hence ensuring that the HTTP session serializes properly). But what happens on the other end. Well, when the MEI sticks an entity instance into the conversation, it embeds the instance in a wrapper that provides information on how to re-associate the instance with a persistence manager post-serialization. So when the application jumps to another node in the cluster (presumably because the target node went down) the MEI infrastructure essentially reconstructs the persistence context. The huge drawback here is that since the persistence context is being reconstructed (from the database), pending changes are dropped. However, what Seam does do is ensure that if the entity instance is versioned, that the guarantees of optimistic locking are upheld. (why isn't the dirty state transferred?)

Note

If you are deploying your application in a cluster and using HTTP session replication, you must use the MEI.

36.2.4. ManagedEntityInterceptor wrap-up

The important point of this section is that the MEI is there for a reason. It's there to ensure that the extended persistence context can retain intact in the face of passivation (of either a SFSB or the HTTP session). This matters because the natural design of Seam applications (and conversational state in general) revolve around the state of this resource.

Performance Tuning

This chapter is an attempt to document in one place all the tips for getting the best performance from your Seam application.

37.1. Bypassing Interceptors

For repetitive value bindings such as those found in a JSF dataTable or other iterative control (like `ui:repeat`), the full interceptor stack will be invoked for every invocation of the referenced Seam component. The effect of this can result in a substantial performance hit, especially if the component is accessed many times. A significant performance gain can be achieved by disabling the interceptor stack for the Seam component being invoked. To disable interceptors for the component, add the `@BypassInterceptors` annotation to the component class.

Warning

It is very important to be aware of the implications of disabling interceptors for a Seam component. Features such as bijection, annotated security restrictions, synchronization and others are unavailable for a component marked with `@BypassInterceptors`. While in most cases it is possible to compensate for the loss of these features (e.g. instead of injecting a component using `@In`, you can use `Component.getInstance()` instead) it is important to be aware of the consequences.

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;
  }
}
```

Testing Seam applications

Most Seam applications will need 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 kinds of tests are very easy to write.

38.1. Unit testing Seam components

All Seam components are POJOs. This is a great place to start if you want easy unit testing. And since Seam emphasises the use of bijection for inter-component interactions and access to contextual objects, it's very easy to test a Seam component outside of its normal runtime environment.

Consider 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 could write a unit test for the calculateTotal method (which tests the business logic of the component) 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;
   }
}
```

You'll notice we aren't testing retrieving data from or persisting data to the database; nor are we testing any functionality provided by Seam. We are just testing the logic of our POJOs. Seam components don't usually depend directly upon container infrastructure, so most unit testing are as easy as that!

However, if you want to test the entire application, read on.

38.2. Integration testing Seam components

Warning

Using JBoss Embedded for integration testing was removed. Seam uses Arquillian with JUnit. Right now TestNG is not recommended test framework with Arquillian.

Integration testing is slightly more difficult. In this case, we can't eliminate the container infrastructure; indeed, that is part of what is being tested! At the same time, we don't want to be forced to deploy our application to an application server to run the automated tests. We need to be able to reproduce just enough of the container infrastructure inside our testing environment to be able to exercise the whole application, without hurting performance too much.

The approach taken by Seam is to let you write tests that exercise your components while running inside a pruned down container environment (Seam, together with the JBoss AS container)

Arquillian makes it possible to run integration tests inside a real container, even without `SeamTest`.

Example 38.1. RegisterTest.java

```
@RunWith(Arquillian)                                                    (1)
public class RegisterTest
{
  @Deployment                                                           (2)
  @OverProtocol("Servlet 3.0")                                          (3)
  public static Archive<?> createDeployment()
  {
    EnterpriseArchive er = ShrinkWrap.create(ZipImporter.class)         (4)
      .importFrom(new File("../registration-ear/target/seam-registration.ear"))
      .as(EnterpriseArchive.class);
    WebArchive web = er.getAsType(WebArchive.class, "registration-web.war");
    web.addClasses(RegisterTest.class);                                 (5)
    return er;
  }

  @Before
  public void before()
  {
     Lifecycle.beginCall();                                             (6)
  }

  @After
  public void after(
  {
     Lifecycle.endCall();
  }

  protected void setValue(String valueExpression, Object value)
  {
    Expressions.instance().createValueExpression(valueExpression).setValue(value);
  }

  @Test
  public void testRegisterComponent() throws Exception
  {
   setValue("#{user.username}", "1ovthafew");
   setValue("#{user.name}", "Gavin King");
   setValue("#{user.password}", "secret");
   Register register = (Register)Component.getInstance("register");
   Assert.assertEquals("success", register.register());
```

```
  }

  ...
}
```

1. The JUnit `@RunWith` annotation must be present to run our tests with Arquillian.
2. Since we want to run our test in a real container, we need to specify an archive that gets deployed.
3. `@OverProtocol` is an Arquillian annotation to specify the protocol used for running the tests. The "Servlet 3.0" protocol is the recommended protocol for running Seam tests.
4. ShrinkWrap can be used to create the deployment archive. In this example, the whole EAR is imported, but we could also use the ShrinkWrap API to create a WAR or an EAR from the scratch and put in just the artifacts that we need for the test.
5. The test class itself must be added to the web archive.
6. `Lifecycle.beginCall()` is needed to setup Seam contexts.

38.2.1. Configuration

The Arquillian configuration depends on the specific container used. See Arquillian documentation for more information.

Assuming you are using Maven as your build tool and want run your tests on JBoss AS 7, you will need to put these dependencies into your `pom.xml`:

```
<dependency>
   <groupId>org.jboss.arquillian.junit</groupId>
   <artifactId>arquillian-junit-container</artifactId>
   <version>${version.arquillian}</version>
   <scope>test</scope>
</dependency>

<dependency>
   <groupId>org.jboss.as</groupId>
   <artifactId>jboss-as-arquillian-container-managed</artifactId>
   <version>${version.jboss.as7}</version>
   <scope>test</scope>
</dependency>
```

The Arquillian JBoss AS Managed Container will automatically start the application server, provided the JBOSS_HOME environment property points to the JBoss AS 7 installation.

38.2.2. Using JUnitSeamTest with Arquillian

It is also possible to use the simulated JSF environment provided by `SeamTest` along with Arquillian. This is useful especially if you are migrating from previous Seam releases and want to keep your existing testsuite mostly unchanged.

Note

SeamTest was primary designated for TestNG integration tests. There are some glitches so we recommend to use JUnitSeamTest which is the JUnit variant for SeamTest.

The following changes must be done to run a JUnitSeamTest with Arquillian:

- Create the `@Deployment` method.
- Convert the test to JUnit. A `JUnitSeamTest` class can now be used instead of the original `SeamTest`.
- Replace the `SeamListener` with `org.jboss.seam.mock.MockSeamListener` in web.xml.

Example 38.2. RegisterTest.java

```
@RunWith(Arquillian)
public class RegisterTest extends JUnitSeamTest
{
  @Deployment
  @OverProtocol("Servlet 3.0")
  public static Archive<?> createDeployment()
  {
    EnterpriseArchive er = ShrinkWrap.create(ZipImporter.class)
      .importFrom(new File("../registration-ear/target/seam-registration.ear"))
      .as(EnterpriseArchive.class);
    WebArchive web = er.getAsType(WebArchive.class, "registration-web.war");
    web.addClasses(RegisterTest.class);

    // Replacing the SeamListener with MockSeamListener
    web.delete("/WEB-INF/web.xml");
    web.addAsWebInfResource("WEB-INF/mock-web.xml", "web.xml");

    return er;
  }

  @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();

  }

  ...
}
```

Example 38.3. mock-web.xml

```
<?xml version="1.0" ?>
<web-app version="3.0"
        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_3_0.xsd">

   <listener>
      <listener-class>org.jboss.seam.mock.MockSeamListener</listener-class>
   </listener>

</web-app>
```

38.2.2.1. Using mocks in integration tests

Occasionally, we need to be able to replace the implementation of some Seam component that depends upon resources which are not available in the integration test environment. For example, suppose we have some Seam component which is a facade to some payment processing system:

```
@Name("paymentProcessor")
public class PaymentProcessor {
   public boolean processPayment(Payment payment) { .... }
}
```

For integration tests, we can mock out this component as follows:

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
   public boolean processPayment(Payment payment) {
      return true;
   }
}
```

Since the `MOCK` precedence is higher than the default precedence of application components, 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.

38.2.3. Integration testing Seam application user interactions

An even harder problem is emulating user interactions. A third problem is where to put our assertions. Some test frameworks let us test the whole application by reproducing user interactions with the web browser. These frameworks have their place, but they are not appropriate for use at development time.

`SeamTest` or `JUnitSeamTest` lets you write *scripted* tests, in a simulated JSF environment. The role of a scripted test is to reproduce the interaction between the view and the Seam components. In other words, you get to pretend you are the JSF implementation!

This approach tests everything except the view.

Let's consider a JSF 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 (the stuff that happens when the user clicks the Register button). We'll reproduce the JSF request lifecycle in an automated JUnit test:

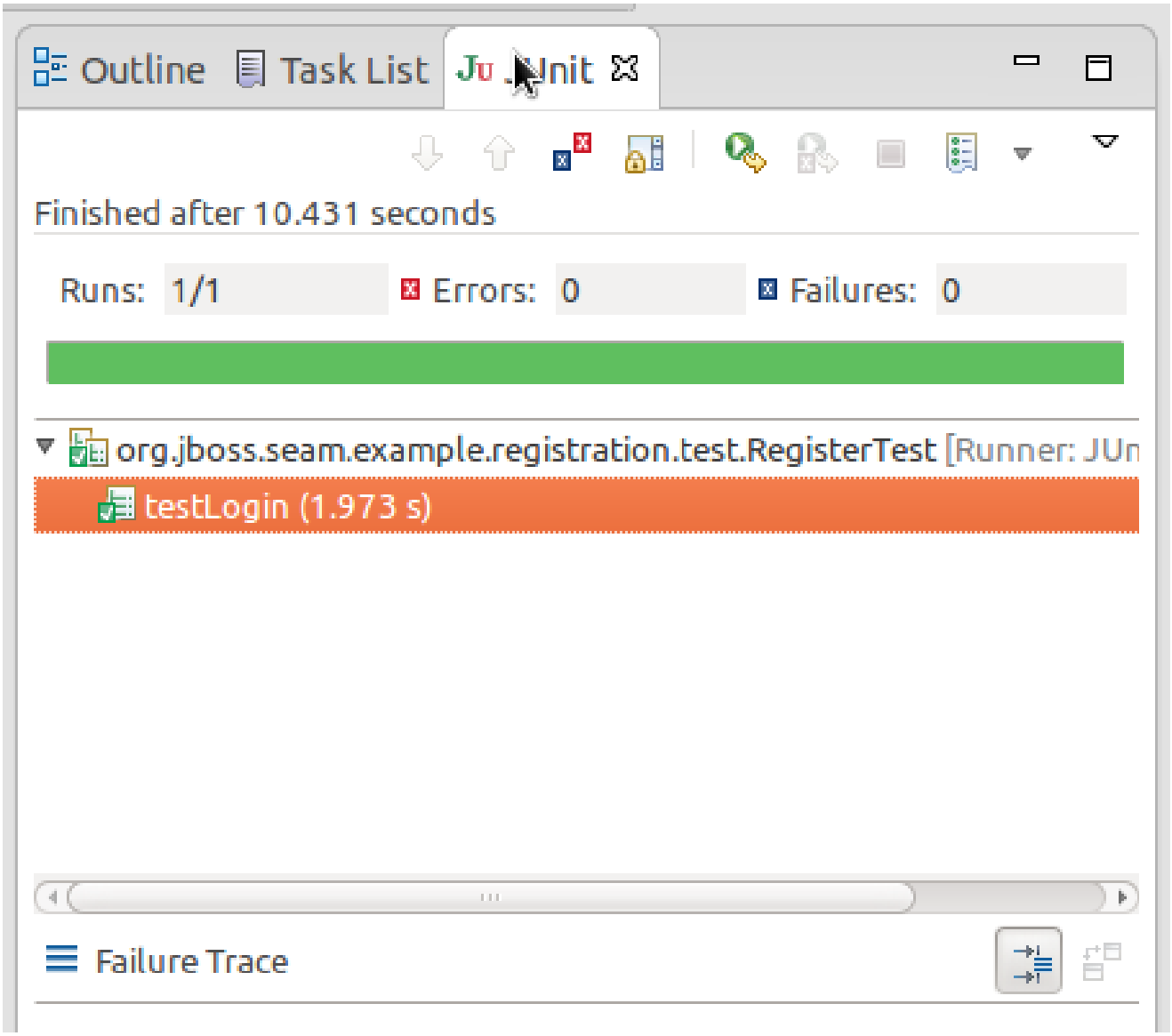
```
@RunWith(Arquillian.class)
public class RegisterTest extends JUnitSeamTest
{
   @Deployment(name="RegisterTest")
   @OverProtocol("Servlet 3.0")
   public static Archive<?> createDeployment()
   {
                  EnterpriseArchive er = ShrinkWrap.create(ZipImporter.class, "seam-
registration.ear").importFrom(new File("../registration-ear/target/seam-registration.ear"))
            .as(EnterpriseArchive.class);
      WebArchive web = er.getAsType(WebArchive.class, "registration-web.war");
      web.addClasses(RegisterTest.class);

      // Install org.jboss.seam.mock.MockSeamListener
```

```
    web.delete("/WEB-INF/web.xml");
    web.addAsWebInfResource("web.xml");

    return er;
  }

  @Test
  public void testLogin() throws Exception
  {

     new FacesRequest("/register.xhtml") {

        @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("/registered.xhtml");
           setOutcome("/registered.xhtml");
        }

        @Override
        protected void afterRequest()
        {
           assert isInvokeApplicationComplete();
           assert !isRenderResponseBegun();
        }

     }.run();
```

```
    ...
}
```

Notice that we've extended `JUnitSeamTest`, which provides a Seam environment for our components, and written our test script as an anonymous class that extends `JUnitSeamTest.FacesRequest`, which provides an emulated JSF request lifecycle. (There is also a `JUnitSeamTest.NonFacesRequest` for testing GET requests.) We've written our code in methods which are named for the various JSF phases, to emulate the calls that JSF would make to our components. Then we've thrown in various assertions.

You'll find plenty of integration tests for the Seam example applications which demonstrate more complex cases. There are instructions for running these tests using Maven, or using the JUnit plugin for eclipse:



38.2.3.1. Configuration

If you used seam-gen to create your project you are ready to start writing tests. Otherwise you'll need to setup the testing environment in your favorite build tool (e.g. ant, maven, eclipse).

For ant or own build tool which uses jars on local - you can use to get all jars by running `ant -f get-arquillian-libs.xml -Dtest.lib.dir=lib/test`. This just copy all Arquillian jars for managed JBoss AS 7.1.1 container and copy all jars into defined directory `lib/test` by using that `test.lib.dir` property.

And, of course you need to put your built project and tests onto the classpath as well as jar for your test framework. Don't forget to put all the correct configuration files for JPA and Seam onto the classpath as well. Seam asks Arquillian to deploy any resource (jar or directory) which has `seam.properties` in it's root. Therefore, if you don't assemble a directory structure that resembles a deployable archive containing your built project, you must put a `seam.properties` in each resource.

38.2.3.2. Using SeamTest with another test framework

Seam provides JUnit support out of the box, but you can also use another test framework, if you want.

You'll need to provide an implementation of `AbstractSeamTest` which does the following:

- Calls `super.begin()` before every test method.
- Calls `super.end()` after every test method.
- Calls `super.setupClass()` to setup integration test environment. This should be called before any test methods are called.
- Calls `super.cleanupClass()` to clean up the integration test environment.
- Calls `super.startSeam()` to start Seam at the start of integration testing.
- Calls `super.stopSeam()` to cleanly shut down Seam at the end of integration testing.

38.2.3.3. Integration Testing with Mock Data

If you want to insert or clean data in your database before each test you can use Seam's integration with DBUnit. To do this, extend `DBJUnitSeamTest` rather than `JUnitSeamTest`.

You have to provide a dataset for DBUnit.

Caution

DBUnit supports two formats for dataset files, flat and XML. Seam's `DBJUnitSeamTest` assumes the flat format is used, 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 with overriding `prepareDBUnitOperations()`:

```
protected void prepareDBUnitOperations() {
   setDatabase("HSQL");
   setDatasourceJndiName("java:/jboss/myDatasource");

   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 above 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 `afterTestOperations` list.

You need to tell DBUnit which datasource you are using. This is accomplished by calling `setDatasourceJndiName`.

DBJUnitSeamTest has support for MySQL and HSQL - you need to tell it which database is being used, otherwise it defaults to HSQL.

It also allows you to insert binary data into the test data set (n.b. this is untested on Windows). You need to tell it where to locate these resources on your classpath:

```
setBinaryUrl("images/");
```

You do not have to configure any of these parameters except the `datasourceJndiName` if you use HSQL and have no binary imports. You 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 `DBJUnitSeamTest` for more details.

38.2.3.4. Integration Testing Seam Mail

Caution

Warning! This feature is still under development.

It's 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();
  }
}
```

We create a new `FacesRequest` as normal. Inside the invokeApplication hook we render the message using `getRenderedMailMessage(viewId);`, passing the viewId of the message to render. The method returns the rendered message on which you can do your tests. You can of course also use any of the standard JSF lifecycle methods.

There is no support for rendering standard JSF components so you can't test the content body of the mail message easily.

Dependencies

39.1. JDK Dependencies

Seam does not work with JDK 1.4 and requires JDK 5 or above as it uses annotations and other JDK 5.0 features. Seam has been thoroughly tested using Oracle's JDKs and OpenJDKs. However there are no known issues specific to Seam with other JDKs.

39.1.1. Oracle's JDK 6 Considerations

Earlier versions of Oracle's JDK 6 contained an incompatible version of JAXB and required overriding it using the "endorsed" directory. Oracle's JDK6 Update 4 release upgraded to JAXB 2.1 and removed this requirement. When building, testing, or executing be sure to use this version or higher.

39.2. Project Dependencies

This section both lists the compile-time and runtime dependencies for Seam. Where the type is listed as `ear`, the library should be included in the /lib directory of your application's ear file. Where the type is listed as `war`, the library should be placed in the `/WEB-INF/lib` directory of your application's war file. The scope of the dependency is either all, runtime or provided (by JBoss AS 7.1.x).

Up to date version information and complete dependency information is not included in the docs, but is provided in the `/dependency-report.txt` which is generated from the Maven POMs stored in `/build`. You can generate this file by running `ant dependencyReport`.

39.2.1. Core

Table 39.1.

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
`jboss-seam-excel.jar`	runtime	war	Required when using Seam's Microsoft® Excel® features

Name	Scope	Type	Notes
jboss-seam-rss.jar	runtime	war	Required when using Seam's RSS generation features
jboss-seam-remoting.jar	runtime	war	Required when using Seam Remoting
jboss-seam-ui.jar	runtime	war	Required to use the Seam JSF controls
jsf-api.jar	provided		JSF API
jsf-impl.jar	provided		JSF Reference Implementation
urlrewrite.jar	runtime	war	URL Rewrite library
quartz.jar	runtime	ear	Required when you wish to use Quartz with Seam's asynchronous features

39.2.2. RichFaces

Table 39.2. RichFaces dependencies

Name	Scope	Type	Notes
richfaces-core-api.jar	all	ear	Required to use RichFaces. Provides Core API classes that you may wish to use from your application e.g. to create a tree
richfaces-core-impl.jar	runtime	war	Required to use RichFaces Core implementations.
richfaces-components-ui.jar	runtime	war	Required to use RichFaces. Provides all the Components UI components.
richfaces-components-api.jar	runtime	war	Required to use RichFaces. Provides all the API for UI components.

39.2.3. Seam Mail

Table 39.3. Seam Mail Dependencies

Name	Scope	Type	Notes
mail.jar	runtime	ear	Required for outgoing mail support
ironjacamar-mail.jar	compile only		Required for incoming mail support

Name	Scope	Type	Notes
			`ironjacamar-mail.jar` should be deployed to the application server at runtime
`jboss-seam-mail.jar`	runtime	war	Seam Mail

39.2.4. Seam PDF

Table 39.4. 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

39.2.5. Seam Microsoft® Excel®

Table 39.5. 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

39.2.6. Seam RSS support

Table 39.6. Seam RSS Dependencies

Name	Type	Scope	Notes
`yarfraw.jar`	runtime	war	YARFRAW RSS library
`JAXB`	runtime	war	JAXB XML parsing libraries
`http-client.jar`	runtime	war	Apache HTTP Client libraries
`commons-io`	runtime	war	Apache commons IO library
`commons-lang`	runtime	war	Apache commons lang library
`commons-codec`	runtime	war	Apache commons codec library
`commons-collections`	runtime	war	Apache commons collections library
`jboss-seam-rss.jar`	runtime	war	Seam RSS core library

39.2.7. Drools

The Drools libraries can be found in the `lib` directory in Seam.

Table 39.7. Drools Dependencies

Name	Scope	Type	Notes
`antlr-runtime.jar`	runtime	ear	ANTLR Runtime Library
`ecj.jar`	runtime	ear	Eclipse Compiler for Java
`knowledge-api.jar`	runtime	ear	
`drools-compiler.jar`	runtime	ear	Drools compiler
`drools-core.jar`	runtime	ear	
`drools-decisiontables.jar`	runtime	ear	
`drools-templates.jar`	runtime	ear	
`mvel2.jar`	runtime	ear	

39.2.8. JBPM

Table 39.8. JBPM dependencies

Name	Scope	Type	Notes
`jbpm-jpdl.jar`	runtime	ear	

39.2.9. GWT

These libraries are required if you with to use the Google Web Toolkit (GWT) with your Seam application.

Table 39.9. GWT dependencies

Name	Scope	Type	Notes
`gwt-servlet.jar`	runtime	war	The GWT Servlet libs

39.2.10. Spring

These libraries are required if you with to use the Spring Framework with your Seam application.

Table 39.10. Spring Framework dependencies

Name	Scope	Type	Notes
`spring.jar`	runtime	ear	The Spring Framework library

39.2.11. Groovy

These libraries are required if you with to use Groovy with your Seam application.

Table 39.11. Groovy dependencies

Name	Scope	Type	Notes
`groovy-all.jar`	runtime	ear	The Groovy libs

39.3. Dependency Management using Maven

We aren't actually going to discuss how to use Maven here, but just run over some Seam usage from user/application point of view you could use.

Released versions of Seam are available in *http://repository.jboss.org/nexus/content/groups/public* [http://repository.jboss.org/nexus/content/groups/public].

All the Seam artifacts are available in Maven:

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ui</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-pdf</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-mail</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-debug</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-remoting</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ioc</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-excel</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-resteasy</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-rss</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-wicket</artifactId>
```

```
</dependency>
```

This sample POM will give you Seam, JPA (provided by Hibernate), Hibernate Validator and Hibernate Search:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.seam.example/groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <name>My Seam Project</name>
  <packaging>jar</packaging>
  <repositories>
    <repository>
      <id>repository.jboss.org</id>
      <name>JBoss Public Repository</name>
      <url>http://repository.jboss.org/nexus/content/groups/public</url>
    </repository>
  </repositories>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.seam</groupId>
        <artifactId>bom</artifactId>
        <version>2.3.1.Final</version>
        <type>pom</type>
        <scope>import</scope>
       </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
    </dependency>
```

```
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-search</artifactId>
    </dependency>

    <dependency>
      <groupId>org.jboss.seam</groupId>
      <artifactId>jboss-seam</artifactId>
    </dependency>

  </dependencies>

</project>
```