Seam - Componenti Contestuali

Un framework per Java Enterprise

2.1.2.CR2

di Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Michael Yuan, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, e Marek Novotny

edited by Samson Kittoli

and thanks to James Cobb (Design Grafico), Cheyenne Weaver (Design Grafico), Mark Newton, Steve Ebersole, Michael Courcy (Traduzione in francese), Nicola Benaglia (Traduzione in italiano), Stefano Travelli (Traduzione in italiano), e Japan JBoss User Group (Traduzione in giapponese)

Introduzione a JBoss Seam xvii
1. Contribuire a Seam xxi
1. Tutorial di Seam 1
1.1. Utilizzo degli esempi di Seam 1
1.1.1. Eseguire gli esempi in JBoss AS 1
1.1.2. Eseguire gli esempi in Tomcat 2
1.1.3. Eseguire i test degli esempi 2
1.2. La prima applicazione di Seam: esempio di registrazione 2
1.2.1. Capire il codice 3
1.2.2. Come funziona 15
1.3. Liste cliccabili in Seam: esempio di messaggi 16
1.3.1. Capire il codice 16
1.3.2. Come funziona 22
1.4. Seam e jBPM: esempio di lista todo 23
1.4.1. Capire il codice 23
1.4.2. Come funziona 31
1.5. Seam pageflow: esempio di indovina-numero 32
1.5.1. Capire il codice 32
1.5.2. Come funziona 41
1.6. Un'applicazione Seam completa: esempio di Prenotazione Hotel 42
1.6.1. Introduzione 42
1.6.2. Panoramica sull'esempio di prenotazione 44
1.6.3. Capire le conversazioni in Seam 44
1.6.4. La pagina di debug di Seam 53
1.7. Conversazioni Annidate: estendere l'esempio di Prenotazione Hotel 54
1.7.1. Introduzione 54
1.7.2. Capire le Conversazioni Annidate 56
1.8. Un'applicazione completa di Seam e jBPM: esempio di Negozio DVD 63
1.9. URL segnalibro con l'esempio Blog 65
1.9.1. Utilizzo di MVC "pull"-style 66
1.9.2. Pagina bookmarkable dei risultati di ricerca 68
1.9.3. Uso di MVC push in un'applicazione RESTful 72
2. Iniziare con Seam usando seam-gen 77
2.1. Prima di iniziare 77
2.2. Comfigurare un nuovo progetto 78
2.3. Creazione di una nuova azione 81
2.4. Creazione di una form con un'azione 82
2.5. Generazione di un'applicazione da database esistente 83
2.6. Generazione di un'applicazione da entity JPA/EJB3 già esistenti 84
2.7. Eseguire il deploy dell'applicazione come EAR 84
2.8. Seam e hot deploy incrementale 84
2.9. Uso di Seam con JBoss 4.0 85
2.9.1. Install JBoss 4.0 85
2.9.2. Installare JSF 1.2 RI 86

3. Iniziare con Seam usando JBoss Tools 87
3.1. Prima di iniziare 87
3.2. Configurare un nuovo progetto Seam 87
3.3. Creazione di una nuova azione 103
3.4. Creazione di una form con un'azione 105
3.5. Generare un'applicazione da un database esistente 107
3.6. Seam ed l'hot deploy incrementale con JBoss Tools 108
4. Il modello a componenti contestuali 109
4.1. Contesti di Seam 109
4.1.1. Contesto Stateless 109
4.1.2. Contesto Evento 110
4.1.3. Contesto Pagina 110
4.1.4. Contesto Conversazione 110
4.1.5. Contesto Sessione 111
4.1.6. Contesto processo di Business 111
4.1.7. Contesto Applicazione 111
4.1.8. Variabili di contesto 111
4.1.9. Priorità di ricerca del contesto 112
4.1.10. Modello di concorrenza 112
4.2. Componenti di Seam 113
4.2.1. Bean di sessione stateless 113
4.2.2. Bean di sessione stateful 114
4.2.3. Entity bean 114
4.2.4. JavaBeans 115
4.2.5. Message-driven bean 115
4.2.6. Intercettazione 116
4.2.7. Nomi dei componenti 116
4.2.8. Definire lo scope di un componente 118
4.2.9. Componenti con ruoli multipli 119
4.2.10. Componenti predefiniti 119
4.3. Bijection 120
4.4. Metodi del ciclo di vita 123
4.5. Installazione condizionale 123
4.6. Logging 125
4.7. L'interfaccia `Mutable` e `@ReadOnly` 126
4.8. Componenti factory e manager 128
5. Configurare i componenti Seam 131
5.1. Configurare i componenti tramire impostazioni di proprietà 131
5.2. Configurazione dei componenti tramite `components.xml` 131
5.3. File di configurazione a grana fine 135
5.4. Tipi di proprietà configurabili 136
5.5. Uso dei namespace XML 139
6. Eventi, interceptor e gestione delle eccezioni 145
6.1. Eventi di Seam 145

6.2. Azioni di pagina 146
6.3. Parametri di pagina 147
6.3.1. Mapping request parameters to the model 148
6.4. Parametri di richiesta che si propagano 148
6.5. URL rewriting with page parameters 149
6.6. Conversione e validazione 150
6.7. Navigazione 151
6.8. Fine-grained files for definition of navigation, page actions and parameters 155
6.9. Eventi guidati da componenti 155
6.10. Eventi contestuali 158
6.11. Interceptor Seam 160
6.12. Gestione delle eccezioni 162
6.12.1. Eccezioni e transazioni 162
6.12.2. Abilitare la gestione delle eccezioni di Seam 162
6.12.3. Uso delle annotazioni per la gestione delle eccezioni 163
6.12.4. Uso di XML per la gestione delle eccezioni 164
6.12.5. Alcune eccezioni comuni 166
6.13. conversation-required 167
7. Conversazioni e gestione del workspace 169
7.1. Il modello di conversazioni di Seam 169
7.2. Conversazioni innestate 172
7.3. Avvio di conversazioni con richieste GET 173
7.4. Usando `<s:link>` e `<s:button>` 175
7.5. Messaggi di successo 176
7.6. Id di una conversazione naturale 177
7.7. Creazione di una conversazione naturale 178
7.8. Redirezione alla conversazione naturale 179
7.9. Gestione del workspace 180
7.9.1. Gestione del workspace e navigazione JSF 180
7.9.2. Gestione del workspace e pageflow jPDL 181
7.9.3. Lo switcher delle conversazioni 181
7.9.4. La lista delle conversazioni 182
7.9.5. Breadcrumbs 183
7.10. Componenti conversazionali ed associazione ai componenti JSF 184
7.11. Chiamare concorrenti ai componenti conversazionali 185
7.11.1. Come si può progettare la nostra applicazione AJAX conversazionale?... 186
7.11.2. Gestione degli errori 187
7.11.3. RichFaces (Ajax4jsf) 188
8. Pageflows e processi di business 191
8.1. Pageflow in Seam 191
8.1.1. I due modelli di navigazione 192
8.1.2. Seam ed il pulsante indietro 196
8.2. Utilizzo dei pageflow jPDL 197
8.2.1. Installazione dei pageflow 197

8.2.2. Avvio dei pageflow 198
8.2.3. Nodi e transizioni di pagina 199
8.2.4. Controllo del flusso 200
8.2.5. Fine del flusso 201
8.2.6. Composizione dei pageflow 201
8.3. La gestione del processo di business in Seam 202
8.4. Uso di jPDL nella definizione del processo di business 203
8.4.1. Installazione delle definizioni di processo 203
8.4.2. Inizializzazione degli actor id 204
8.4.3. Iniziare un processo di business 204
8.4.4. Assegnazione task 204
8.4.5. Liste di task 205
8.4.6. Esecuzione di un task 206
9. Seam e Object/Relational Mapping 209
9.1. Introduzione 209
9.2. Transazioni gestite da Seam 210
9.2.1. Disabilitare le transazioni gestite da Seam 211
9.2.2. Configurazione di un gestore di transazioni Seam 211
9.2.3. Sincronizzazione delle transazioni 212
9.3. Contesti di persistenza gestiti da Seam 212
9.3.1. Using a Seam-managed persistence context with JPA 213
9.3.2. Uso delle sessioni Hibernate gestite da Seam 213
9.3.3. Seam-managed persistence contexts and atomic conversations 214
9.4. Usare il JPA "delegate" 216
9.5. Uso di EL in EJB-QL/HQL 217
9.6. Uso dei filtri Hibernate 217
10. Validazione delle form JSF in Seam 219
11. Integrazione con Groovy 227
11.1. Introduzione a Groovy 227
11.2. Scrivere applicazioni Seam in Groovy 227
11.2.1. Scrivere componenti Groovy 227
11.2.2. seam-gen 229
11.3. Esecuzione 229
11.3.1. Eseguire il codice Groovy 230
11.3.2. Esecuzione di file .groovy durante lo sviluppo 230
11.3.3. seam-gen 230
12. Scrivere la parte di presentazione usando Apache Wicket 231
12.1. Aggiungere Seam ad un'applicazione Wicket 231
12.1.1. Bijection 231
12.1.2. Orchestrazione 232
12.2. Impostare il progetto 233
12.2.1. Definire l'applicazione 234
13. Seam Application Framework 237
13.1. Introduzione 237

13.2. Oggetti Home 239
13.3. Oggetti Query 245
13.4. Oggetti controllori 248
14. Seam e JBoss Rules 251
14.1. Installazione delle regole 251
14.2. Utilizzo delle regole da un componente SEAM 252
14.3. Utilizzo delle regole da una definizione di processo jBPM 253
15. Sicurezza 257
15.1. Panoramica 257
15.2. Disabilitare la sicurezza 257
15.3. Autenticazione 258
15.3.1. Configurare un componente Authenticator 258
15.3.2. Scrivere un metodo di autenticazione 259
15.3.3. Scrivere una form di accesso 262
15.3.4. Riepilogo della configurazione 262
15.3.5. Ricordami su questo computer 262
15.3.6. Gestire le eccezioni della sicurezza 266
15.3.7. Redirezione alla pagina di accesso 267
15.3.8. Autenticazione HTTP 268
15.3.9. Caratteristiche di autenticazione avanzate 269
15.4. Gestione delle identità 270
15.4.1. Configurare l'IdentityManager 270
15.4.2. JpaIdentityStore 271
15.4.3. LdapIdentityStore 278
15.4.4. Scrivere il proprio IdentityStore 282
15.4.5. L'autenticazione con la gestione delle identità 282
15.4.6. Usare IdentityManager 282
15.5. Messaggi di errore 289
15.6. Autorizzazione 289
15.6.1. Concetti principali 289
15.6.2. Rendere sicuri i componenti 290
15.6.3. La sicurezza nell'interfaccia utente 293
15.6.4. Rendere sicure le pagine 294
15.6.5. Rendere sicure le entità 295
15.6.6. Annotazioni tipizzate per i permessi 298
15.6.7. Annotazioni tipizzate per i ruoli 299
15.6.8. Il modello di autorizzazione dei permessi 300
15.6.9. RuleBasedPermissionResolver 303
15.6.10. PersistentPermissionResolver 308
15.7. Permission Management 320
15.7.1. PermissionManager 320
15.7.2. Permission checks for PermissionManager operations 323
15.8. SSL Security 323
15.8.1. Overriding the default ports 324

15.9. CAPTCHA 325
15.9.1. Configuring the CAPTCHA Servlet 325
15.9.2. Adding a CAPTCHA to a form 325
15.9.3. Customising the CAPTCHA algorithm 326
15.10. Security Events 326
15.11. Run As 327
15.12. Extending the Identity component 328
15.13. OpenID 329
15.13.1. Configuring OpenID 329
15.13.2. Presenting an OpenIdDLogin form 330
15.13.3. Logging in immediately 330
15.13.4. Deferring login 331
15.13.5. Logging out 331
16. Internazionalizzazione, localizzazione e temi 333
16.1. Internazionalizzare un'applicazione 333
16.1.1. Configurazione dell'application server 333
16.1.2. Traduzione delle stringhe dell'applicazione 333
16.1.3. Altre impostazioni per la codifica 334
16.2. Traduzioni 334
16.3. Etichette 336
16.3.1. Definire le etichette 336
16.3.2. Mostrare le etichette 337
16.3.3. Messaggi Faces 338
16.4. Fusi orari 338
16.5. Temi 339
16.6. Registrare la scelta della lingua e del tema tramite cookies 340
17. Seam Text 341
17.1. Formattazione di base 341
17.2. Inserire codice e testo con caratteri speciali 344
17.3. Link 345
17.4. Inserire codice HTML 345
18. Generazione di PDF con iText 347
18.1. Utilizzo del supporto PDF 347
18.1.1. Creazione di un documento 347
18.1.2. Elementi base per il testo 348
18.1.3. Intestazioni e pié di pagina 353
18.1.4. Capitoli e Sezioni 354
18.1.5. Liste 356
18.1.6. Tabelle 357
18.1.7. Costanti nei documenti 360
18.2. Grafici 361
18.3. Codici a barre 370
18.4. Form da riempire 371
18.5. Componenti per i rendering Swing/AWT 372

18.6. Configurazione di iText 373
18.7. Ulteriore documentazione 374
19. The Microsoft® Excel® spreadsheet application 375
19.1. Supporto The Microsoft® Excel® spreadsheet application 375
19.2. Creazione di un semplice workbook 376
19.3. Workbooks 377
19.4. Worksheets 379
19.5. Colonne 382
19.6. Celle 384
19.6.1. Validazione 385
19.6.2. Maschere per il formato 388
19.7. Formule 388
19.8. Immagini 389
19.9. Hyperlinks 390
19.10. Intestazioni e pié di pagina 391
19.11. Stampa di aree e titoli 393
19.12. Comandi per i fogli di lavoro (worksheet) 394
19.12.1. Raggruppamento 394
19.12.2. Interruzioni di pagina 395
19.12.3. Fusione (merge) 396
19.13. Esportatore di datatable 397
19.14. Font e layout 398
19.14.1. Link ai fogli di stile 398
19.14.2. Font 399
19.14.3. Bordi 399
19.14.4. Background 400
19.14.5. Impostazioni colonna 400
19.14.6. Impostazioni cella 401
19.14.7. L'exporter delle datatable 401
19.14.8. Esempi di layout 401
19.14.9. Limitazioni 402
19.15. Internazionalizzazione 402
19.16. Link ed ulteriore documentazione 402
20. Supporto RSS 403
20.1. Installazione 403
20.2. Generare dei feed 403
20.3. I feed 404
20.4. Elementi 404
20.5. Link e ulteriore documentazione 405
21. Email 407
21.1. Creare un messaggio 407
21.1.1. Allegati 408
21.1.2. HTML/Text alternative part 410
21.1.3. Destinatari multipli 410

21.1.4. Messaggi multipli 410
21.1.5. Comporre template 411
21.1.6. Internazionalizzazione 412
21.1.7. Altre intestazioni 412
21.2. Ricevere email 412
21.3. Configurazione 413
21.3.1. `mailSession` 414
21.4. Meldware 415
21.5. Tag 415
22. Asincronicità e messaggistica 419
22.1. Asincronicità 419
22.1.1. Metodi asincroni 420
22.1.2. Metodi asincroni con il Quartz Dispatcher 423
22.1.3. Eventi asincroni 426
22.1.4. Gestione delle eccezione da chiamate asincrone 426
22.2. Messaggistica in Seam 427
22.2.1. Configurazione 427
22.2.2. Spedire messaggi 427
22.2.3. Ricezione dei messaggi usando un bean message-driven 428
22.2.4. Ricezione dei messaggi nel client 429
23. Gestione della cache 431
23.1. Usare la cache in Seam 432
23.2. Cache dei frammenti di pagina 434
24. Web Service 437
24.1. Configurazione ed impacchettamento 437
24.2. Web Service conversazionali 438
24.2.1. Una strategia raccomandata 439
24.3. Esempio di web service 439
24.4. Webservice RESTful HTTP con RESTEasy 441
24.4.1. RESTEasy configuration and request serving 441
24.4.2. Risorse e provider come componenti Seam 444
24.4.3. Mapping exceptions to HTTP responses 445
25. Remoting 447
25.1. Configurazione 447
25.2. L'oggetto "Seam" 448
25.2.1. Esempio Hello World 448
25.2.2. Seam.Component 450
25.2.3. Seam.Remoting 452
25.3. Valutazione delle espressioni EL 453
25.4. Interfacce client 453
25.5. Il contesto 454
25.5.1. Impostazione e lettura dell'ID di conversazione 454
25.5.2. Remote calls within the current conversation scope 454
25.6. Richieste batch 455

25.7. Lavorare con i tipi di dati 455
25.7.1. Primitives / Basic Types 455
25.7.2. JavaBeans 456
25.7.3. Date e orari 456
25.7.4. Enums 456
25.7.5. Collections 457
25.8. Debugging 458
25.9. Gestione delle eccezioni 458
25.10. Il messaggio di caricamento 458
25.10.1. Cambiare il messaggio 459
25.10.2. Nascondere il messaggio di caricamento 459
25.10.3. A Custom Loading Indicator 459
25.11. Controlling what data is returned 459
25.11.1. Constraining normal fields 460
25.11.2. Constraining Maps and Collections 460
25.11.3. Constraining objects of a specific type 461
25.11.4. Combining Constraints 461
25.12. Richieste transazionali 461
25.13. Messaggistica JMS 462
25.13.1. Configurazione 462
25.13.2. Iscriversi ad un Topic JMS 462
25.13.3. Disiscriversi da un Topic 463
25.13.4. Tuning the Polling Process 463
26. Seam e il Google Web Toolkit 465
26.1. Configurazione 465
26.2. Preparare i componenti 465
26.3. Collegare un componente GWT ad un componente Seam 466
26.4. Target Ant per GWT 468
27. Integrazione con il framework Spring 471
27.1. Injecting Seam components into Spring beans 471
27.2. Injecting Spring beans into Seam components 473
27.3. Making a Spring bean into a Seam component 474
27.4. Seam-scoped Spring beans 474
27.5. Using Spring PlatformTransactionManagement 475
27.6. Using a Seam Managed Persistence Context in Spring 476
27.7. Using a Seam Managed Hibernate Session in Spring 478
27.8. Spring Application Context as a Seam Component 478
27.9. Using a Spring TaskExecutor for @Asynchronous 479
28. Integrazione con Guice 481
28.1. Creazione di un componente ibrido Seam-Guice 481
28.2. Configurare un injector 482
28.3. Uso di injector multipli 483
29. Hibernate Search 485
29.1. Introduzione 485

29.2. Configurazione 485
29.3. Utilizzo 487
30. Configurare Seam ed impacchettare le applicazioni Seam 491
30.1. Configurazione base di Seam 491
30.1.1. Integrazione di Seam con JSF ed il servlet container 491
30.1.2. Usare Facelets 492
30.1.3. Resource Servlet di Seam 493
30.1.4. Filtri servlet di Seam 494
30.1.5. Integrating Seam with your EJB container 498
30.1.6. Non dimenticare! 503
30.2. Uso di provider JPA alternativi 503
30.3. Configurazione di Seam in java EE 5 504
30.3.1. Packaging 504
30.4. Configurare Seam in J2EE 506
30.4.1. Boostrapping di Hibernate in Seam 506
30.4.2. Boostrapping di JPA in Seam 507
30.4.3. Packaging 507
30.5. Configurazione di Seam in java EE 5 senza JBoss Embedded 508
30.6. Configurazione di Seam in java EE 5 con JBoss Embedded 508
30.6.1. Installare JBoss Embedded 509
30.6.2. Packaging 511
30.7. Configurazione jBPM in Seam 512
30.7.1. Packaging 513
30.8. Configurazione di SFSB e dei timeout di sessione in JBoss AS 514
30.9. Esecuzione di Seam in un Portlet 516
30.10. Deploying custom resources 516
31. Annotazioni di Seam 519
31.1. Annotazioni per la definizione di un componente 519
31.2. Annotazioni per la bijection 523
31.3. Annotazioni per i metodi del ciclo di vita dei componenti 526
31.4. Annotazioni per la demarcazione del contesto 527
31.5. Annotazioni per l'uso con i componenti JavaBean di Seam in ambiente J2EE.... 531
31.6. Annotazioni per le eccezioni 532
31.7. Annotazioni per Seam Remoting 533
31.8. Annotazioni per gli interceptor di Seam 533
31.9. Annotazioni per l'asincronicità 534
31.10. Annotazioni per l'uso di JSF 535
31.10.1. Annotazioni per l'uso con `dataTable` 535
31.11. Meta-annotationi per il databinding 536
31.12. Annotazioni per i pacchetti 537
31.13. Annotazioni per l'integrazione con un servlet container 537
32. Componenti Seam predefiniti 539
32.1. Componenti per l'iniezione del contesto 539
32.2. Componenti JSF 539

32.3. Componenti d'utilità 541
32.4. Componenti per l'internazionalizzazione ed i temi 542
32.5. Componenti per il controllo delle conversazioni. 543
32.6. Componenti per jBPM 544
32.7. Componenti per la sicurezza 546
32.8. Componenti per JMS 546
32.9. Componenti relativi alla Mail 547
32.10. Componenti infrastrutturali 547
32.11. Componenti misti 550
32.12. Componenti speciali 550
33. Controlli JSF di Seam 553
33.1. Tag 553
33.1.1. Controlli di navigazione 553
33.1.2. Convertitori e Validatori 556
33.1.3. Formattazione 562
33.1.4. Seam Text 565
33.1.5. Supporto per le form 566
33.1.6. Altro 570
33.2. Annotazioni 574
34. JBoss EL 577
34.1. Espressioni parametrizzate 577
34.1.1. Utilizzo 577
34.1.2. Limitazioni e suggerimenti 578
34.2. Proiezione 580
35. Tuning delle performance 583
35.1. Bypassare gli interceptor 583
36. Test delle applicazioni Seam 585
36.1. Test d'unità dei componenti Seam 585
36.2. Test d'integrazione dei componenti Seam 586
36.2.1. Uso dei mock nei test d'intergrazione 587
36.3. Test d'integrazione delle interazioni utente in applicazioni Seam 588
36.3.1. Configurazione 592
36.3.2. Uso di SeamTest con un altro framework di test 593
36.3.3. Test d'integrazione con Dati Mock 593
36.3.4. Test d'integrazione di Seam Mail 595
37. Strumenti di Seam 597
37.1. Visualizzatore e designer jBPM 597
37.1.1. Designer del processo di business 597
37.1.2. Visualizzatore Pageflow 597
38. Seam on BEA's Weblogic 599
38.1. Installation and operation of Weblogic 599
38.1.1. Installing 10.3 600
38.1.2. Creating your Weblogic domain 600
38.1.3. How to Start/Stop/Access your domain 601

38.1.4. Setting up Weblogic's JSF Support 602
38.2. The jee5/booking Example 602
38.2.1. EJB3 Issues with Weblogic 602
38.2.2. Getting the jee5/booking Working 604
38.3. The jpa booking example 610
38.3.1. Building and deploying jpa booking example 610
38.3.2. What's different with Weblogic 10.x 611
38.4. Deploying an application created using seam-gen on Weblogic 10.x 613
38.4.1. Running seam-gen setup 614
38.4.2. What to change for Weblogic 10.X 615
38.4.3. Building and Deploying your application 618
39. Seam on IBM's Websphere AS 621
39.1. Websphere AS environment and deployment information 621
39.1.1. Installation versions 621
39.2. The jee5/booking example 621
39.2.1. Configuration file changes 621
39.2.2. Building the jee5/booking example 626
39.2.3. Deploying the application to Websphere 626
39.3. The jpa booking example 628
39.3.1. Building the jpa example 628
39.3.2. Deploying the jpa example 629
39.3.3. What's different for Websphere AS V7 629
39.4. Deploying an application created using seam-gen on Websphere V7 631
39.4.1. Running seam-gen Setup 631
39.4.2. Changes needed for deployment to Websphere 633
40. Seam on GlassFish application server 641
40.1. GlassFish environment and deployment information 641
40.1.1. Installazione 641
40.2. The jee5/booking example 642
40.2.1. Building the jee5/booking example 642
40.2.2. Deploying the application to GlassFish 642
40.3. The jpa booking example 643
40.3.1. Building the jpa example 643
40.3.2. Deploying the jpa example 643
40.3.3. What's different for GlassFish v2 UR2 644
40.4. Deploying an application generated by seam-gen on GlassFish v2 UR2 644
40.4.1. Eseguire il setup in seam-gen 644
40.4.2. Changes needed for deployment to GlassFish 646
41. Dipendenze 653
41.1. Dipendenze JDK 653
41.1.1. Considerazioni su JDK 6 di Sun 653
41.2. Dipendenze del progetto 653
41.2.1. Core 653
41.2.2. RichFaces 654

41.2.3. Seam Mail 655
41.2.4. Seam PDF 655
41.2.5. Seam Microsoft® Excel® 655
41.2.6. Supporto Seam RSS 655
41.2.7. JBoss Rules 656
41.2.8. JBPM 656
41.2.9. GWT 656
41.2.10. Spring 657
41.2.11. Groovy 657
41.3. Gestione delle dipendenze usando Maven 657

Introduzione a JBoss Seam

Seam è un'application framework per Java Enterprise. Si ispira ai seguenti principi:

Unico tipo di "cosa"

Seam definisce un modello uniforme a componenti per tutte le business logic dell'applicazione. Un componente Seam può essere stateful, con uno stato associato ad uno dei tanti contesti ben-definiti, che includono long-running, persistenza, *contesto del processo di business* e il *contesto conversazionale*, che viene preservato lungo le diverse richieste web durante l'interazione dell'utente.

Non c'è alcuna distinzione in Seam tra i componenti del livello presentazione ed i componenti di business logic. Si può stratificare l'applicazione secondo una qualsiasi architettura a proprio piacimento, piuttosto che essere forzati a modellare la logica dell'applicazione in uno schema innaturale con una qualsiasi combinazione di framework aggrovigliati come avviene oggi.

A differenza dei componenti J2EE o del semplice Java EE, i componenti Seam possono *simultaneamente* accedere allo stato associato alla richiesta web e allo stato mantenuto nelle risorse transazionali (senza il bisogno di propagare manualmente lo stato della richiesta web attraverso i parametri). Si potrebbe obbiettare che la stratificazione dell'applicazione imposta dalla vecchia piattaforma J2EE fosse una Cosa Buona. Bene, niente vieta di creare un'architettura a strati equivalente usando Seam — la differenza è che *tu* decidi l'architettura dell'applicazione e decidi quali sono i layer e come lavorano assieme

Integrazione di JSF con EJB 3.0

JSF e EJB 3.0 sono due delle migliori caratteristiche di Java EE 5. EJB3 è nuovo modello a componenti per la logica di business e di persistenza lato server. Mentre JSF è un eccezionale modello a componenti per il livello di presentazione. Sfortunatamente, nessuno dei due componenti da solo è capace di risolvere tutti i problemi. Invece JSE e EJB3 utilizzati assieme funzionano meglio. Ma la specifica Java EE 5 non fornisce alcuno standard per integrare i due modelli a componenti. Fortunatamente, i creatori di entrambi i modelli hanno previsto questa situazione ed hanno elaborato delle estensioni allo standard per consentire un ampiamento ed un'integrazione con altri framework.

Seam unifica i modelli a componenti di JSF e EJB3, eliminando il codice colla, e consentendo allo sviluppatore di pensare al problema di business.

E' possibile scrivere applicazioni Seam dove "qualsiasi cosa" sia un EJB. Questo potrebbe essere sorprendente, se si è abituati a pensare agli EJB come ad oggetti cosiddetti a grana-grossa, "di peso massimo". Comunque la versione 3.0 ha completamente cambiato la natura di EJB dal punto di vista dello sviluppatore. Un EJB è un oggetto a grana-fine — non più complesso di un JavaBean con annotazioni. Seam incoraggia ad usare i session bean come action listener JSF!

Dall'altro lato, se si preferisce non adottare EJB 3.0 adesso, è possibile non farlo. Virtualmente ogni classe java può essere un componente Seam, e Seam fornisce tutte le funzionalità che ci si attende da un "lightweight" container, ed in più, per ogni componente, EJB o altro.

AJAX integrato

Seam supporta le migliori soluzioni open source AJAX basate su JSF: JBoss RichFaces e ICEFaces. Queste soluzioni ti permettono di aggiungere funzionalità AJAX all'interfaccia utente senza il bisogno di scrivere codice JavaScript.

In alternativa Seam fornisce al suo interno uno strato remoto di JavaScript che ti consente di chiamare i componenti in modo asincrono da JavaScript lato client senza il bisogno di uno strato di azione intermedio. Si può anche sottoscrivere topic JMS lato server e ricevere messaggi tramite push AJAX.

Nessuno di questi approcci funzionerebbe bene, se non fosse per la gestione interna di Seam della concorrenza e dello stato, la quale assicura che molte richieste AJAX a grana fine (fine-grained) concorrenti e asincrone vengano gestite in modo sicuro ed efficiente lato server.

Processo di business come primo costrutto di classe

Opzionalmente Seam può fornire una gestione trasparente del processo di business tramite jBPM. Non ci crederai quanto è facile implementare workflow complessi, collaborazioni ed una gestione dei compiti utilizzando jBPM e Seam.

Seam consente pure di definire il pageflow del livello di presentazione utilizzando lo stesso linguaggio (jPDL) che jBPM utilizza per la definizione dei processi di business.

JSF fornisce un ricco ed incredibile modello a eventi per il livello di presentazione. Seam migliora questo modello esponendo gli eventi del processo di business jBPM attraverso lo stesso identico meccanismo di gestione eventi, dando un modello a eventi uniforme per l'intero modello a componenti di Seam.

Gestione dichiarativa dello stato

Siamo tutti abituati al concetto di gestione dichiarativa delle transazioni e sicurezza dichiarativa fin dai primi giorni di EJB. EJB3 introduce anche la gestione dichiarativa del contesto di persistenza. Ci sono tre esempi di un ampio problema di gestione dello stato che è associato ad un particolare *contesto*, mentre tutte i dovuti cleanup avvengono quando il contesto termina. Seam porta oltre il concetto di gestione dichiarativa dello stato e lo applica allo *stato dell'applicazione*. Tradizionalmente le applicazioni J2EE implementano manualmente la gestione dello stato con il get e set della sessione servlet e degli attributi di richiesta. Questo approccio alla gestione dello stato è l'origine di molti bug e problemi di memoria (memory leak) quando l'applicazione non riesce a pulire gli attributi di sessione, o quando i dati di sessione associati a diversi workflow collidono all'interno di un'applicazione multi-finestra. Seam ha il potenziale per eliminare quasi interamente questa classe di bug.

La gestione dichiarativa dello stato dell''applicazione è resa possibile grazie alla ricchezza del *modello di contesto* definito da Seam. Seam estende il modello di contesto definito dalla specifica servlet — richiesta, sessione, applicazione — con due nuovi contesti — conversazione e processo di business — che sono più significatici dal punto di vista della logica di business.

Resterai stupito di come molte cose divengano più semplici non appena inizia ad usare le conversazioni. Hai mai sofferto nell'utilizzo dell'associazione lazy in una soluzione ORM come

Hibernate o JPA? I contesti di Seam di persistenza basati sulle conversazioni ti consentiranno di vedere raramente una `LazyInitializationException`. Hai mai avuto problemi con il pulsante di aggiornamento? Con il pulsante indietro? Con una form inviata due volte? Con la propagazione di messaggi attraverso un post-then-redirect? La gestione delle conversazioni di Seam risolve questi problemi senza che tu debba pensarci. Questi sono tutti sintomi di un'architettura errata di gestione dello stato che è stata prevalente fin dai primi giorni della comparsa del web.

Bijection

La nozione di *Inversione del Controllo* o *dependency injection* esiste in entrambi JSF e EJB3, così come in numerosi così chiamati lightweight container. La maggior parte di questi container predilige l'injection di componenti che implementano *servizi stateless*. Anche quando l'injection di componenti stateful è supportata (come in JSF), è virtualmente inutile per la gestione dello stato dell'applicazione poiché lo scope del componente stateful non può essere definita con sufficiente flessibilità e poiché i componenti appartenenti a scope più ampi potrebbero non essere iniettati nei componenti appartenenti a scope più ristretti.

La *Bijection* differisce da IoC poiché è *dinamica*, *contestuale*, e *bidirezionale*. E' possibile pensare ad essa come un meccanismo per la denominazione di variabili contestuali (nomi in vari contesti legati al thread corrente) in attributi dei componenti. La bijection consente l'autoassemblamento dei componenti da parte del container. Permette pure che un componente possa in tutta sicurezza e semplicità manipolare il valore di una variabile di contesto, solamente assegnandola ad un attributo del componente.

Gestione del workspace e navigazione multi-finestra

Le applicazioni Seam consentono all'utente di passare liberamente a più tab del browser, ciascuno associato ad una conversazione differente ed isolata. Le applicazioni possono addirittura avvantaggiarsi della *gestione del workspace*, consentendo all'utente di spostarsi fra le varie conversazioni (workspace) all'interno del singolo tab del browser. Seam fornisce non solo una corretta funzionalità multi-finestra, ma anche funzionalità multi-finestra in una singola finestra!

Preferenza delle annotazioni all'XML

Tradizionalmente la comunità Java è sempre stata in uno stato di profonda confusione su quali tipologie di meta-informazione debbano essere considerate configurazione. J2EE ed i più noti lightweight container hanno entrambi fornito descrittori per il deploy basati su XML per cose che sono configurabili tra differenti deploy del sistema e per altri tipi di cose o dichiarazioni che non sono facilmente esprimibili in Java. Le annotazioni Java 5 hanno cambiato tutto questo.

EJB 3.0 sceglie le annotazioni e la "configurazione tramite eccezione" come il miglior modo per fornire informazioni al container in una forma dichiarativa. Sfortunatamente JSF è fortemente dipendente da file XML di configurazione molto lunghi. Seam estende le annotazioni fornite da EJB 3.0 con un set di annotazioni per la gestione dichiarativa dello stato e la demarcazione dichiarativa del contesto. Questo consente di eliminare le noiose dichiarazioni JSF dei bean gestiti e riduce l'XML richiesto alla sola informazione che veramente appartiene a XML (le regole di navigazione JSF).

I test d'integrazione sono facili

I componenti Seam, essendo semplici classi Java, sono per natura unità testabili. Ma per le applicazioni complesse, il test dell'unità (testing unit) da solo è insufficiente. Il test d'integrazione (integration testing) è tradizionalmente stato complicato e difficile per le applicazioni web Java. Seam fornisce la testabilità delle applicazioni come caratteristica essenziale del framework. Si potranno facilmente scrivere test JUnit o TestNG che riproducano tutta l'interazione con l'utente, provando tutti i componenti del sistema separati dalla vista (pagina JSP o Facelet). Si potranno eseguire questi test direttamente dentro il proprio IDE, dove Seam automaticamente eseguirà il deploy dei componenti EJB usando JBoss Embedded.

Le specifiche non sono perfette

Pensiamo che l'ultima incarnazione di Java EE sia ottima. Ma sappiamo che non sarà mai perfetta. Dove ci sono dei buchi nella specifica (per esempio limitazioni nel ciclo di vita JSF per le richieste GET), Seam li risolve. E gli autori di Seam stanno lavorando con i gruppi esperti JCP per assicurare che queste soluzioni siano incorporate nelle prossime revisioni degli standard.

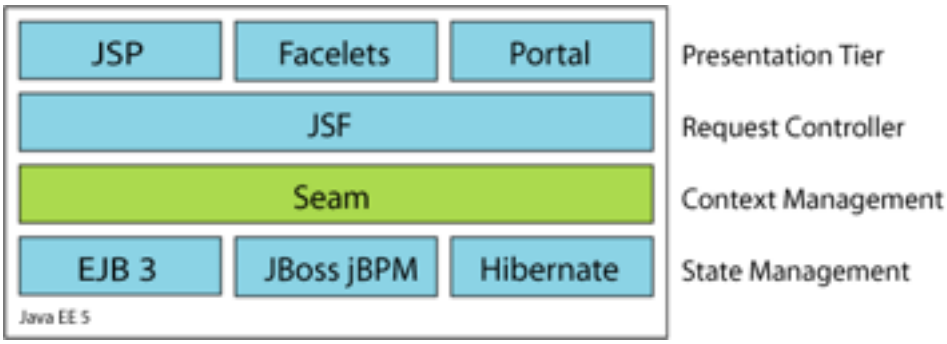
Una web application non genera soltanto pagine html ma fa molto di più

I web framework di oggi pensano troppo poco. Ti consentono di estrarre gli input dell'utente da una form e di metterlo in un oggetto Java. E poi ti abbandonano. Un vero web framework dovrebbe indirizzarsi verso problemi come la persistenza, la concorrenza, l'asincronicità, la gestione dello stato, la sicurezza, le email, la messaggistica, la generazione di PDF e grafici, il workflow, il rendering di wikitext, i web service, il caching e altro ancora. Dopo aver provato Seam, si resterà stupiti di come questi problemi vengano semplificati...

Seam integra JPA e Hibernate3 per la persistenza, EJB Timer Service e Quartz per l'asincronicità, jBPM per i workflow, JBoss Rules per le regole di business, Meldware Mail per le email, Hibernate Search e Lucene per la ricerca full text, JMS per la messaggistica e JBoss Cache per il caching delle pagine. Seam pone un framework di sicurezza basato sulle regole sopra JAAS JBoss Rules. Ci sono anche le librerie JSP per la creazione dei PDF, le mail in uscita, i grafici ed il testo wiki. I componenti Seam possono essere chiamati in modo sincrono come un Web Service, oppure in modo asincrono da JavaScript lato client o da Google Web Toolkit o, sicuramente, direttamente da JSF.

Inizia ora!

Seam funziona in qualsiasi application server Java EE, e perfino in Tomcat. Se il proprio ambiente supporta EJB 3.0, benissimo! Altrimenti, nessun problema, si può utilizzare la gestione delle transazioni interna a Seam con JPA o Hibernate3 per la persistenza. Oppure si può fare il deploy di JBoss Embedded in Tomcat, ed ottenere pieno supporto per EJB 3.0.



La combinazione di Seam, JSF e EJB3 è *il* modo più semplice per scrivere una complessa applicazione web in Java. Ci si stupirà di quanto poco codice viene richiesto!

1. Contribuire a Seam

Visita *SeamFramework.org* [http://www.seamframework.org/Community/Contribute] per scoprire come contribuire a Seam!

Tutorial di Seam

1.1. Utilizzo degli esempi di Seam

Seam fornisce un ampio numero di applicazioni d'esempio per mostrare l'uso delle varie funzionalità di Seam. Questo tutorial ti guiderà attraverso alcuni di questi esempi per aiutarti nell'apprendimento di Seam. Gli esempi di Seam sono posizionati nella sottodirectory `examples` della distribuzione Seam. L'esempio di registrazione, che è il primo esempio che vediamo, si trova nella directory `examples/registration`.

Ciascun esempio ha la medesima struttura di directory:

- La directory `view` contiene i file relativi alla vista come template di pagine web, immagini e fogli di stile.
- La directory `resources` contiene i descrittori per il deploy ed altri file di configurazione.
- La directory `src` contiene il codice sorgente dell'applicazione.

Le applicazioni d'esempio girano sia su JBoss AS sia su Tomcat senza configurazioni aggiuntive. Le sezioni seguenti spiegano la procedura in entrambi i casi. Nota che tutti gli esempi sono costruiti ed eseguiti da `build.xml` di Ant, e quindi ti servirà installata una versione recente di Ant prima di iniziare.

1.1.1. Eseguire gli esempi in JBoss AS

Gli esempi sono configurati per usare JBoss 4.2 o 5.0. Dovrai impostare la variabile `jboss.home` affinché punti alla locazione dell'installazione di JBoss AS. Questa variabile si trova nel file condiviso `build.properties` nella cartella radice dell'installazione di Seam.

Una volta impostata la locazione di JBoss AS ed avviato il server, si può eseguire il build ed il deploy degli esempi semplicemente scrivendo `ant explode` nella directory dell'esempio. Ogni esempio che viene impacchettato come EAR viene messo in un URL del tipo `/seam-`*`example`*, dove *`example`* è il nome della cartella dell'esempio, con una eccezione. Se la cartella d'esempio inizia con seam, il prefisso "seam" viene omesso. Per esempio, se JBoss AS gira sulla porta 8080, l'URL per l'esempio registrazione è *`http://localhost:8080/seam-registration/`* [http://localhost:8080/seam-registration/], mentre l'URL per l'esempio seamspace è *`http://localhost:8080/seam-space/`* [http://localhost:8080/seam-space/].

Se, dall'altro lato, l'esempio viene impacchettato come WAR, allora viene messo in un URL del tipo `/jboss-seam-`*`example`*. La maggior parte degli esempi può essere messa come WAR in Tomcat con JBoss Embedded digitando `ant tomcat.deploy`. Diversi esempi possono venire deployati solo come WAR. Questi esempi sono groovybooking, hibernate, jpa, and spring.

1.1.2. Eseguire gli esempi in Tomcat

Questi esempi sono configurati anche per essere usati in Tomcat 6.0. Occorreràseguire le istruzioni in *Sezione 30.6.1, «Installare JBoss Embedded»* per installare JBoss Embedded in Tomcat 6.0. JBoss Embedded è richiesto per eseguire le demo di Seam che usano componenti EJB3 in Tomcat. Ci sono anche esempio di applicazioni non-EJB3 che possono funzionare in Tomcat senza JBoss Embedded.

Occorrerà impostare al percorso di Tomcat la variabile `tomcat.home`, la quale si trova nel file condiviso `build.properties` della cartella padre nell'installazione di Seam.

Dovrai usare un diverso target Ant per utilizzare Tomcat. Usa `ant tomcat.deploy` nella sotto-directory d'esempio per il build ed il deploy in Tomcat.

Con Tomcat gli esempi vengono deployati con URL del tipo `/jboss-seam-`*`example`*, così per l'esempio di registrazione, l'URL sarebbe *`http://localhost:8080/jboss-seam-registration/`* [http://localhost:8080/jboss-seam-registration/]. Lo stesso vale per gli esempi che vengono deployati come WAR, come già detto nella precedente sezione.

1.1.3. Eseguire i test degli esempi

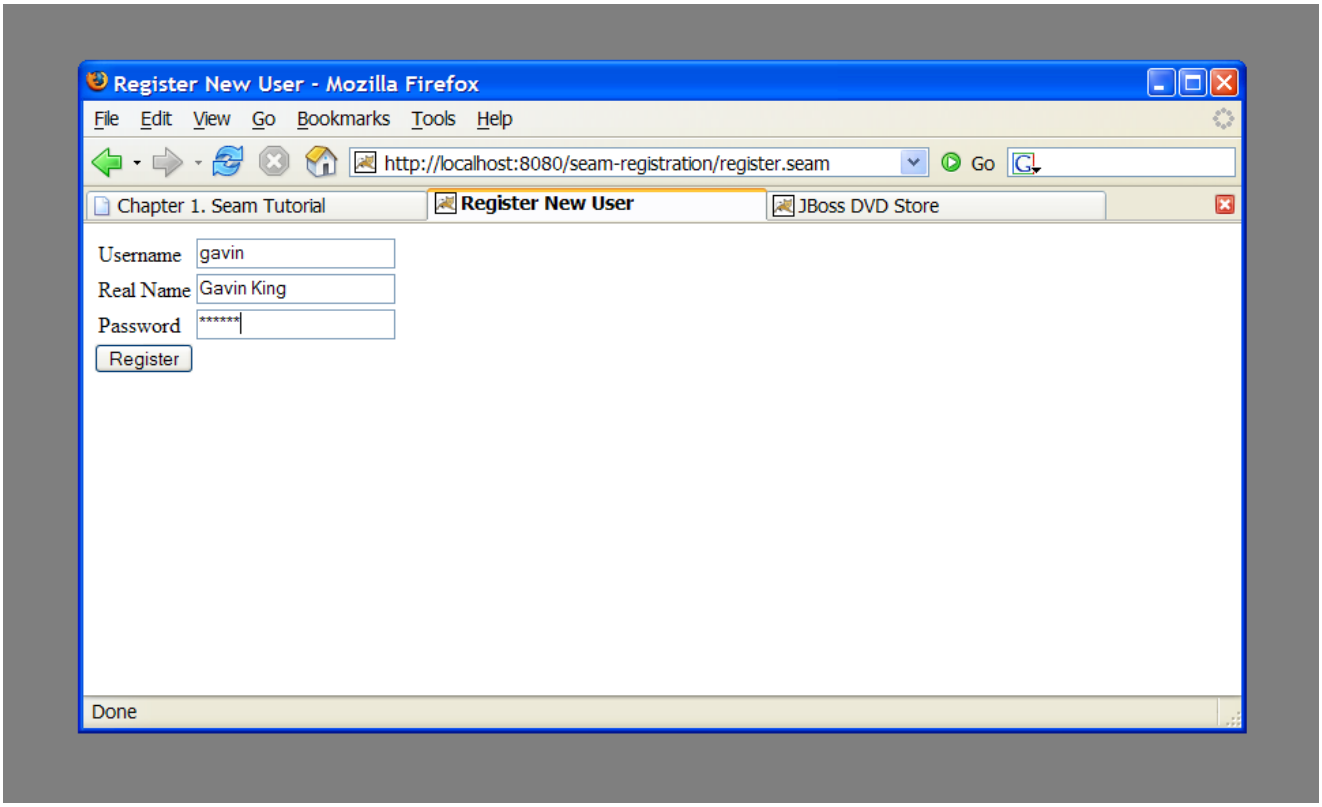
La maggior parte degli esempi è fornita di una suite di test d'integrazione TestNG. Il modo più semplice per eseguire i test è `ant test`. E' anche possibile eseguire i test all'interno del proprio IDE usandi il plugin di TestNG. Per ulteriori informazioni consultare il file readme.txt nella directory degli esempi nella distribuzione di Seam.

1.2. La prima applicazione di Seam: esempio di registrazione

L'esempio di registrazione è una semplice applicazione per consentire all'utente di memorizzare nel database il proprio username, il nome vero e la password. L'esempio non vuole mostrare tutte le funzionalità di Seam. Comunque mostra l'uso di un EJB3 session bean come JSF action listener e la configurazione base di Seam.

Andiamo piano, poiché ci rendiamo conto che EJB 3.0 potrebbe non essere familiare.

La pagina iniziale mostra una form molto semplice con tre campi d'input. Si provi a riempirli e ad inviare la form. Verrà salvato nel database un oggetto user.



1.2.1. Capire il codice

Questo esempio è implementato con due template Facelets, un entity bean e un session bean stateless. Si guardi ora il codice, partendo dal "basso".

1.2.1.1. Entity bean: `User.java`

Occorre un entity bean EJB per i dati utente. Questa classe definisce *persistenza* e *validazione* in modo dichiarativo tramite le annotazioni. Ha bisogno anche di altre annotazioni per definire la classe come componente Seam.

Esempio 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 @Length(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 @Length(min=5, max=15)                         (8)
public String getUsername()
{
  return username;
}

public void setUsername(String username)
```

```
  {
    this.username = username;
  }

}
```

1. L'annotazione EJB3 standard `@Entity` indica che la classe `User` è un entity bean.
2. Un componente Seam ha bisogno di un *nome componente* specificato dall'annotazione *`@Name`* . Questo nome deve essere unico all'interno dell'applicazione Seam. Quando JSF chiede a Seam di risolvere una variabile di contesto son un nome che corrisponde ad un componente Seam, e la variabile di contesto è indefinita (null), Seam istanzia quel componente e lo associa la nuova istanza alla variabile di contesto. In questo caso Seam istanzierà uno `User` la prima volta che JSF incontrerà una variabile chiamata `user`.
3. Quando Seam istanzia un componente, associa la nuova istanza alla variabile di contesto nel *contesto di default* del componente. Il contesto di default viene specificato usando l'annotazione *`@Scope`* . Il bean `User` è un componente con scope di sessione.
4. L'annotazione standard EJB `@Table` indica che la classe `User` è mappata sulla tabella `users`.
5. `name`, `password` e `username` sono gli attributi di persistenza dell'entity bean. Tutti gli attributi di persistenza definiscono i metodi d'accesso. Questi sono necessari quando il componente viene usato da JSF nelle fasi di generazione risposta (render response) e aggiornamento dei valori del modello (update model values).
6. Un costruttore vuoto è richiesto sia dalla specifica EJB sia da Seam.
7. Le annotazioni `@NotNull` e `@Length` sono parte del framework Hibernate Validator. Seam integra Hibernate Validator e consente di usarlo per la validazione dei dati (anche se non viene usato Hibernate per la persistenza).
8. L'annotazione standard EJB `@Id` indica l'attributo di chiave primaria di un entity bean.

Le cose più importanti da notare in quest'esempio sono le annotazioni `@Name` e `@Scope`. Queste annotazioni stabiliscono che questa classe è un componente Seam.

Si vedrà sotto che le proprietà della classe `User` sono legate direttamente ai componenti JSF e sono popolati da JSF durante la fase di aggiornamento dei valori del modello ("update model values"). Non occorre nessun codice colla per copiare i dati avanti ed indietro tra le pagine JSP ed il modello di dominio degli entity bean.

Comunque, gli entity bean non dovrebbero occuparsi della gestione delle transazioni o dell'accesso al database. Quindi non si può usare questo componente come action listener JSF. Per questo occorre un session bean.

1.2.1.2. Classe del bean di sessione stateless: `RegisterAction.java`

La maggior parte delle applicazioni Seam utilizzano session bean come action listener JSF (si possono utilizzare JavaBean se si vuole).

C'è esattamente una azione JSF nell'applicazione ed un metodo di session bean attaccato ad essa. In questo caso si utilizzerà un bean di sessione stateless, poiché tutto lo stato associato all'azione è mantenuto dal bean `User`.

Questo è l'unico codice veramente interessante nell'esempio!

Esempio 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. L'annotazione EJB `@Stateless` marca questa classe come session bean stateless.
2. L'annotazione `@In` marca un attributo del bean come iniettato da Seam. In questo caso, l'attributo viene iniettato da una variabile di contesto chiamata `user` (il nome della variabile istanza).
3. L'annotazione EJB standard `@PersistenceContext` è usata per iniettare l'entity manager EJB3.
4. L'annotazione `@Logger` di Seam è usata per iniettare l'istanza `Log` del componente.
5. Il metodo action listener utilizza l'API EJB3 standard `EntityManager` per interagire con il database, e restituisce l'esito JSF. Notare che, poiché questo è un session bean, si inizia automaticamente una transazione quando viene chiamato il metodo `register()`, e viene eseguito il commit quando questo completa.
6. Si noti che Seam consente di utilizzare espressioni JSF EL dentro EJB-QL. Sotto il coperchio, questo proviene da un'ordinaria chiamata JPA `setParameter()` sull'oggetto standard `Query`. Interessante, vero?
7. L'API `Log` consente facilmente di mostrare i messaggi di log, i quali possono anche impiegare espressioni JSF EL.
8. I metodi JSF action listener restituiscono un esito di tipo stringa, che determina quale pagina verrà mostrata come successiva. Un estio null (o un metodo action listener di tipo void) regenera la pagina precedente. Nel semplice JSF, è normale impiegare sempre una *regola di navigazione* JSF per determinare l'id della vista JSF dall'esito. Per applicazioni complesse quest'azione indiretta (indirection) è sia utile sia una buona pratica. Comunque, per ogni esempio semplice come questo, Seam consente di usare l'id della vista JSF come esito, eliminando l'uso della regola di navigazione. *Si noti che quando viene usato l'id della vista come esito, Seam esegue sempre un redirect del browser.*
9. Seam fornisce un numero di *componenti predefiniti* per aiutare a risolvere problemi comuni. Il componente `FacesMessages` agevola la visualizzazione di messaggi template di errore o di successo. (Da Seam 2.1, si può impiegare `StatusMessages` invece di rimuovere la dipendenza semantica di JSF.) I componenti Seam predefiniti possono essere ottenuti tramite iniezione, o chiamando il metodo `instance()` sulla classe del componente predefinito.

Si noti che questa volta non si è esplicitamente specificato uno `@Scope`. Ciascun tipo di componente Seam ha uno scope di default se non esplicitamente specificato. Per bean di sessione stateless, lo scope di default è nel contesto stateless, che è l'unico valore sensato.

L'action listenere del bean di sessioni esegue la logica di persistenza e di business per quest'applicazione. In applicazioni più complesse, può essere opportuno separare il layer di servizio. Questo è facile da farsi in Seam, ma è critico per la maggior parte delle applicazioni web. Seam non forza nell'impiego di una particolare strategia per il layering dell'applicazione, consentendo di rimanere semplici o complessi a proprio piacimento.

Si noti che in questa semplice applicazione, abbiamo reso le cose di gran lunga più complicate di quanto necessario. Se si fossero impiegati i controllori di Seam, si sarebbe eliminato molto codice dell'applicazione. Comunque non avremmo avuto molto da spiegare.

1.2.1.3. Interfaccia locale del session bean: `Register.java`

Certamente il nostro session bean richiede un'interfaccia locale.

Esempio 1.3. Register.java

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

Questa è la fine del codice Java. Vediamo ora la vista.

1.2.1.4. La vista: `register.xhtml` e `registered.xhtml`

Le pagine di vista di per un'applicazione Seam possono essere implementate usando qualsiasi tecnologia supporti JSF. In quest'esempio si usa Facelets, poiché noi pensiamo sia migliore di JSP.

Esempio 1.4. register.xhtml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:s="http://jboss.com/products/seam/taglib"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <title
>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <s:validateAll>
          <h:panelGrid columns="2">
            Username: <h:inputText value="#{user.username}" required="true"/>
```

```
            Real Name: <h:inputText value="#{user.name}" required="true"/>
            Password: <h:inputSecret value="#{user.password}" required="true"/>
          </h:panelGrid>
        </s:validateAll>
        <h:messages/>
        <h:commandButton value="Register" action="#{register.register}"/>
      </h:form>
    </f:view>
  </body>

</html
>
```

L'unica cosa che qua è specifica di Seam è il tag `<s:validateAll>`. Questo componente JSF dice a JSFdi validare tutti i campi d'input contenuti con le annotazioni di Hibernate Validator specificate nell'entity bean.

Esempio 1.5. registered.xhtml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:f="http://java.sun.com/jsf/core">

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

</html>
```

Questa è una semplice pagina JSF che utilizza EL. Qua non c'è niente di specifico di Seam.

1.2.1.5. Il descrittore di deploy dei componenti Seam: `components.xml`

"Poiché questa è la prima applicazione vista, si prenderanno in esame i descrittori di deploy. Ma prima di iniziare, vale la pena di notare che Seam apprezza molto una configurazione minimale.

Questi file di configurazione verranno creati al momento della creazione di un'applicazione Seam. Non sarà mai necessario metter mano alla maggior parte di questi file. Qua vengono presentati solo per aiutare a capire tutti pezzi dell'esempio preso in considerazione.

Se in precedenza si sono utilizzati altri framework Java, si è abituati a dichiarare le classi componenti in un qualche file XML che gradualmente cresce sempre più e diventa sempre più ingestibile man mano che il progetto evolve. Si resterà sollevati dal sapere che Seam non richiede che i componenti dell'applicazione siano accompagnati da file XML. La maggior parte delle applicazioni Seam richiede una quantità molto piccola di XML che non aumenta man mano che il progetto cresce.

Tuttavia è spesso utile fornire una *qualche* configurazione esterna per *qualche* componente (particolarmente per i componenti predefiniti di Seam). Ci sono due opzioni, ma l'opzione più flessibile è fornire questa configurazione in un file chiamato `components.xml`, collocato nella directory `WEB-INF`. Si userà il file `components.xml` per dire a Seam dove trovare i componenti EJB in JNDI:

Esempio 1.6. 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://jboss.com/products/seam/core
      http://jboss.com/products/seam/core-2.1.xsd
      http://jboss.com/products/seam/components
      http://jboss.com/products/seam/components-2.1.xsd">

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

</components
>
```

Questo codice configura una proprietà chiamata `jndiPattern` di un componente Seam predefinito chiamato `org.jboss.seam.core.init`. Il divertente simbolo `@` viene impiegato poiché lo script di build Ant vi mette al suo posto il corretto JDNI pattern al momento del deploy dell'applicazione, ricavato dal file components.properties. Maggiori informazioni su questo processo in *Sezione 5.2, «Configurazione dei componenti tramite `components.xml`»*.

1.2.1.6. La descrizione del deploy web: `web.xml`

Il layer di presentazione dell'applicazione verrà deployato in un WAR. Quindi sarà necessario un descrittore di deploy web.

Esempio 1.7. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">

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

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

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

   <servlet-mapping>
      <servlet-name
>Faces Servlet</servlet-name>
      <url-pattern
>*.seam</url-pattern>
   </servlet-mapping>

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

```
</web-app
>
```

Il file `web.xml` configura Seam e JSF. La configurazione vista qua è più o meno la stessa in tutte le applicazioni Seam.

1.2.1.7. Configurazione JSF: `faces-config.xml`

La maggior parte delle applicazioni Seam utilizza le viste JSF come layer di presentazione. Così solitamente si avrà bisogno di `faces-config.xml`. In ogni caso noi utilizzeremo Facelets per definire le nostre viste, così avremo bisogno di dire a JSF di usare Facelets come suo motore di template

Esempio 1.8. faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">

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

</faces-config
>
```

Si noti che non occorre alcuna dichiarazione di managed bean JSF! I managed bean sono componenti Seam annotati. Nelle applicazioni Seam, `faces-config.xml` è usato meno spesso che nel semplice JSF. Qua, viene usato per abilitare Faceltes come gestore di viste al posto di JSP.

Infatti una volta configurati tutti i descrittori base, l'*unico* XML necessario da scrivere per aggiungere nuove funzionalità ad un'applicazione Seam è quello per l'orchestrazione (orchestration): regole di navigazione o definizione di processi jBPM. Un punto fermo di Seam è che *flusso di processo* e *configurazione dei dati* siano le uniche cose che veramente appartengano alla sfera dell'XML.

Questo semplice esempio non è neppure stato necessario usare una regola di navigazione, poiché si è deciso di incorporare l'id della vista nel codice dell'azione.

1.2.1.8. Descrittore di deploy EJB: `ejb-jar.xml`

Il file `ejb-jar.xml` integra Seam con EJB3, attaccando `SeamInterceptor` a tutti i bean di sessione nell'archivio.

```
<?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. Descrittore di deploy per la persistenza EJB: `persistence.xml`

Il file `persistence.xml` dice all'EJB persistence provider dove trovare il datasource, e contiene alcune impostazioni vendor-specific. In questo caso abilita automaticamente l'esportazione dello schema all'avvio.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/persistence
```

```
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">

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

</persistence
>
```

1.2.1.10. Descrittore di deploy EAR: `application.xml`

Infine poiché l'applicazione viene deployata come EAR, occorre anche un descrittore di deploy.

Esempio 1.9. applicazione di registrazione

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/application_5.xsd"
   version="5">

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

   <module>
      <web>
         <web-uri
>jboss-seam-registration.war</web-uri>
         <context-root
>/seam-registration</context-root>
      </web>
   </module>
   <module>
      <ejb
```

```
>jboss-seam-registration.jar</ejb>
   </module>
   <module>
      <ejb
>jboss-seam.jar</ejb>
   </module>
   <module>
      <java
>jboss-el.jar</java>
   </module>

</application
>
```

Questo descrittore di deploy punta a moduli nell'archivio enterprise ed associa l'applicazione web al contesto radice `/seam-registration`.

Adesso sono stati analizzati tutti i file dell'intera applicazione!

1.2.2. Come funziona

Quando la form viene inviata, JSF chiede a Seam di risolvere la variabile chiamata `user`. Poiché non c'è alcun valore associato a questo nome (in un qualsiasi contesto Seam), Seam istanzia il componente `user` e restituisce a JSF un'istanza dell'entity bean `User` dopo averla memorizzata nel contesto Seam di sessione.

I valori di input della form vengono ora validati dai vincoli di Hibernate Validator specificati nell'entity `User`. Se i vincoli vengono violati, JSF rivisualizza la pagina, Altrimenti, JSF associa i valori di input alle proprietà dell'entity bean `User`.

Successivamente JSF chiede a Seam di risolvere la variabile chiamata `register`. Seam utilizza il pattern JNDI menzionato in precedenza per localizzare il session bean stateless, lo impiega come componente Seam tramite il wrap e lo restituisce. Seam quindi presenta questo componente a JSF e JSF invoca il metodo action listener `register()`.

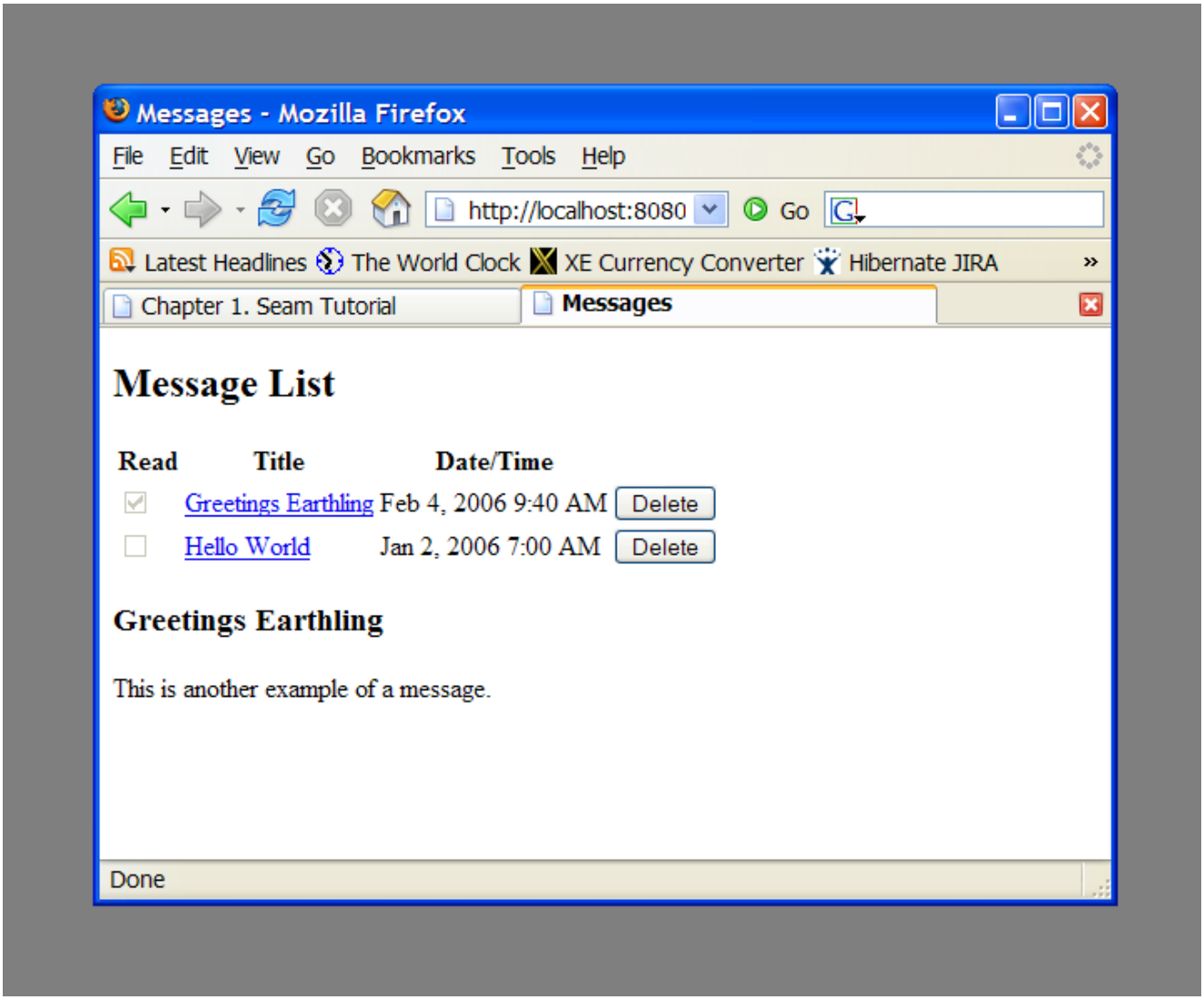
Ma Seam non ha ancora terminato. Seam intercetta la chiamata al metodo e inietta l'entity `User` dal contestosessione di Seam, prima di consentire all'invocazione di continuare.

Il metodo `register()` controlla se esiste già un utente lo username inserito. Se è così, viene accodato un errore al componente `FacesMessages`, e viene restituito un esito null, causando la rivisualizzazione della pagina. Il componente `FacesMessages` interpola l'espressione JSF incorporata nella stringadi messaggio e aggiunge un `FacesMessage` JSF alla vista.

Se non esiste nessun utente con tale username, l'esito di `"/registered.xhtml"` causa un redirect del browser verso la pagina `registered.xhtml`. Quando JSF arriva a generare la pagina, chiede a Seam di risolvere la variabile chiamata `user` ed utilizza il valori di proprietà dell'entity `User` restituito dallo scope di sessione di Seam.

1.3. Liste cliccabili in Seam: esempio di messaggi

Le liste cliccabili dei risultati di ricerca del database sono una parte così importante di qualsiasi applicazione online che Seam fornisce una funzionalità speciale in cima a JSF per rendere più facile l'interrogazione dei dati usando EJB-QL o HQL e la mostra comelista cliccabile usando il JSF `<h:dataTable>`. I messaggi d'esempio mostrano questa funzionalità.



1.3.1. Capire il codice

L'esempio di lista messaggi ha un entity bean, `Message`, un session bean, `MessageListBean` ed una JSP.

1.3.1.1. Entity bean: `Message.java`

L'entity `Message` definisce il titolo, il testo, la data e l'orario del messaggio e un flag indica se il messaggio è stato letto:

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

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

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

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

}
```

1.3.1.2. Il bean di sessione stateful: `MessageManagerBean.java`

Come nel precedente esempio, esiste un session bean, `MessageManagerBean`, che definisce i metodi di action listener per i due bottoni della form. Uno di questi seleziona un messaggio dalla lista, e mostra tale messaggio. L'altro cancella il messaggio. Finora non è molto diverso dal precedente esempio.

Ma `MessageManagerBean` è anche responsabile per il recupero della lista dei messaggi la prima volta che si naviga nella pagina della lista messaggi. Ci sono vari modi in cui l'utente può navigare nella pagina, e non tutti sono preceduti da un'azione JSF — l'utente può avere un memorizzato la pagina, per esempio. Quindi il compito di recuperare la lista messaggi avviene in un *metodo factory* di Seam, invece che in un metodo action listener.

Si vuole memorizzare la lista dei messaggi tra le varie richieste server, e quindi questo session bean diventerà stateful.

Esempio 1.11. MessageManagerBean.java

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
```

```
public class MessageManagerBean implements Serializable, MessageManager
{
  @DataModel
  private List<Message
> messageList;

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

  @PersistenceContext(type=EXTENDED)
  private EntityManager em;

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

  public void select()
  {
    message.setRead(true);
  }

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

  @Remove
  public void destroy() {}

}
```

1. L'annotazione `@DataModel` espone alla pagina JSF un attributo di tipo `java.util.List` come istanza di `javax.faces.model.DataModel`. Questo permette di usare la lista in un `<h:dataTable>` di JSF con link cliccabili per ogni riga. In questo caso il `DataModel` è reso disponibile in una variabile con contesto sessione chiamata `messageList`.
2. L'annotazione `@DataModelSelection` dice a Seam di iniettare l'elemento `List` che corrisponde al link cliccato.
3. L'annotazione `@Out` espone direttamente alla pagina il valore selezionato. Ogni volta che una riga della lista viene selezionata, il `Message` viene iniettato nell'attributo del bean stateful, e in seguito viene fatta *l'outjection* nella variabile con contesto evento chiamata `message`.
4. Questo bean stateful ha un *contesto di persistenza EJB3 esteso*. I messaggi recuperati nella query rimangono nello stato gestito finché esiste il bean, quindi ogni chiamata di metodo conseguente al bean può aggiornarli senza il bisogno di chiamare esplicitamente l'`EntityManager`.
5. La prima volta che si naviga in un pagina JSP, non c'è alcun valore nella variabile di contesto `messageList`. L'annotazione `@Factory` dice a Seam di creare un'istanza di `MessageManagerBean` e di invocare il metodo `findMessages()` per inizializzare il valore. `findMessages()` viene chiamato *metodo factory* di `messages`.
6. Il metodo action listener `select()` marca il `Message` selezionato come letto e lo aggiorna nel database.
7. Il metodo action listener `delete()` rimuove il `Message` dal database.
8. Tutti i componenti Seam bean di sessione stateful *devono* avere un metodo senza parametri marcato `@Remove` che Seam utilizza per rimuovere il bean stateful quando termina il contesto di Seam, e viene pulito tutto lo stato lato server.

Si noti che questo è un componente Seam di sessione. E' associato alla sessione di login dell'utente e tutte le richieste da una login di sessione condividono la stessa istanza del componente. (Nelle applicazioni Seam, solitamente si usano componenti con scope di sessione in maniera contenuta.)

1.3.1.3. L'interfaccia locale del session bean local: `MessageManager.java`

Tutti i session bean hanno un'interfaccia di business, naturalmente.

Esempio 1.12. MessageManager.java

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

D'ora in poi non verranno mostrate interfacce locali nei codici d'esempio.

Saltiamo i file `components.xml`, `persistence.xml`, `web.xml`, `ejb-jar.xml`, `faces-config.xml` e `application.xml` poiché sono praticamente uguali all'esempio precedente e andiamo dritti alla pagina JSP.

1.3.1.4. La vista: `messages.jsp`

La pagina JSP è un semplice utilizzo del componente JSF `<h:dataTable>`. Ancora nulla di specifico di Seam.

Esempio 1.13. messages.jsp

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

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

1.3.2. Come funziona

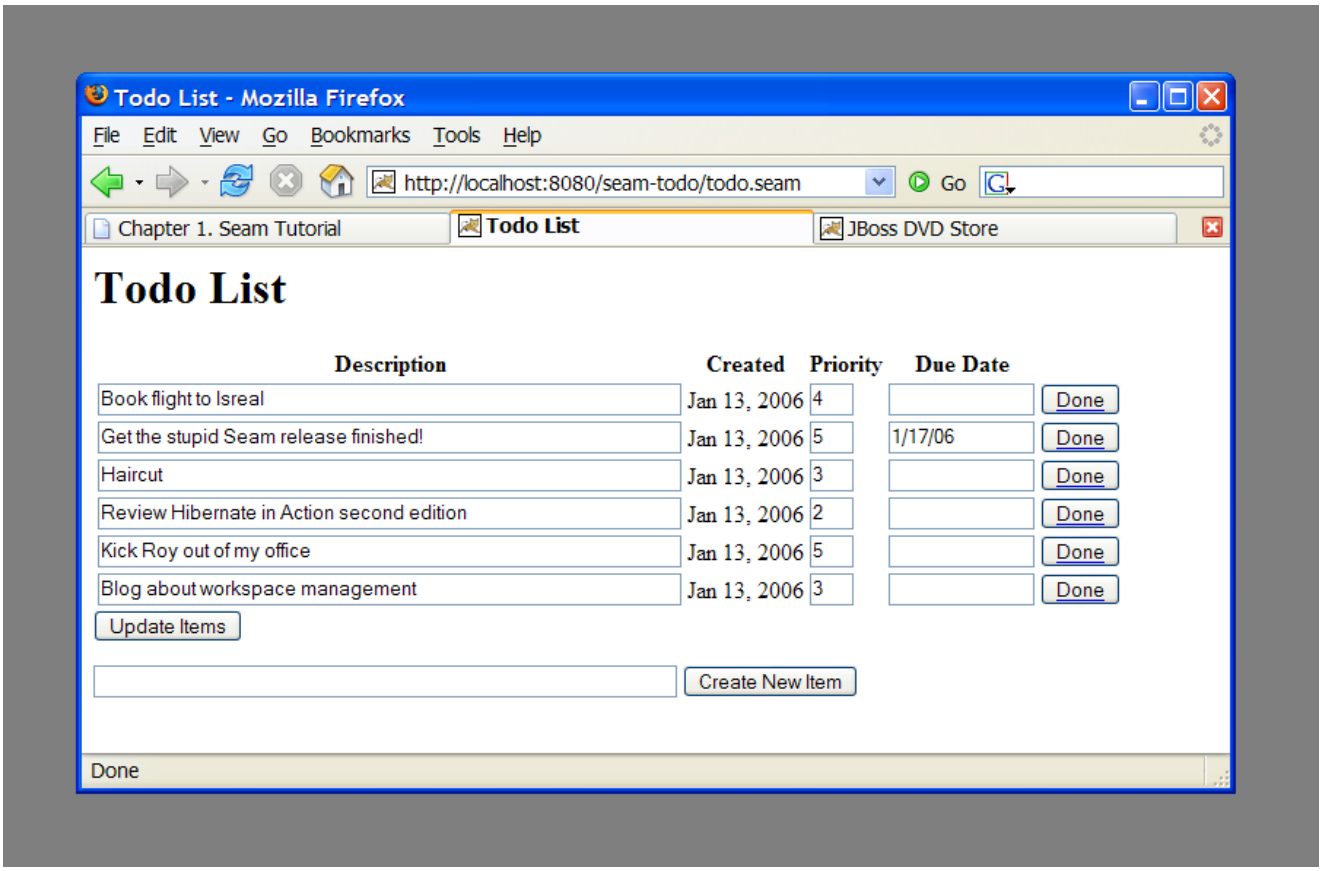
La prima volta che si naviga nella pagina `messages.jsp`, la pagina proverà a risolvere la variabile di contesto `messageList`. Poiché questa variabile non è inizializzata, Seam chiamerà il metodo factory `findMessages()`, che esegue la query del database e mette i risultati in un `DataModel` di cui verrà fatta l'outjection. Questo `DataModel` fornisce i dati di riga necessari per generare la `<h:dataTable>`.

Quando l'utente clicca il `<h:commandLink>`, JSF chiama l'action listener `select()`. Seam intercetta questa chiamata ed inietta i dati di riga selezionati nell'attributo del componente `messageManager`. L'action listener viene eseguito, marcando come letto il `Message` selezionato. Alla fine della chiamata, Seam esegue l'outjection del `Message` selezionato nella variabile di contesto chiamata `message`. Poi il container EJB committa la transazione ed i cambiamenti a `message` vengono comunicati al database. Infine la pagina vienere rigenerata, rimostrando la lista dei messaggi e mostrando sotto il messaggio selezionato.

Se l'utente clicca `<h:commandButton>`, JSF chiama l'action listener `delete()`. Seam intercetta questa chiamata ed inietta i dati selezionati nell'attributo `message` del componente `messageList`. L'action listener viene eseguito, rimuovendo dalla lista il `Message`, e chiamando anche il metodo `remove()` dell'`EntityManager`. Alla fine della chiamata, Seam aggiorna la variabile di contesto `messageList` e pulisce la variabile di contesto chiamata `message`. Il container EJB committa la transazione e cancella `Message` dal database. Infine la pagina viene rigenerata, rimostrando la lista dei messaggi.

1.4. Seam e jBPM: esempio di lista todo

jBPM fornisce una funzionalità sofisticata per il workflow e la gestione dei task. Per provare come jBPM si integra con Seam, viene mostrata l'applicazione "todo list". Poiché gestire liste di task è la funzione base di jBPM, non c'è praticamente alcun codice Java in quest'esempio.



1.4.1. Capire il codice

La parte centrale dell'esempio è la definizione del processo jBPM. Ci sono anche due pagine JSP e due banalissimi JavaBean (Non c'è alcuna ragione per usare session bean, poiché questi non accedono al database, e non hanno un comportamento transazionale). Cominciamo con la definizione del processo:

Esempio 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. Il nodo `<start-state>` rappresenta l'inizio logico del processo. Quando il processo inizia, transita subito nel nodo `todo`.
2. Il nodo `<task-node>` rappresenta uno *stato di attesa*, dove l'esecuzione del processo di business va in pausa, ed aspetta che uno o più task vengano eseguiti.
3. L'elemento `<task>` definisce un task che deve essere eseguito da un utente. Poiché c'è solo un task definito su questo nodo, quando completa, l'esecuzione riprende e si transita verso lo stato finale. Il task recupera la sua descrizione da un componente Seam chiamato `todoList` (uno dei JavaBean).
4. I task devono essere assegnati ad un utente od un gruppo di utenti quando vengono creati. In questo caso il task viene assegnato all'utente corrente che viene recuperato dal componente Seam predefinito chiamato `actor`. Ogni componente Seam può essere impiegato per assegnare un task.
5. Il nodo `<end-state>` definisce la fine logica del processo di business. Quando l'esecuzione raggiunge questo nodo, l'istanza di processo viene distrutta.

Se viene impiegato l'editor per le definizioni di processo fornito da JBossIDE, questa apparirà così:

Questo documento definisce il *processo di business* come un grafo di nodi. Questo è un processo di business molto banale: c'è un *task* da eseguire e quando questo viene completato, il processo termina.

Il primo javaBean gestisce la pagina `login.jsp`. Il suo compito è quello di inizializzare l'id actor jBPM usando il componente `actor`. Nelle applicazioni occorrerà autenticare l'utente.

Esempio 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.jsp";
  }
}
```

Qua si vede l'uso di `@In` per iniettare il componente predefinito `Actor`.

Lo stesso JSP è banale:

Esempio 1.16. login.jsp

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

Il secondo JavaBean è responsabile per l'avvio delle istanze del processo di business e della fine dei task.

Esempio 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. La proprietà descrizione accetta l'input utente dalla pagina JSP e lo espone alla definizione del processo, consentendo che venga impostata la descrizione del task.
2. L'annotazione Seam `@CreateProcess` crea una nuova istanza di processo jBPM dalla definizione del processo.
3. L'annotazione Seam `@StartTask` avvia un task. `@EndTask` termina il task, e consente di ripristinare l'esecuzione del processo di business.

In un esempio più realistico `@StartTask` e `@EndTask` non apparirebbero nello stesso metodo, poiché solitamente c'è del lavoro da fare in un'applicazione prima che il task venga terminato.

Infine, il cuore dell'applicazione è in `todo.jsp`:

Esempio 1.18. todo.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

```
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
<head>
<title
>Todo List</title>
</head>
<body>
<h1
>Todo List</h1>
<f:view>
  <h:form id="list">
    <div>
      <h:outputText value="There are no todo items."
                    rendered="#{empty taskInstanceList}"/>
      <h:dataTable value="#{taskInstanceList}" var="task"
                   rendered="#{not empty taskInstanceList}">
        <h:column>
          <f:facet name="header">
            <h:outputText value="Description"/>
          </f:facet>
          <h:inputText value="#{task.description}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Created"/>
          </f:facet>
          <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
            <f:convertDateTime type="date"/>
          </h:outputText>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Priority"/>
          </f:facet>
          <h:inputText value="#{task.priority}" style="width: 30"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Due Date"/>
          </f:facet>
          <h:inputText value="#{task.dueDate}" style="width: 100">
            <f:convertDateTime type="date" dateStyle="short"/>
          </h:inputText>
        </h:column>
```

```
          <h:column>
             <s:button value="Done" action="#{todoList.done}" taskInstance="#{task}"/>
          </h:column>
        </h:dataTable>
      </div>
      <div>
      <h:messages/>
      </div>
      <div>
        <h:commandButton value="Update Items" action="update"/>
      </div>
    </h:form>
    <h:form id="new">
      <div>
        <h:inputText value="#{todoList.description}"/>
        <h:commandButton value="Create New Item" action="#{todoList.createTodo}"/>
      </div>
    </h:form>
</f:view>
</body>
</html
>
```

Si prenda un pezzo alla volta.

La pagina renderizza una lista di task prelevati da un componente di Seam chiamato `taskInstanceList`. La lista è definita dentro una form JSF.

Esempio 1.19. todo.jsp

```
<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
>
```

Ciascun elemento della lista è un'istanza della classe jBPM `TaskInstance`. Il codice seguente mostra semplicemente le proprietà di interesse per ogni task della lista. Per consentire all'utente di aggiornare i valori di descrizione, priorità e data di ultimazione, si usano i controlli d'input.

```
<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
>
```

Nota

Seam fornisce di default un converter JSF di date per convertire una stringa in una data (no tempo). Quindi , il converter non è necessario per un'associazione di campo a `#{task.dueDate}`.

Questo pulsante termina il task chiamando il metodo d'azione annotato con `@StartTask @EndTask`. Inoltre passa l'id del task come parametro di richiesta a Seam.

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

Si noti che questo sta usando un controllo JSF Seam `<s:button>` del pacchetto `seam-ui.jar`. Questo pulsante è usato per aggiornare le proprietà dei task. Quando la form viene aggiornata, Seam e jBPM renderanno persistenti i cambiamenti ai task. Non c'è bisogno di alcun metodo action listener:

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

Viene usata una seconda form per creare nuovi item, chiamando il metodo d'azione annotato con `@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. Come funziona

Dopo la login, todo.jsp utilizza il componente `taskInstanceList` per mostrare un tabella con i compiti da eseguire da parte dell'utente corrente. Inizialmente non ce ne sono. Viene presentata anche una form per l'inserimento di una nuova voce. Quando l'utente digita il compito da eseguire e preme il pulsante "Create New Item", viene chiamato `#{todoList.createTodo}`. Questo inizia il processo todo, così come definito in `todo.jpdl.xml`.

L'istanza di processo viene creata a partire dallo stato di start ed immediatamente viene eseguita una transizione allo stato `todo`, dove viene creato un nuovo task. La descrizione del task viene impostata in base all'input dell'utente, che è stato memorizzato in `#{todoList.description}`. Poi il task viene assegnato all'utente corrente, memorizzato nel componente Seam chiamato actor. Si noti che in quest'esempio il processo non ha ulteriori stati di processo. Tutti gli stati sono memorizzati nella definizione del task. Il processo e le informazioni sul task sono memorizzati nel database alla fine della richiesta.

Quando `todo.jsp` viene rivisualizzata, `taskInstanceList` trova il task appena creato. Il task viene mostrato in un `h:dataTable`. Lo stato interno del task è mostrato in ciascuna colonna:

`#{task.description}`, `#{task.priority}`, `#{task.dueDate}`, ecc... Questi campi possono essere tutti editati e salvati nel database.

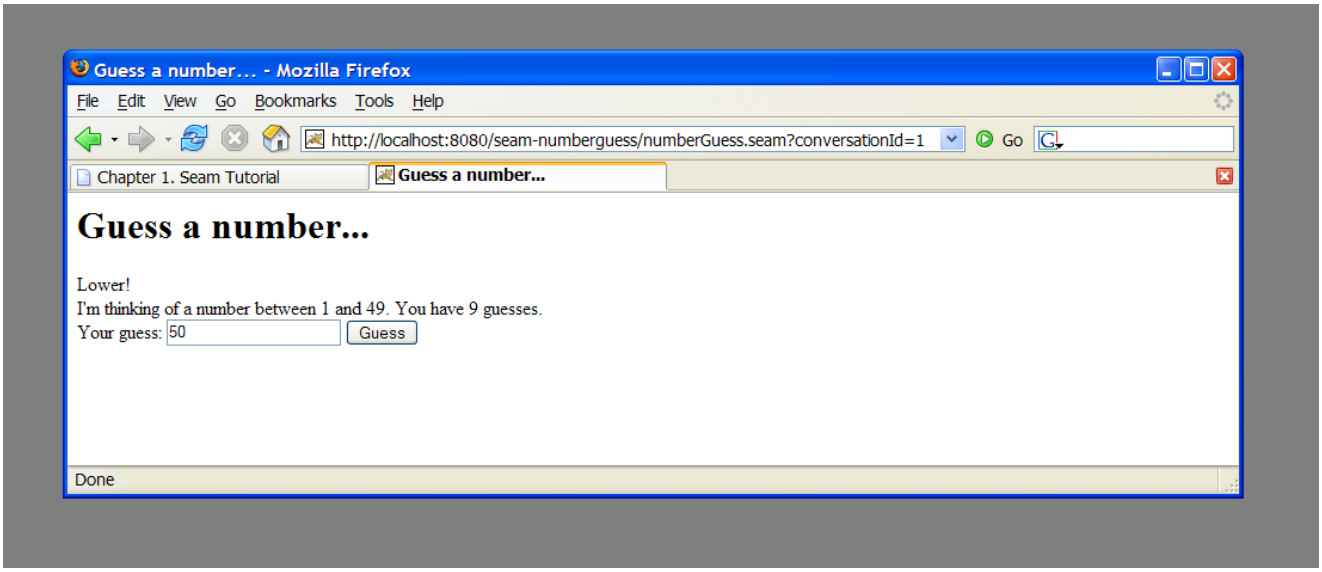
Ogni elemento todo ha anche un pulsante "Done", che chiama `#{todoList.done}`. Il componente `todoList` sa a quale task si riferisce il pulsante, poiché ogni s:button specifica `taskInstance="#{task}"`, che si riferisce al task per quella particolare linea della tabella. Le annotazioni `@StartTast` e `@EndTask` obbligano seam a rendere attivo il task e a completarlo. Il processo originale quindi transita verso lo stato `done`, secondo la definizione del processo, dove poi termina. Lo stato del task e del processo sono entrambi aggiornati nel database.

Quando `todo.jsp` viene di nuovo visualizzata, il task adesso completato non viene più mostrato in `taskInstanceList`, poiché questo componente mostra solo i task attivi per l'utente.

1.5. Seam pageflow: esempio di indovina-numero

Per le applicazioni Seam con una navigazione relativamente libera, le regole di navigazione JSF/Seam sono un modo perfetto per definire il flusso di pagine. Per applicazioni con uno stile di navigazione più vincolato, specialmente per interfacce utente più stateful, le regole di navigazione rendono difficile capire il flusso del sistema. Per capire il flusso occorre mettere assieme le pagine, le azioni e le regole di navigazione.

Seam consente di usare la definizione di processo con jPDL per definire il flusso di pagine. L'esempio indovina-numero mostra come fare.



1.5.1. Capire il codice

Quest'esempio è implementato usando un JavaBean, tre pagine JSP ed una definizione di pageflow jPDL. Iniziamo con il pageflow:

Esempio 1.20. pageflow.jpdl.xml

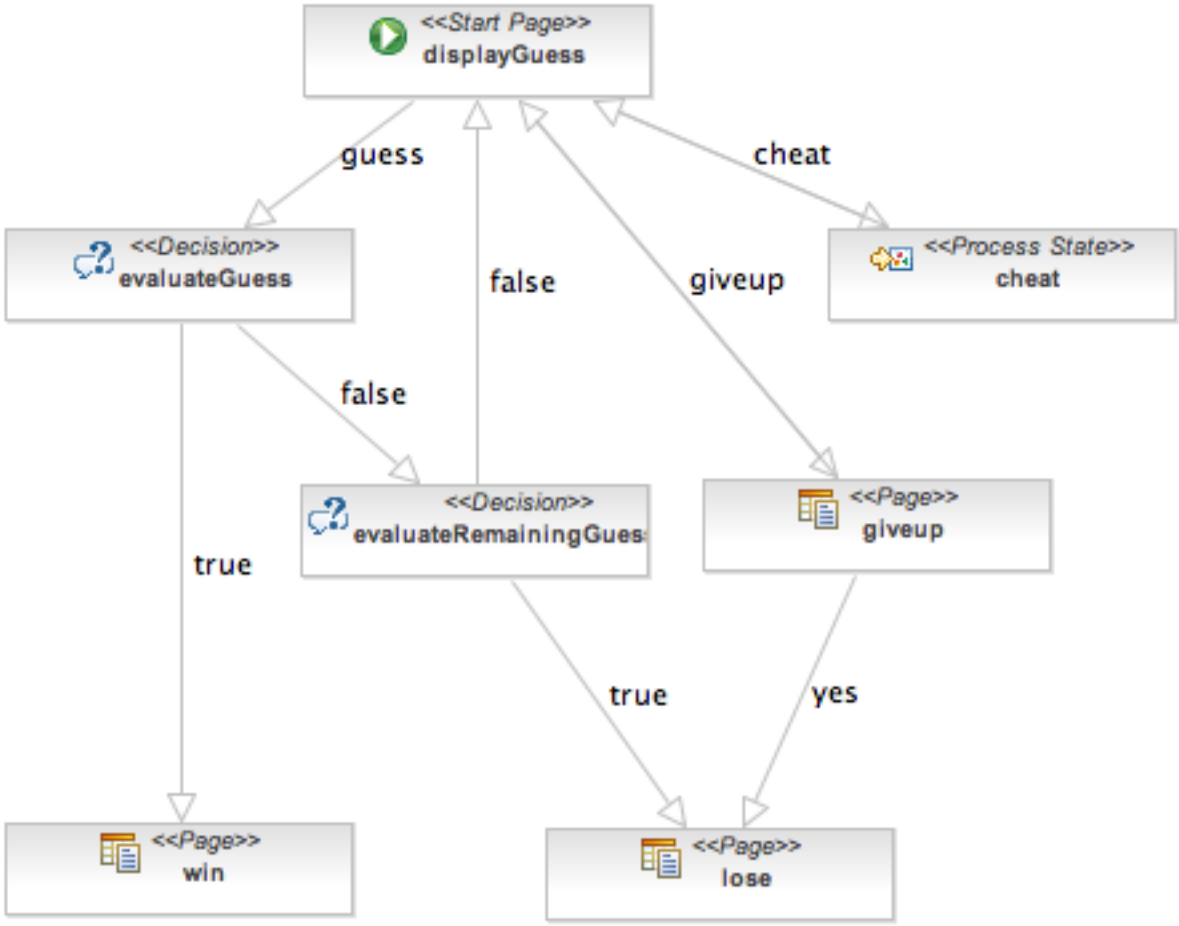
```
<<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:s="http://jboss.com/products/seam/taglib"
        xmlns="http://www.w3.org/1999/xhtml"
        version="2.0">

  <jsp:output doctype-root-element="html"                                    1
           doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"

           doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-transitio 2 nal.dtd"/>

  <jsp:directive.page contentType="text/html"/>                              3
  <html>
  <head>
    <title
>Guess a number...</title>

    <link href="niceforms.css" rel="stylesheet" type="text/css" />           4
    <script language="javascript" type="text/javascript" src="niceforms.js" />
  </head>
  <body>
    <h1
>Guess a number...</h1>
    <f:view>
      <h:form styleClass="niceform">

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

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

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

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

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

1. L'elemento `<page>` definisce uno stato di attesa dove il sistema mostra una particolare vista JSF ed attende input da parte dell'utente. `view-id` è lo stesso id view usato nelle regole di navigazione nel pure JSF. L'attributo `redirect` dice a Seam di usare il post-then-redirect quando si passa ad un'altra pagina. (Questo capita con gli URL dei browser.)
2. L'elemento `<transition>` chiama un esito JSF. La transizione è lanciata quando un'azione JSF ha tale esito. L'esecuzione quindi procederà verso il successivo nodo del grafo pageflow, dopo l'invocazione di una qualsiasi azione di transizione jBPM.
3. Una transizione `<action>` è come un'azione JSF, tranne che avviene quando si verifica una transizione jBPM. L'azione di transizione può invocare qualsiasi componente Seam.

4 Un nodo `<decision>` divide il pageflow e determina il successivo nodo da eseguire valutando un'espressione JSF EL.

Ecco come appare il pageflow nell'editor di pageflow di JBoss Developer Studio:



Ora che abbiamo visto il pageflow, è molto facile capire il resto dell'applicazione.

Ecco la pagina principale dell'applicazione, `numberGuess.jspx`:

Esempio 1.21. numberGuess.jspx

```
<<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:s="http://jboss.com/products/seam/taglib"
        xmlns="http://www.w3.org/1999/xhtml"
        version="2.0">
  <jsp:output doctype-root-element="html"
```

```
          doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
          doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
  <html>
  <head>
    <title
>Guess a number...</title>
    <link href="niceforms.css" rel="stylesheet" type="text/css" />
    <script language="javascript" type="text/javascript" src="niceforms.js" />
  </head>
  <body>
    <h1
>Guess a number...</h1>
    <f:view>
      <h:form styleClass="niceform">

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

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

        <div>
        Your guess:
        <h:inputText value="#{numberGuess.currentGuess}" id="inputGuess"
                  required="true" size="3"
                  rendered="#{(numberGuess.biggest-numberGuess.smallest) gt 20}">
          <f:validateLongRange maximum="#{numberGuess.biggest}"
                        minimum="#{numberGuess.smallest}"/>
        </h:inputText>
        <h:selectOneMenu value="#{numberGuess.currentGuess}"
                    id="selectGuessMenu" required="true"
                    rendered="#{(numberGuess.biggest-numberGuess.smallest) le 20 and
                            (numberGuess.biggest-numberGuess.smallest) gt 4}">
          <s:selectItems value="#{numberGuess.possibilities}" var="i" label="#{i}"/>
```

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

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

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

Si noti come il pulsante di comando chiama la transizione `guess` invece di chiamare direttamente un'azione.

La pagina `win.jspx` è prevedibile:

Esempio 1.22. win.jspx

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

```
  <body>
    <h1
>You won!</h1>
    <f:view>
      Yes, the answer was <h:outputText value="#{numberGuess.currentGuess}" />.
      It took you <h:outputText value="#{numberGuess.guessCount}" /> guesses.
      <h:outputText value="But you cheated, so it doesn't count!"
                    rendered="#{numberGuess.cheat}"/>
      Would you like to <a href="numberGuess.seam"
>play again</a
>?
    </f:view>
  </body>
  </html>
</jsp:root>
```

`lose.jspx` è più o meno uguale, quindi si passa oltre.

Infine diamo un'occhiata al codice dell'applicazione:

Esempio 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. La prima volta che una pagina JSP richiede un componente `numberGuess`, Seam ne crea uno nuovo, ed il metodo `@Create` viene invocato, consentendo che il componente si inizializzi.

Il file `pages.xml` inizia una *conversazione* Seam (maggiori informazioni più avanti), e specifica la definizione pageflow da usare per il flusso delle pagine della conversazione.

Esempio 1.24. 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-2.1.xsd">

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

</pages
>
```

Come si può vedere, questo componente Seam è pura logica di business! Non ha bisogno di sapere niente riguardo il flusso delle interazioni utente. Questo rende il componente potenzialmente più riutilizzabile.

1.5.2. Come funziona

Si analizzerà ora il flusso base dell'applicazione. Il gioco comincia con la vista `numberGuess.jspx`. Quando la pagina viene mostrata la prima volta, la configurazione `pages.xml` porta ad iniziare la conversazione ed associa il pageflow `numberGuess` a tale conversazione. Il pageflow inizia con un tag `start-page` che è uno stato d'attesa, e poi viene visualizzata la pagina `numberGuess.xhtml`.

La vista fa riferimento al componente `numberGuess`, provocando la creazione di una nuova istanza e la sua memorizzazione all'interno della conversazione. Viene chiamato il metodo `@Create` che inizializza lo stato del gioco. La vista mostra un `h:form` per consentire all'utente di editare `#{numberGuess.currentGuess}`.

Il pulsante "Guess" lancia l'azione `guess`. Seam usa il pageflow per gestire l'azione, la quale impone che il pageflow transiti allo stato `evaluateGuess`, innanzitutto invocando `#{numberGuess.guess}` che aggiorna il contatore ed i suggerimenti più alto/più basso nel componente `numberGuess`.

Lo stato `evaluateGuess` controlla il valore di `#{numberGuess.correctGuess}` e le transizioni agli stati `win` o `evaluatingRemainingGuesses`. Si assumache ilnumero sia sbagliato, nel qual caso il pageflow transita verso `evaluatingRemainingGuesses`. Questo è anche uno stato di decisione, che testa lo stato `#{numberGuess.lastGuess}` per determinare se l'utente ha ulteriori tentativi oppure no. Se ne ha (`lastGuess` è `falso`), si torna allo stato originale `displayGuess`. Infine si

raggiunge lo stato page, e quindi viene mostrata la pagina associata `/numberGuess.jspx`. Poiché la pagina ha un elemento redirect, Seam invia un redirect al browser dell'utente, ricominciando il processo.

Non si analizzerà ulteriormente lo stato, tranne per notare che se in una richiesta futura venisse presa la transizione `win` oppure `lose`, l'utente verrebbe portato a `/win.jspx` oppure `/lose.jspx`. Entrambi gli stati specificano che Seam debba terminare la conversazione, liberandosi dello stato del gioco e di quello del pageflow, prima di reindirizzare l'utente alla pagina finale.

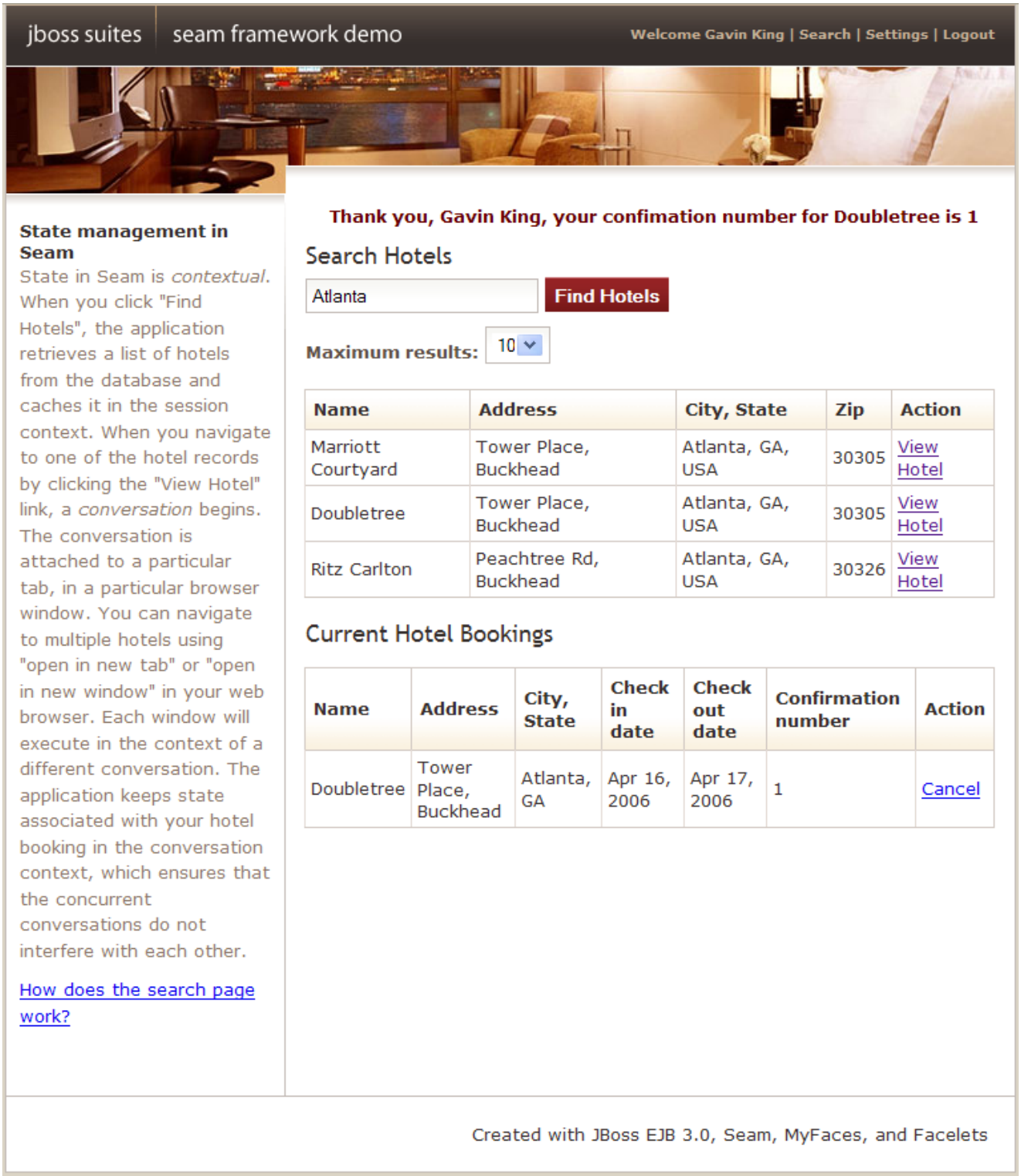
L'esempio indovina-numero contiene anche i pulsanti Giveup (abbandona) e Cheat (imbroglia). Si dovrebbe essere facilmente in grado di tracciare lo stato pageflow per le relative azioni. Si presti attenzione alla transizione `cheat`, che carica un sotto-processo per gestire tale flusso. Sebbene sia superfluo per quest'applicazione, questo dimostra come pageflow complessi possano venire spezzati in parti più piccole per renderle più facili da capire.

1.6. Un'applicazione Seam completa: esempio di Prenotazione Hotel

1.6.1. Introduzione

L'applicazione booking (prenotazione) è un sistema completo per la prenotazione di camere d'hotel, ed incorpora le seguenti funzionalità:

- Registrazione utente
- Login
- Logout
- Impostazione password
- Ricerca hotel
- Scelta hotel
- Prenotazione stanza
- Conferma prenotazione
- Lista di prenotazioni esistenti



L'applicazione booking utilizza JSF, EJB 3.0 e Seam, assieme a Facelets per la vista. C'è anche un port di quest'applicazione con JSF, Facelets, Seam, JavaBeans e Hibernate3.

Una delle cose che si noteranno utilizzando quest'applicazione per qualche tempo è che questa risulta essere estremamente *robusta*. Si può giocare con il pulsante indietro ed aggiornare le pagine, aprire più finestre ed inserire dati senza senso quanto si vuole, ma si vedrà che è molto

difficile mettere in difficoltà l'applicazione. Si può pensare che siano occorse settimane per testare e risolvere bug prima di raggiungere questo risultato. In verità non è così. Seam è stato progettato per rendere semplice la costruzione di applicazioni web robuste, e molta della robustezza che si è soliti doversi codificare da soli, con Seam viene naturale e automatica.

Quando si sfoglia il codice sorgente delle applicazioni e si impara come questa funziona, si osservi come sono stati usati la gestione dichiarativa dello stato e la validazione integrata per ottenere questa robustezza.

1.6.2. Panoramica sull'esempio di prenotazione

La struttura del progetto è identica al precedente, per installare e deployare quest'applicazione, si faccia riferimento a *Sezione 1.1, «Utilizzo degli esempi di Seam»*. Una volta avviata l'applicazione, si può accedere a questa puntando il browser all'indirizzo `http://localhost:8080/seam-booking/` [http://localhost:8080/seam-booking/]

L'applicazione utilizza sei bean di sessione per implementare la logica di business per le funzionalità nella lista.

- `AuthenticatorAction` fornisce la logica per l'autenticazione della login.
- `BookingListAction` recupera le prenotazioni esistenti per l'utente attualmente loggato.
- `ChangePasswordAction` aggiorna la password per l'utente attualmente loggato.
- `HotelBookingAction` implementa le funzionalità di prenotazione e conferma. Questa funzionalità è implementata come *conversazione*, e quindi è una delle classi più interessanti dell'applicazione.
- `HotelSearchingAction` implementa la funzionalità di ricerca hotel.
- `RegisterAction` registra un nuovo utente di sistema.

Tre entity bean implementano il modello di dominio di persistenza dell'applicazione.

- `Hotel` è un entity bean che rappresenta un hotel
- `Booking` è l'entity bean che rappresenta una prenotazione esistente
- `User` è un entity bean che rappresenta un utente che può fare una prenotazione

1.6.3. Capire le conversazioni in Seam

Si incoraggia a guardare il codice sorgente a piacimento. In questo tutorial ci concentreremo su alcune particolari funzionalità: ricerca hotel, selezione, prenotazione e conferma. Dal punto di vista dell'utente, tutto - dalla selezione dell'hotel alla conferma dellaprenotazione - è un'unica continua unità di lavoro, una *conversazione*. La ricerca, comunque, *non* è una parte della conversazione. L'utente può selezionare più hotel dalla stessa pagina dei risultati, in diversi tab del browser.

La maggior parte delle architetture delle applicazioni non ha alcun costrutto per rappresentare una conversazione. Questo causa enormi problemi nella gestione dello stato conversazionale.

Solitamente le applicazioni web Java usano una combinazione di diverse tecniche. Alcuni stati possono essere trasferiti nell'URL. Ciò che non può è messo o in `HttpSession` o mandato a database dopo ogni richiesta, e ricostruito dal database all'inizio di ogni nuova richiesta.

Poiché il database è il livello meno scalabile, questo risulta essere spesso ad un livello inaccettabile di scalabilità. La latenza è un ulteriore problema, dovuto al traffico extra verso e dal database ad ogni richiesta. Per ridurre questo traffico ridondante, le applicazioni Java spesso introducono una cache di dati (di secondo livello) che mantiene i dati comunemente acceduti tra le varie richieste. Questa cache è necessariamente inefficiente, poiché l'invalidazione è basato su una policy LRU invece di essere basata su quando l'utente termina di lavorare con i dati. Inoltre, poiché la cache è condivisa da diverse transazioni concorrenti, si è introdotta una schiera di problemi associati al fatto di mantenere lo stato della cache consistente con il database.

Ora si consideri lo stato mantenuto nella `HttpSession`. HttpSession è un ottimo posto per i veri dati di sessione, cioè dati che sono comuni a tutte le richieste che l'utente fa con l'applicazione. Comunque, non è un posto dove vanno memorizzati i dati riguardanti serie individuali di richieste. L'uso della sessione si complica velocemente quando si ha a che fare con il pulsante indietro e con le finestre multiple. In cima a questo, senza una programmazione attenta, i dati nella sessione HTTP posso crescere parecchio, rendendo la sessione HTTP difficile da tenere assieme. Lo sviluppo di meccanismi per isolare lo stato della session associata a differenti conversazioni concorrenti, e l'aggiunta di meccanismi di sicurezza per assicurare che lo stato della conversazione venga distrutto quando l'utente interrompe una delle conversazioni chiudendo una finestra del browser non è una questione per gente poco coraggiosa. Fortunatamente con Seam non occorre preoccuparsi di queste problematiche.

Seam introduce il *contesto conversazionale* come first class construct. Si può mantenere in modo sicuro lo stato conversazionale in questo contesto ed essere certi che avrà un ciclo di vita ben definito. Ancor meglio non servirà mandare continuamente avanti ed indietro i dati tra server e database, poiché il contesto di conversazione è una cache naturale di dati su cui l'utente sta lavorando.

In quest'applicazione si userà il contesto di conversazione per memorizzare i session bean stateful. C'è un'antica credenza nella comunità Java che ritiene che i session bean stateful siano nocivi alla scalabilità. Questo poteva essere vero nei primissimi giorni di Java Enterprise, ma oggi non è più vero. I moderni application server hanno meccanismi estremamente sofisticati per la replicazione dello stato dei session bean stateful. JBoss AS, per esempio, esegue una replicazione a grana fine, replicando solo quei valori degli attributi bean che sono cambiati. Si noti che tutti gli argomenti tecnici tradizionali per cui i bean stateful sono inefficienti si applicano allo stesso modo alla `HttpSession`, e quindi risulta fuorviante la pratica di cambiare stato dai componenti (session bean stateful) del business tier alla sessione web per cercare di migliorare le performance. E' certamente possibile scrivere applicazioni non scalabili usando session bean stateful non in modo corretto, o usandoli per la cosa sbagliata. Ma questo non significa che non si debba *mai*. Se non si è convinti, Seam consente di usare POJO invece dei session bean statefull. Con Seam la scelta è vostra.

L'applicazione di esempio prenotazione mostra come i componenti stateful con differenti scope possano collaborare assieme per ottenere comportamenti complessi. La pagina principale

dell'applicazione consente all'utente di cercare gli hotel. I risultati di ricerca vengono mantenuti nello scope di sessione di Seam. Quando l'utente naviga in uno di questi hotel, inizia una conversazione ed il componente con scope conversazione chiama il componente con scope sessione per recuperare l'hotel selezionato.

L'esempio di prenotazione mostra anche l'uso di RichFaces Ajax per implementare un comportamento rich client senza usare Javascript scritto a mano.

La funzionalità di ricerca è implementata usando un session bean statefull con scope di sessione, simile a quello usato nell'esempio di lista messaggi.

Esempio 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;
}                                                    4

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

```
}
```

1. L'annotazione EJB standard `@Stateful` identifica questa classe come un bean di sessione stateful. Bean di sessione stateful hanno di default uno scope legato al contesto di conversazione.
2. L'annotazione `@Restrict` applica al componente una restrizione di sicurezza. Restringe l'accesso al componente consentendolo solo agli utenti loggati. Il capitolo sicurezza spiega con maggior dettaglio la sicurezza in Seam.
3. L'annotazione *`@DataModel`* espone una `List` come `ListDataModel` JSF. Questo facilita l'implementazione di liste cliccabili per schermate di ricerca. In questo caso, la lista degli hotel è esposta nella pagina come `ListDataModel` all'interno della variabile di conversazione chiamata `hotels`.
4. L'annotazione standard EJB `@Remove` specifica che un bean di session stateful deve essere rimosso ed il suo stato distrutto dopo l'invocazione del metodo annotato. In Seam tutti i bean di sessione stateful devono definire un metodo senza parametri marcato con `@Remove`. Questo metodo verrà chiamato quando Seam distrugge il contesto di sessione.

La pagina principale dell'applicazione è una pagina Facelets. Guardiamo al frammento relativo alla ricerca hotel:

Esempio 1.26. main.xhtml

```
<div class="section">

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

   <h1
>Search Hotels</h1>

   <h:form id="searchCriteria">
   <fieldset
>
     <h:inputText id="searchString" value="#{hotelSearch.searchString}"

                  style="width: 165px;">                                    (1)
       <a:support event="onkeyup" actionListener="#{hotelSearch.find}"
                  reRender="searchResults" />
     </h:inputText>
      
     <a:commandButton id="findHotels" value="Find Hotels" action="#{hotelSearch.find}"

                     reRender="searchResults"/>                             (2)
```

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

    </h:form>

</div>

<a:outputPanel id="searchResults">
  <div class="section">
    <h:outputText value="No Hotels Found"
                  rendered="#{hotels != null and hotels.rowCount==0}"/>
    <h:dataTable id="hotels" value="#{hotels}" var="hot"
                 rendered="#{hotels.rowCount
>0}">
        <h:column>
          <f:facet name="header"
>Name</f:facet>
          #{hot.name}
        </h:column>
        <h:column>
          <f:facet name="header"
>Address</f:facet>
          #{hot.address}
        </h:column>
        <h:column>
          <f:facet name="header"
>City, State</f:facet>

          #{hot.city}, #{hot.state}, #{hot.country}
        </h:column
>
```

3

4

```
      <h:column>
         <f:facet name="header"
>Zip</f:facet>
         #{hot.zip}
      </h:column>
      <h:column>
         <f:facet name="header"
>Action</f:facet>
         <s:link id="viewHotel" value="View Hotel"
               action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
   </h:dataTable>
   <s:link value="More results" action="#{hotelSearch.nextPage}"
         rendered="#{hotelSearch.nextPageAvailable}"/>
 </div>
</a:outputPanel
>
```

1. Il tag RichFaces Ajax `<a:support>` consente ad un event action listener JSF di essere chiamato da `XMLHttpRequest` asincrono quando avviene un evento JavaScript `onkeyup`. Ancor meglio, l'attributo `reRender` consente di rigenerare un frammento di pagina JSF e di eseguire un aggiornamento parziale quando si riceve una risposta asincrona.
2. Il tag RichFaces Ajax `<a:status>` consente di mostrare un'immagine animata mentre si attende la restituzione di richieste asincrone.
3. Il tag RichFaces Ajax `<a:outputPanel>` definisce una regione della pagina che può essere rigenerata da una richiesta asincrona.
4. Il tag Seam `<s:link>` consente di attaccare un action listener JSF ad un link HTML ordinario (non-JavaScript). Il vantaggio rispetto al JSF `<h:commandLink>` è che mantiene le operazioni "Apri in nuova finestra" and "Apri in nuova scheda". Si noti inoltre che è stato usato un method binding con un parametro: `#{hotelBooking.selectHotel(hot)}`. Questo non è possibile con lo standard Unified EL, ma Seam fornisce un'estensione a EL che consente l'uso dei parametri sul qualsiasi espressione di method binding.

   Se ci si chiede come avvenga la navigazione, si possono trovare tutte le regole in `WEB-INF/pages.xml`; questo viene discusso in *Sezione 6.7, «Navigazione»*.

Questa pagina mostra i risultati di ricerca in modo dinamico man mano si digita, e consente di scegliere un hotel e passarlo al metodo `selectHotel()` di `HotelBookingAction`, che è il posto in cui *veramente* succede qualsosa di interessante.

Vediamo ora come l'applicazione d'esempio usa un bean di sessione stateful con scope di conversazione per ottenere una naturale cache di dati persistenti relativi alla conversazione. Il seguente codice d'esempio è abbastanza lungo. Ma se si pensa a questo come una lista di azioni che implementano vari passi della conversazione, risulta comprensibile. Si legga la classe dalla cima verso il fondo, come se fosse un racconto.

Esempio 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. Questo bean utilizza un *contesto di persistenza esteso* EJB3, e quindi ogni istanza di entity rimane gestita per l'intero ciclo di vita del session bean stateful.
2. L'annotazione `@Out` dichiara che il valore dell'attributo viene *outjected* in una variablie di contesto dopo le invocazioni del metodo. In questo caso, la variabile di contesto chiamata `hotel` verrà impostata al valore della variabile d'istanza `hotel` dopo che viene completata ciascuna invocazione dell'action listener.
3. L'annotazione `@Begin` specifica che il metodo annotato inizi una *conversazione long-running*, e quindi l'attuale contesto della conversazione non verrà distrutto alla fine della richiesta. Invece verrà riassociato ad ogni richiesta dalla finestra attuale e distrutto o dopo un timeout dovuto all'inattività della conversazione o dopo l'invocazione di un metodo annotato con `@End`.
4. L'annotazione `@End` specifica che il metodo annotato finisca l'attuale conversazione long-running, e quindi il contesto della conversazione attuale verrà distrutto alla fine della richiesta.
5. Questo metoto EJB di rimozione verrà chiamato quando Seam distruggerà il contesto della conversazione. Non si dimentichi di definire questo metodo!

`HotelBookingAction` contiene tutti i metodi action listenet che implementano, selezione, prenotazione e conferma, e mantiene lo stato relativo a questo lavoro nelle variabili di istanza. Pensiamo che questo codice sia molto più pulito e semplice degli attributi get e set in `HttpSession`.

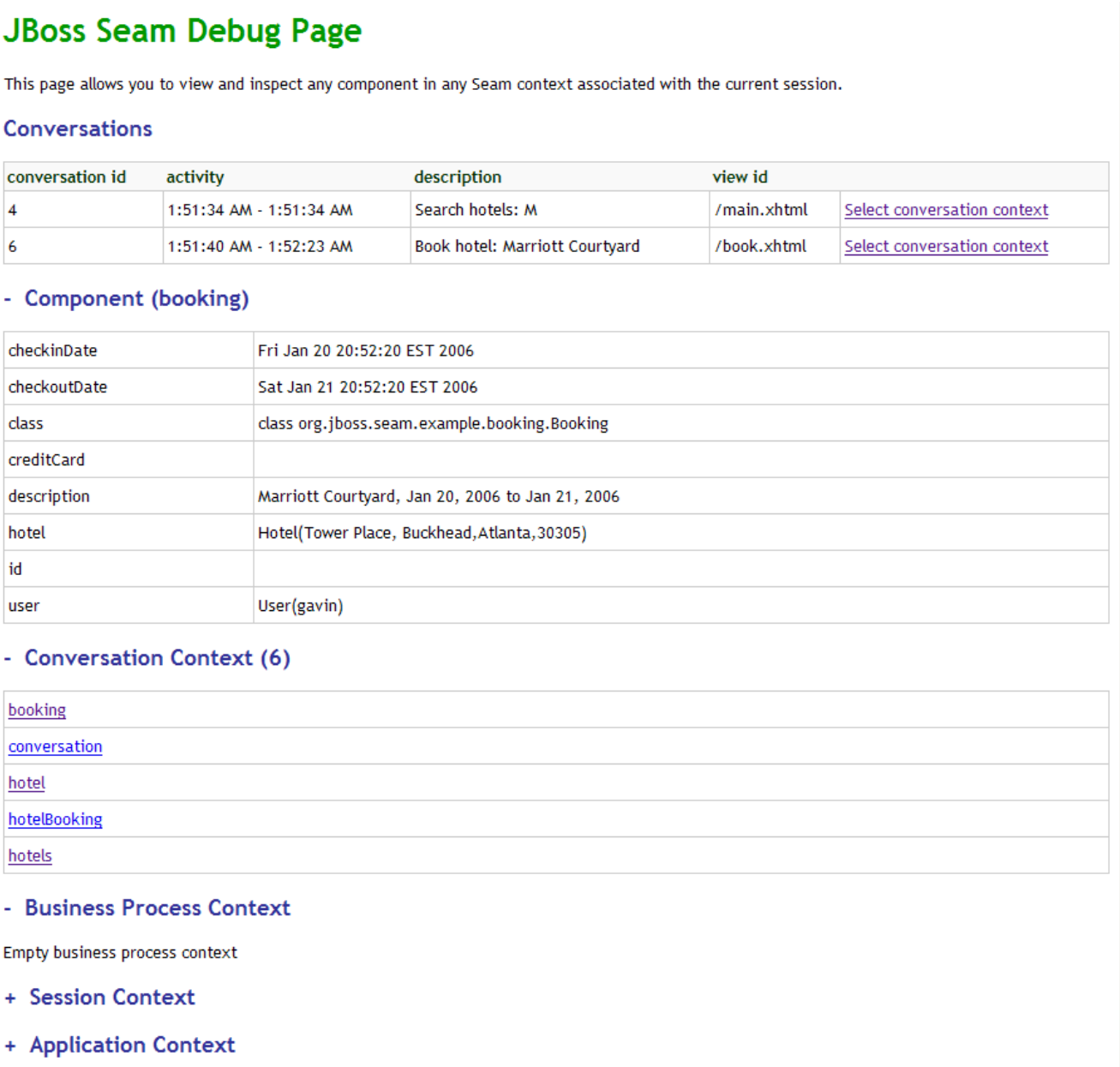
Ancor meglio, un utente può avere conversazioni multiple isolate per ogni sessione di login. Si provi! Loggarsi, eseguire una ricerca e navigare in diverse pagine d'hotel in diverse schede del browser. Si sarà in grado di lavorare e creare due differenti prenotazioni contemporaneamente. Se una conversazione viene lasciata a lungo inattiva, Seam andrà in timeout e distruggerà lo stato di quella conversazione. Se, dopo la chiusura di una conversazione, si premerà il pulsante indietro per tornare alla pagina precedente e si eseguirà un'azione, Seam si accorgerà che la conversazione è già terminata, e rimanderà l'utente alla pagina di ricerca.

1.6.4. La pagina di debug di Seam

Il WAR include anche `seam-debug.jar`. La pagina di debug di Seam sarà disponibile se questo jar è deployato in `WEB-INF/lib`, assieme a Facelets e se è stata impostata la proprietà di debug nel componente `init`:

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

Questa pagina consentirà di sfogliare ed ispezionare i componenti Seam in ogni contesto Seam associato alla sessione di login corrente. Si punti il browser su *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

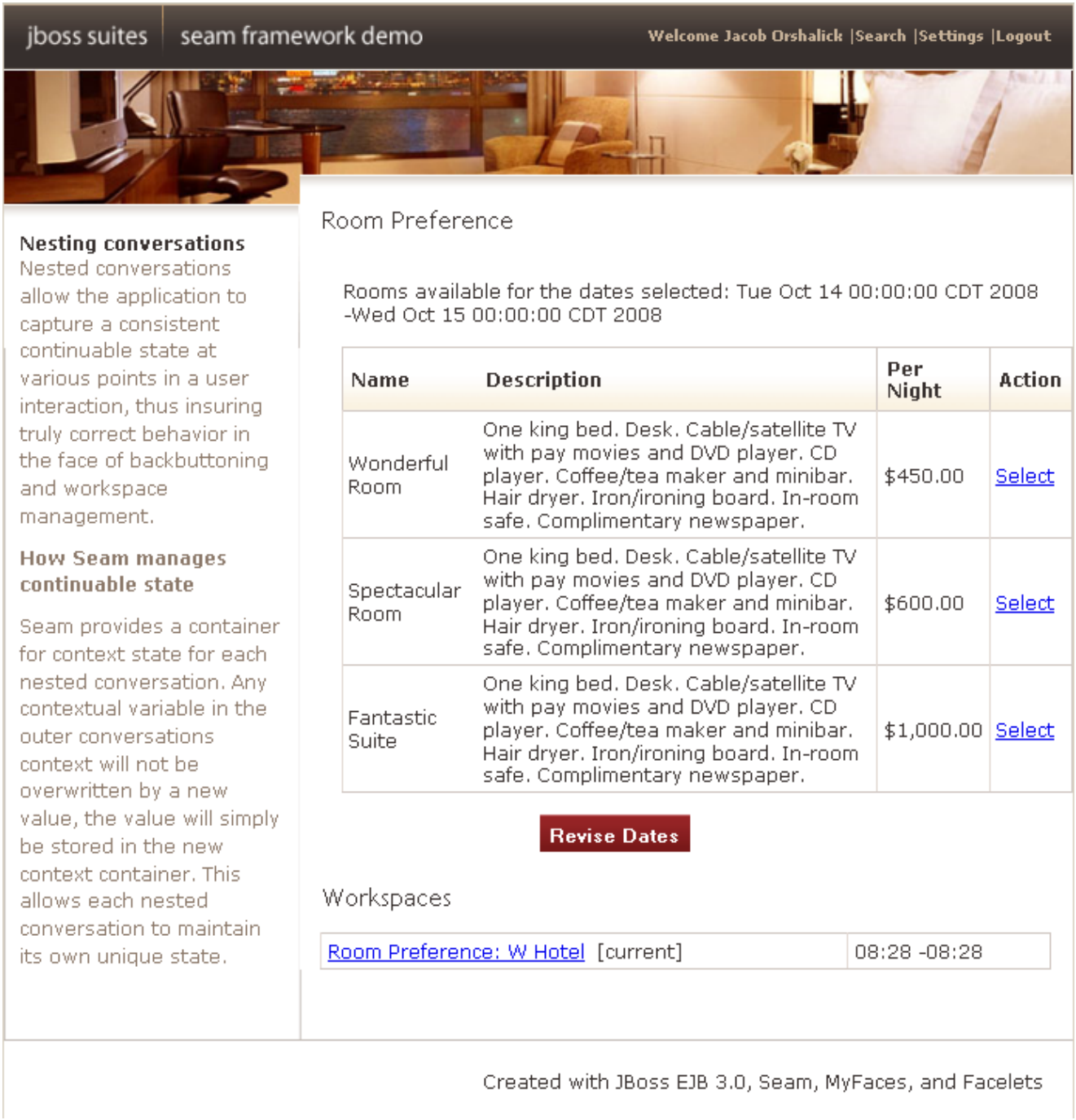
1.7. Conversazioni Annidate: estendere l'esempio di Prenotazione Hotel

1.7.1. Introduzione

Le conversazioni long-running rendono semplice mantenere la consistenza dello stato in un'applicazione anche in presenza di operazioni con finestre multiple o con il pulsante indietro. Sfrotunatamente, iniziare e finire una conversazione long-running non è sempre sufficiente. A

seconda dei requisiti dell'applicazione, le inconsistenze tra le aspettative dell'utente ed il reale stato dell'applicazione possono comunque sussistere.

L'applicazione prenotazione annidata estende le caratteristiche dell'applicazione prenotazione hotel aggiungendo la selezione della stanza. Ogni hotel ha camere disponibili con delle descrizioni che l'utente può scegliere. Questo richiede l'aggiunta di una pagina di selezione camera nel flusso di prenotazione hotel.



L'utente adesso ha l'opzione di selezionare una camera disponibile da aggiungere alla prenotazione. Come per l'applicazione precedentemente vista, questo porta a problemi di

consistenza dello stato. Come per la memorizzazione dello stato in `HTTPSession`, se una variabile di conversazione cambia, questo influenza tutte le finestre che operano dentro lo stesso contesto di conversazione.

Per dimostrare questo si supponga che l'utente cloni la schermata di selezione delle camera in una nuova finestra. L'utente quindi seleziona la *Wonderful Room* e procede alla schermata di conferma. Per vedere solamente quando costa vivere alla grande, l'utente ritorna alla finestra originale, seleziona la *Fantastic Suite* ed procede quindi alla conferma. Dopo aver visto il costo totale, l'utente decide che la praticità vince e ritorna alla finestra della *Wonderful Room* per procedere alla conferma.

In questo scenario, se semplicemente si memorizza lo stato nella conversazione non si è protetti da operazioni a finestre multiple all'interno della stessa conversazione. Le conversazioni innestate consentono di ottenere un comportamento corretto quando il contesto può variare all'interno della stessa conversazione.

1.7.2. Capire le Conversazioni Annidate

Si veda ora come l'esempio di prenotazione innestata estenda il comportamento dell'applicazione di prenotazione hotel tramite l'utilizzo di conversazioni innestate. Ancora, si può leggere la classe dalla cima verso il fondo, come un racconto.

Esempio 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")                                          (1)
  public void loadAvailableRooms()
  {
                        availableRooms    =    hotel.getAvailableRooms(booking.getCheckinDate(),
booking.getCheckoutDate());
    log.info("Retrieved #0 available rooms", availableRooms.size());
  }

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

    return booking.getTotal(roomSelection);
  }
                                                                      (2)
  @Begin(nested=true)
  public String selectPreference()
  {
    log.info("Room selected");
                                                                      (3)
    this.room = this.roomSelection;

    return "payment";
  }

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

    return "confirm";
  }


  @End(beforeRedirect=true)                                           (4)
  public String cancel()
  {
    log.info("ending conversation");
```

```
    return "cancel";
  }

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

1. L'istanza `hotel` viene iniettata dal contesto conversazione. L'hotel viene caricato tramite un *contesto di persistenza esteso* cosicché l'entity rimanga gestito lungo la conversazione. Questo consente di caricare in modo lazy la `availableRooms` tramite il metodo `@Factory` semplicemente seguendo l'associazione.
2. Quando si incontra `@Begin(nested=true)` , viene aggiunta una conversazione innestata allo stask delle conversazioni. Dentro una conversazione innestata, i componenti hanno accesso a tutto lo stato della conversazione più esterna, ma il settaggio di valori nel container dello stato delle conversazione innestata non influenza la conversazione più esterna. In aggiunta, le conversazioni esterne possono esistere in modo concorrente sopra la stessa conversazione più esterna, consentendo per ciascuna uno stato indipendente.
3. `roomSelection` viene messa in outjection nella conversazione tramite `@DataModelSelection`. Si noti che, poiché la conversazione innestata haun contesto indipendente, `roomSelection` è impostata solo nella nuova conversazione innestata. Dovesse l'utente selezionare un'altra preferenza in un'altra finestra o scheda, una nuova conversazione innestata verrebbe generata.
4. L'annotazione `@End` rimuove la conversazione dallo stack (pop) e ripristina la conversazione più esterna. `roomSelection` viene distrutta assieme al contesto della conversazione.

Quando si inizia una conversazione innestata, questa viene messa nello stack delle conversazioni. Nell'esempio `nestedbooking`, lo stack consiste in una conversazione long-running più esterna (la prenotazione) e ciascuna delle conversazioni innestate (selezione camere).

Esempio 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}"/>
                                                                    (1)
        <br/><br/>

        <h:dataTable value="#{availableRooms}" var="room"
            rendered="#{availableRooms.rowCount
> 0}">
          <h:column>
            <f:facet name="header"
>Name</f:facet>
            #{room.name}
          </h:column>
          <h:column>
            <f:facet name="header"
>Description</f:facet>
            #{room.description}
          </h:column>
          <h:column>
                                                                    (2)
            <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>
                                                                    (3)
            <h:commandLink id="selectRoomPreference"
              action="#{roomPreference.selectPreference}"
>Select</h:commandLink>
          </h:column>
        </h:dataTable>
      </div>
      <div class="entry">
        <div class="label"
> </div>
        <div class="input">
          <s:button id="cancel" value="Revise Dates" view="/book.xhtml"/>
```

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

1. Quando richiesto da EL, `#{availableRooms}` viene caricata dal metodo `@Factory` definito in `RoomPreferenceAction`. Il metodo `@Factory` verrà eseguito solo una volta per caricare i valore nel contesto attuale come istanza *`@DataModel`* .
2. L'invocazione dell'azione `#{roomPreference.selectPreference}` ha come risultato la selezione della riga e la sua impostazione in `@DataModelSelection`. Questo valore è quindi messo in outjection nel contesto della conversazione innestata.
3. Un cambiamento alle date semplicemente riporta a `/book.xhtml`. Si noti che ancora non è stata innestata alcuna conversazione (non è stata selezionata nessuna camera), e quindi la conversazione attuale può essere ristabilita. Il componente `<s:button >` semplicemente propaga la conversazione corrente quando viene mostrata la vista `/book.xhtml`.

Ora che si è visto come innestare una conversazione, vediamo come si può confermare la prenotazione una volta selezionata la camera. Questo può essere ottenuto semplicemente estendendo il comportamento di `HotelBookingAction`.

Esempio 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) Annotare un'azione con `@End(root=true)` termina la conversazione radice che distrugge effettivamente tutto lo stack delle conversazioni. Quando una conversazione termina, terminano anche le sue conversazioni innestate. Poiché la conversazione radice è quella che inizia tutto, questo è un modo semplice per distruggere e rilasciare tutto lo stato associato al workspace una volta confermata la prenotazione.

(2) `roomSelection` è associata solamente a `booking` su conferma dell'utente. Mentre l'outjection dei valori nel contesto della conversazione innestata non impatta sulla conversazione più esterna, qualsiasi oggetto iniettato dalla conversazione più esterna viene iniettato per referimento. Questo significa che qualsiasi cambiamento degli oggetti si rifletterà nella conversazione padre così come in ogni altra conversazione innestata.

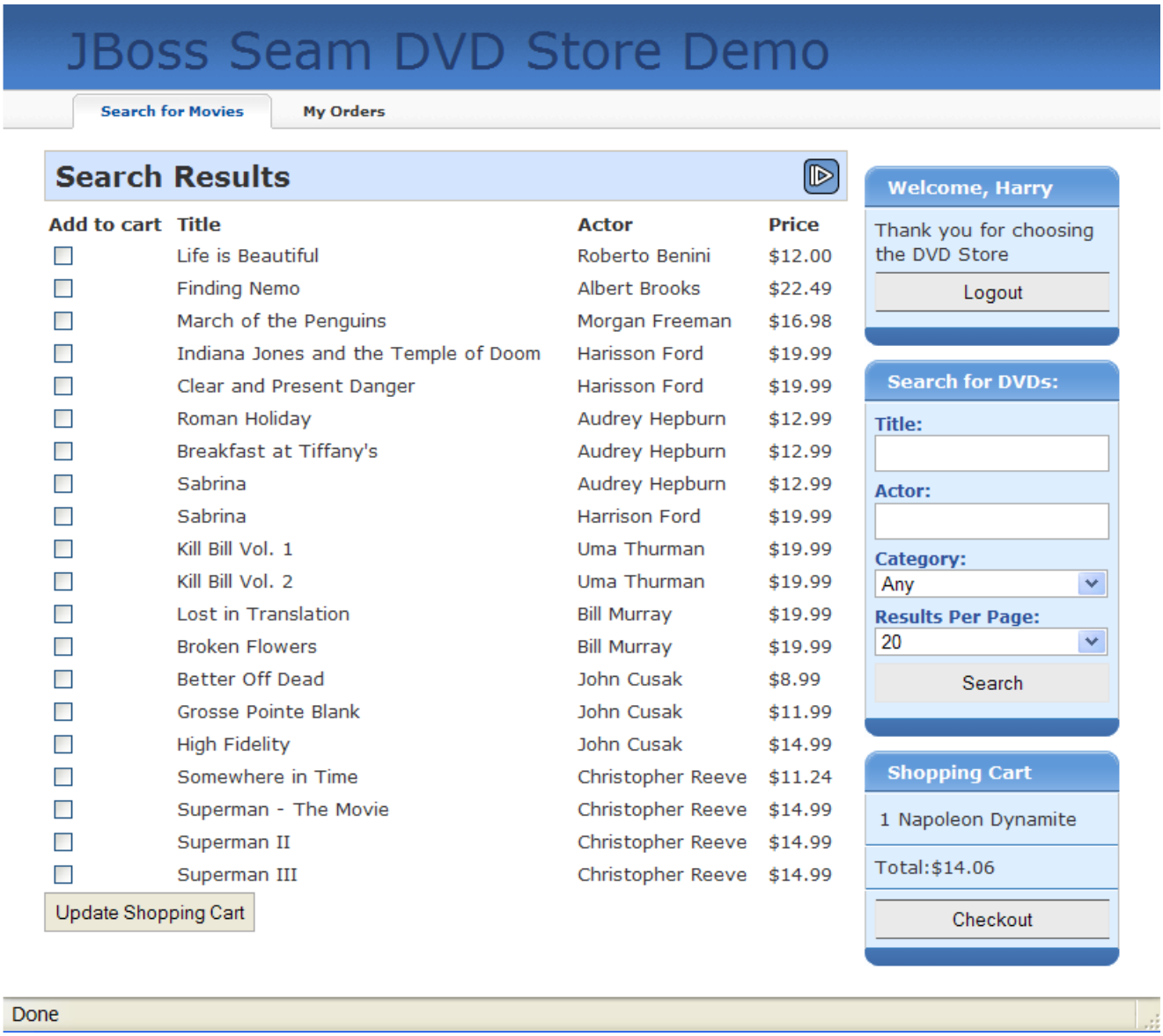
3 Annotando semplicemente l'azione di cancellazione con `@End(root=true, beforeRedirect=true)` , è possibile distruggere e rilasciare tutto lo stato associato al workspace prima di redirigere l'utente alla vista della selezione hotel.

Prova il deploy dell'applicazione, apri più finestre o tab e prova combinzioni di vari hotel con varie opzioni di camera. La conferma risulterà sempre nel giusto hotel e con la corretta opzione grazie al modello di conversazioni innestate.

1.8. Un'applicazione completa di Seam e jBPM: esempio di Negozio DVD

L'applicazione demo Negozio DVD mostra un utilizzo pratico di jBPM sia per la gestione task sia per il pageflow.

Le schermate utente sfruttano il pageflow jPDL per implementare la ricerca e la funzionalità di carrello della spesa.

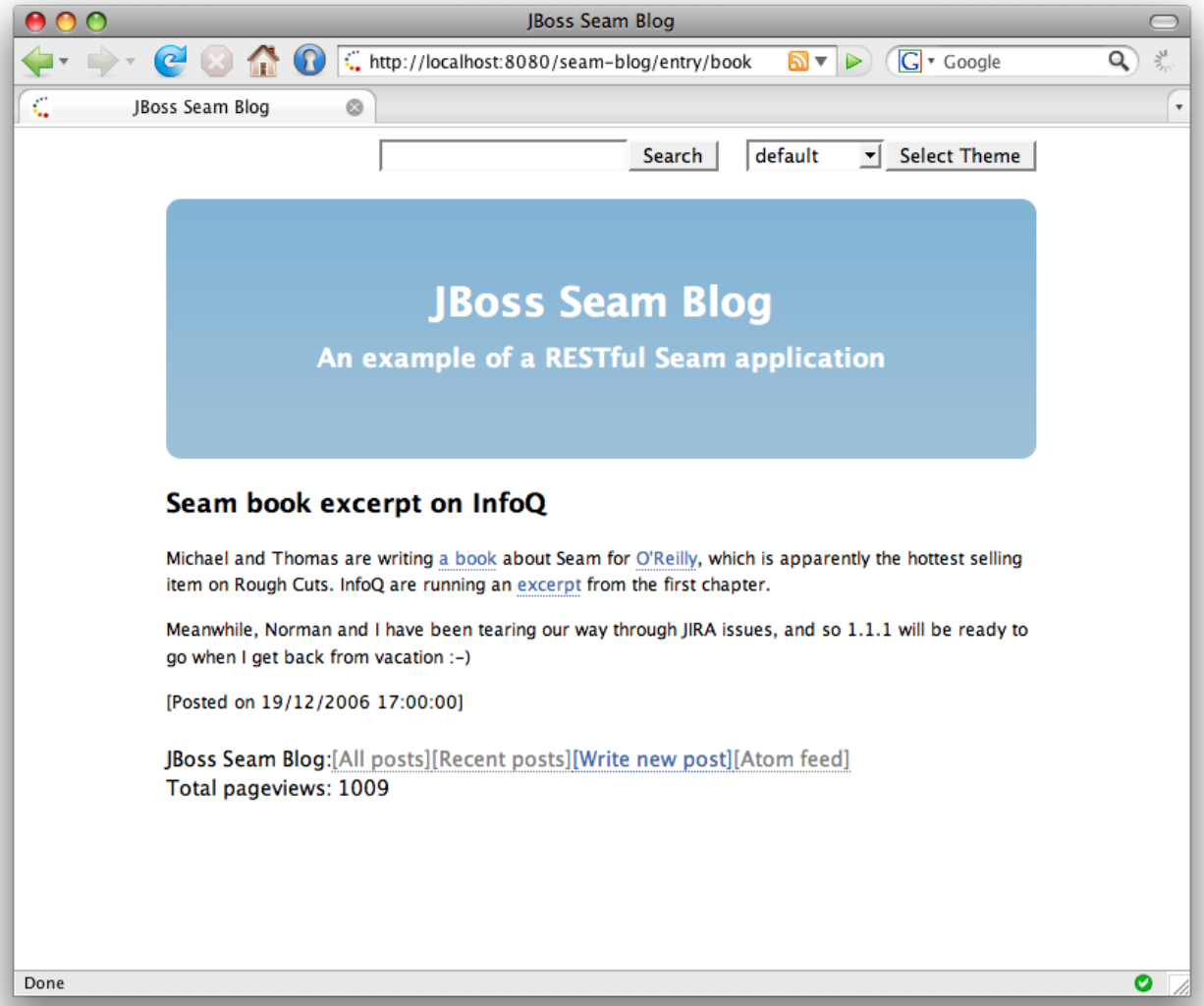


Le schermate di amministrazione utilizzano jBPM per gestire il ciclo di approvazione e di spedizione degli ordini. Il processo di business può anche essere cambiato dinamicamente, selezionando una diversa definizione di processo!

La demo Negozio DVD può essere eseguita dalla directory `dvdstore`, così come le altre applicazioni.

1.9. URL segnalibro con l'esempio Blog

Seam facilita l'implementazione di applicazioni che mantengano lo stato lato server. Comunque lo stato lato server non è sempre appropriato, specialmente per funzionalità che lavorano per il contenuto. Per questo genere di problemi spesso si vuole mantenere lo stato dell'applicazione nell'URL affiché ogni pagina possa essere acceduta in qualsiasi momento attraverso un segnalibro. L'esempio Blog mostra come implementare un'applicazione che supporti i segnalibri, anche nel caso di pagine con risultati di ricerca. Questo esempio mostra come Seam può gestire nell'URL lo stato di un'applicazione, così come Seam può riscrivere questi URL.



L'esempio Blog mostra l'uso di MVC di tipo "pull", dove invece di usare metodi action listener per recuperare i dati e preparare i dati per la vista, la vista preleva (pull) i dati dai componenti quando viene generata.

1.9.1. Utilizzo di MVC "pull"-style

Questo frammento della pagina facelets `index.xhtml` mostra una lista di messaggi recenti al blog:

Esempio 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
>
```

Se si arriva in quaste pagina da un segnalibro, come viene inizializzato `#{blog.recentBlogEntries}` usato da `<h:dataTable>`? `Blog` viene recuperato in modo lazy — "tirato" — quando serve, da un componente Seam chiamato `blog`. Questo è il flusso di controllo opposto a quello usato nei tradizionali framework web basati sull'azione, come ad esempio Struts.

Esempio 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. Questo componente utilizza un *contesto di persistenza gestito da Seam*. A differenza deglialtri esempi visti, questo contesto di persistenza è gestito da Seam, invece che dal container EJB3. Il contesto di persistenza estende l'intera richiesta web, consentendo di evitare le eccezioni che avvengono quando nella vista si accede ad associazioni non recuperate (unfetched).
2. L'annotazione `@Unwrap` impone a Seam di restituire il valore di ritorno del metodo — `Blog` — invece dell'attuale componente `BlogService` ai client. Questo è il *manager component pattern* di Seam.

Finora va bene, ma cosa succede se si memorizza il risultato di un invio di form, come ad esempio una pagina di risultati di ricerca?

1.9.2. Pagina bookmarkable dei risultati di ricerca

L'esempio Blog ha una piccola form in alto a destra di ogni pagina, che consente all'utente di cercare le entry del blog. E' definito in un file, `menu.xhtml`, incluso nel template facelets, `template.xhtml`:

Esempio 1.33.

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

Per implementare una pagina di risultati di ricerca memorizzabili come segnalibro, occorre eseguire un redirect del browser dopo aver elaborato la form di ricerca inviata. Poiché si è impiegato come esito d'azione l'id della vista JSF, Seam redirige automaticamente all'id vista quando la form viene inviata. In alternativa, si può definire una regola di navigazione come questa:

```
<navigation-rule>
```

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

Quindi la form avrebbe dovuto essere così:

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

Ma quando viene fatto il redirect, occorre includere i valori sottomessi con la form dentro l'URL per ottenere un URL memorizzabile come ad esempio `http://localhost:8080/seam-blog/search/`. JSF non fornisce un modo semplice per farlo, ma Seam sì. Per ottenere questo si usano due funzionalità di Seam: *i parametri di pagina* e *la riscrittura dell'URL*. Entrambi sono definiti in `WEB-INF/pages.xml`:

Esempio 1.34.

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

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

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

Il parametro di pagina istruisce Seam a fare collegare il parametro di richiesta chiamato `searchPattern` al valore di `#{searchService.searchPattern}`, sia quando arriava una richiesta per la pagina di ricerca, sia quando viene generato un link alla pagina di ricerca. Seam si prende la responsabilità di mantenere il link tra lo stato dell'URL e lo stato dell'applicazione, mentre voi, come sviluppatori, non dovete preoccuparvene.

Senza riscrittura, l'URL di una ricerca di un termine `book` sarebbe `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book`. Questo può andare bene, ma Seam può semplificare l'URL usando una regola di riscrittura. La prima regola, per il pattern `/search/{searchPattern}`, dice che in ogni volta che si ha un URL per search.xhtml con un parametro di richiesta searchPattern, si può semplificare quest'URL. E quindi l'URL visto prima, `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book` viene riscritto come `http://localhost:8080/seam-blog/search/book`.

Come per i parametri di pagina, la riscrittura dell'URL è bidirezionale. Questo significa che Sean inoltra le richieste di URL più semplici alla giusta vista e genera automaticamente la vista più semplice per voi. Non serve preoccuparsi della costruzione dell'URL. Viene tutto gestito in modo trasparente dietro. L'unico requisito è che per usare la riscrittura dell'URL, occorre abilitare il filtro di riscrittura in `components.xml`.

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

Il redirect di porta alla pagina `search.xhtml`:

```
<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
>
```

Il quale usa ancora MVC di tipo "pull" per recuperare i risultati di ricerca usando 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. Uso di MVC push in un'applicazione RESTful

Alcune volte ha più senso usare MVC push-style per processare pagine RESTful, e quindi Seam fornisce la nozione di *azione di pagina*. L'esempio di Blog utilizza l'azione di pagina per pagina di entry del blog, `entry.xhtml`. Notare che questo è un pò forzato, sarebbe stato più facile usare anche qua lo stile MVC pull-style.

Il componente `entryAction` funziona come una action class in un framework tradizionale orientato alle azioni e push-MVC come 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);
  }

}
```

Le azione nella pagina vengono anche dichiarate 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
>
```

Notare che l'esempio utilizza azioni di pagina per la validazione e per il conteggio delle pagine visitate. Si noti anche l'uso di un parametro nel binding di metodo all'interno della azione di pagina. Questa non è una caratteristiva standard di JSF EL, ma Seam consente di usarla, non solo per le azioni di pagina, ma anche nei binding di metodo JSF.

Quando la pagina `entry.xhtml` viene richiesta, Seam innanzitutto lega il parametro della pagina `blogEntryId` al modello. Si tenga presente che a causa della riscrittura dell'URL, il nome del parametro blogEntryId non verrà mostrato nell'URL. Seam quindi esegue l'azione, che recupera i dati necessari — `blogEntry` — e li colloca nel contesto di evento di Seam. Infine, viene generato il seguente:

```
<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
>
```

Se l'entry del blog non viene trovata nel database, viene lanciata l'eccezione `EntryNotFoundException`. Si vuole che quest'eccezione venga evidenziata come errore 404, non 505, e quindi viene annotata la classe dell'eccezione:

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

Un'implementazione alternativa dell'esempio non utilizza il parametro nel 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
>
```

E' una questione di gusti su quale implementazione tu preferisca.

La demo del blog mostra anche una semplice autenticazione di password, un invio di un post al blog, un esempio di caching frammentato della pagina e la generazione di atom feed.

Iniziare con Seam usando seam-gen

La distribuzione Seam comprende una utility da linea di comando che facilita la configurazione di un progetto eclipse, la generazione di un semplice codice skeleton Seam, ed il reverse engineer di un'applicazione da un database esistente.

Questo è il modo più semplice di sporcarti le mani con Seam e di preparare il colpo in canna per la prossima volta che ti troverai intrappolato in ascensore con uno di quei noiosi tipi di Ruby-on-Rail che farneticano quanto magnifico e meraviglioso sia l'ultimo giochino che hanno scoperto per realizzare applicazioni completamente banali che schiaffano delle cose nel database.

In questa relase, seam-gen funziona meglio per coloro che hanno JBoss AS. Si può usare il progetto generato con altri server J2EE o Java EE 5 facendo alcuni cambiamenti alla configurazione del progetto.

Si *può* usare seam-gen senza Eclipse, ma in questo tutorial, si vuole mostrare l'uso assieme ad Eclipse per il debugging ed i test. Se non si vuole installare Eclipse, si può seguire comunque questo tutorial - tutti i passi possono essere eseguiti da linea di comando.

Seam-gen è essenzialmente uno script Ant avvolto attorno a Hibernate Tools, assieme a qualche template. Questo facilita la sua personalizzazione in caso di bisogno.

2.1. Prima di iniziare

Assicurarsi di avere JDK 5 o JDK 6 (vedere *Sezione 41.1, «Dipendenze JDK»* per maggiori dettagli), JBoss AS 4.2 o 5.0 e Ant 1.7.0, con una versione recente di Eclipse, il plugin JBoss IDE di Eclipse ed il plugin TestNG per Eclipse correttamente installati prima di avviare. Aggiungere l'installazione di JBoss alla vista di JBoss Server in Eclipse. Avviare JBoss in modalità debug. Infine, avviare da comando all'interno della directory dove si è scompattato la distribuzione Seam.

JBoss ha un supporto sofisticato per l'hot re-deploy di WAR e EAR. Sfortunatamente, a causa di bug alla JVM, ripetuti redeploy di un EAR—situazione frequente durante lo sviluppo—causano un perm gen space della JVM. Per questa ragione, in fase di sviluppo si raccomanda di eseguire JBoss in una JVM avente parecchio perm gen space. Se si esegue JBoss da JBoss IDE, si può configurare questo nella configurazione di lancio del server, sotto "VM arguments". Si suggeriscono i seguenti valori:

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

Se non si ha molta memoria disponibile, si raccomanda come minimo:

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

Se si esegue JBoss da linea di comando, si possono configurare le opzioni JVM in `bin/run.conf`.

Se non si vuole avere a che fare con queste cose adesso, si lasci stare—ce ne si occuperà quando capiterà la prima `OutOfMemoryException`.

2.2. Comfigurare un nuovo progetto

La prima cosa da fare è configurare seam-gen per il proprio ambiente: la directory di installazione JBoss AS, il workspace, e la connessione del database. E' facile, si digiti:

```
cd jboss-seam-2.0.x
seam setup
```

E verranno richieste le informazioni necessarie:

```
~/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 home directory [C:/Program Files/jboss-4.2.3.GA] [C:/Program Files/
jboss-4.2.3.GA]
/Applications/jboss-4.2.3.GA
    [input] Enter the project name [myproject] [myproject]
helloworld
     [echo] Accepted project name as: helloworld
    [input] Select a RichFaces skin (not applicable if using ICEFaces) [blueSky] ([blueSky], classic,
 ruby, wine, deepMarine, emeraldTown, sakura, DEFAULT)

     [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? [hsql]  ([hsql], mysql, oracle, postgres, mssql,
db2, sybase, enterprisedb, h2)
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
   [input] Enter your ICEfaces home directory (leave blank to omit ICEfaces) [] []

[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 $
```

Il tool fornisce dei valori di default, che possono essere accettati semplicemente premendo Invio alla richiesta.

La scelta più importante da fare è tra il deploy EAR e il deploy WAR del progetto. I progetti EAR supportano EJB 3.0 e richiedono Java EE 5. I progetti WAR non supportano EJB 3.0, ma possono

essere deployati in ambienti J2EE. L'impacchettamento di un WAR è più semplice da capire. Se si installa un application server predisposto per EJB3, come JBoss, si scelga `ear`. Altrimenti si scelga `war`. Assumeremo per il resto del tutorial che la scelta sia il deploy EAR, ma si potranno compiere gli stessi passi per il deploy WAR.

Se si sta lavorando con un modello di dati esistente, ci si assicuri di dire a seam-ger che le tabelle esistono già nel database.

Le impostazioni vengono memorizzate in `seam-gen/build.properties`, ma si possono anche modificare eseguendo semplicemente una seconda volta `seam setup`.

Ora è possibile creare un nuovo progetto nella directory di workspace di Eclipse, digitando:

```
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>
```

Questo copia i jar Seam, i jar dipendenti ed il driver JDBC nel nuovo progetto Eclipse, e genera tutte le risorse necessarie ed il file di configurazione, i file template di facelets ed i fogli di stile, assieme ai metadati di Eclipse e allo script per il build di Ant. Il progetto Eclipse verrà automaticamente deployato in una struttura di directory esplosa in JBoss AS non appena si aggiungerà il progetto usando `New -> Project... -> General -> Project -> Next`, digitando `Project name` (`helloworld` in questo caso), e poi cliccando `Finish`. Non selezionare `Java Project` dallo wizard New Project.

Se in Eclipse la JDK di default non è Java SE 5 o Java SE 6 JDK, occorre selezionare un JDK compatibile Java SE 5 usando `Project -> Properties -> Java Compiler`.

in alternativa, si può eseguire il deploy del progetto dal di fuori di Eclipse digitando `seam explode`.

Si vada in `http://localhost:8080/helloworld` per vedere la pagina di benvenuto. Questa è una pagina facelets, `view/home.xhtml`, che utilizza il template `view/layout/template.xhtml`. In Eclipse si può modificare questa pagina, oppure il template, e vedere *immediatamente* i risultati, cliccando il pulsante aggiorna del browser.

Non si abbia paura dell'XML, i documenti di configurazione generati nella directory di progetto. Per la maggiore parte delle volte sono standard per Java EE, parti che servono per creare la prima volta e poi non si guarderanno più, ed al 90% sono sempre le stesse per ogni progetto Seam. (Sono così facili da scrivere che anche seam-gen può farlo.)

Il progetto generato include tre database e le configurazioni per la persistenza. I file `persistence-test.xml` e `import-test.sql` vengono usati quando di eseguono i test di unità TestNG con HSQLDB. Lo schema del database ed i dati di test in `import-test.sql` vengono sempre esportati nel database prima dell'esecuzione dei test. I file `myproject-dev-ds.xml`, `persistence-dev.xml` e `import-dev.sql` sono usati per il deploy dell'applicazione nel database di sviluppo. Lo schema può essere esportato automaticamente durante il deploy, a seconda che si sia detto a seam-gen che si sta lavorando con un database esistente. I file `myproject-prod-ds.xml`, `persistence-prod.xml` e `import-prod.sql` sono usati per il deploy dell'applicazione nel database di produzione. Lo schema non viene esportato automaticamente durante il deploy.

2.3. Creazione di una nuova azione

Se si è abituati ad usare un framework web action-style, ci si domanderà come in Java sia possibile creare una semplice pagina web con un metodo d'azione stateless. Se si digita:

```
seam new-action
```

Seam chiederà alcune informazioni, e genererà per il progetto una nuova pagina facelets ed i componenti Seam.

```
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>
```

Poiché è stato aggiunto un nuovo componente Seam, occorre riavviare il deploy della directory esplosa. E' possibile farlo digitando `seam restart`, od eseguendo il target `restart` nel file `build.xml` del progetto generato all'interno di Eclipse. Un altro modo per forzare il riavvio è editare in Eclipse il file `resources/META-INF/application.xml`. *Si noti che non occorre riavviare JBoss ogni volta che cambia l'applicazione.*

Adesso si vada in `http://localhost:8080/helloworld/ping.seam` e si clicchi il pulsante. Si può vedere il codice sottostante l'azione guardando il progetto nella directory `src`. Si metta un breakpoint nel metodo `ping()`, e si clicchi nuovamente il pulsante.

Infine si cerchi il file `PingTest.xml` nel pacchetto dei test e si eseguano i test d'integrazione usando il plugin TestNG di Eclipse. In alternativa, si eseguano i test usando `seam test` od il target `test` del build generato.

2.4. Creazione di una form con un'azione

Il prossimo passo è creare una form. Si digiti:

```
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>
```

Si riavvii di nuovo l'applicazione e si vada in `http://localhost:8080/helloworld/hello.seam`. Quindi si guardi il codice generato. Avviare i test. Si aggiungano nuovi campi alla form e al componente Seam (ricordarsi di riavviare il deploy ad ogni cambiamento del codice Java).

2.5. Generazione di un'applicazione da database esistente

Si creino manualmente le tabelle nel database. (Se occorre passare ad un database diverso, si esegua di nuovo `seam setup`.) Adesso si digiti:

```
seam generate-entities
```

Riavviare il deploy ed andare in `http://localhost:8080/helloworld`. Si può sfogliare il database, modificare gli oggetti esistenti e creare nuovi oggetti. Se si guarda al codice generato, probabilmente ci si meraviglierà di quanto è semplice! Seam è stato progettato affiché sia semplice scrivere a mano il codice d'accesso ai dati, anche per persone che non vogliono barare usando seam-gen.

2.6. Generazione di un'applicazione da entity JPA/EJB3 già esistenti

Si mettano le classi entity esistenti dentro `src/main`. Ora si digiti:

```
seam generate-ui
```

Si avvi il deploy, e si vada alla pagina `http://localhost:8080/helloworld`.

2.7. Eseguire il deploy dell'applicazione come EAR

Infine si vuole essere in grado di eseguire il deploy dell'applicazione usando l'impacchettamento standard di Java EE 5. In primo luogo occorre rimuovere la directory esplosa eseguendo `seam unexplode`. Per fare il deploy dell'EAR, si può digitare da linea di comando `seam deploy`, od eseguire il target `deploy` dello script di build del progetto generato. L'undeploy può essere eseguito usando `seam undeploy` od il target `undeploy`.

Di default l'applicazione verrà deployata con il *profile dev*. L'EAR includerà i file `persistence-dev.xml` e `import-dev.sql`, e verrà deployato il file `myproject-dev-ds.xml`. Si può cambiare il profilo ed usare il *profile prod*, digitando:

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

Si possono anche definire nuovi profili di deploy per l'applicazione. Basta aggiungere file al progetto—per esempio, `persistence-staging.xml`, `import-staging.sql` e `myproject-staging-ds.xml`—e selezionare il nome del profile usando `-Dprofile=staging`.

2.8. Seam e hot deploy incrementale

Quando si fa il deploy di un'applicazione Seam come directory esplosa, si ottiene il supporto al deploy a caldo (hot deploy) durante il deploy. Occorre abilitare la modalità debug sia in Seam sia in Facelets, aggiungendo questa linea a `components.xml`:

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

Ora i seguenti file potranno essere rideployati senza riavviare l'applicazione:

- qualsiasi pagina facelets
- qualsiasi file `pages.xml`

Ma se si vuole cambiare il codice Java, occorre comunque riavviare nuovamente l'applicazione. (In JBoss questo può essere ottenuto toccando il descrittore di deploy: `application.xml` per un deploy con EAR, oppure `web.xml` per un deploy WAR.)

Ma se si vuole velocizzare il ciclo modifica/compila/testa, Seam supporta il redeploy incrementale dei compoenti JavaBean. Per usarlo occorre fare il deploy dei componenti JavaBean nella directory `WEB-INF/dev`, cosicché vengano caricati da uno speciale classloader di Seam, invece del classloader WAR o EAR.

Occorre essere consapevoli delle seguenti limitazioni:

- i componenti devono essere componenti JavaBean, non possono essere bean EJB3 (stiamo lavorando per sistemare questa limitazione)
- gli entity non possono mai essere deployati a caldo (hot deployment)
- i componenti deployati via `components.xml` non possono essere deployati a caldo
- i componenti deployabili a caldo non saranno visibili alle classi deployate fuori da `WEB-INF/dev`
- La modalità di degub di Seam deve essere abilitata e `jboss-seam-debug.jar` deve trovarsi in `WEB-INF/lib`
- Occorre avere installato in web.xml il filtro Seam.
- Si possono vedere errori se il sistema è messo sotto carico ed è abilitato il debug.

Se si crea un progetto WAR usando seam-gen, il deploy incrementale a caldo è già abilitato per le classi collocate nella directory dei sorgenti `src/hot`. Comunque seam-gen non supporta il deploy incrementale a caldo per i progetti EAR.

2.9. Uso di Seam con JBoss 4.0

Seam 2.0 è stato sviluppato per JavaServer Faces 1.2. Usando JBoss AS, si raccomanda di usare JBoss 4.2 o JBoss 5.0, che incorpora l'implementazione di riferimento JSF 1.2. Comunque è possibile usare Seam 2.0 su piattaforma JBoss 4.0. Ci sono due passi base richiesti per farlo: installare la versione JBoss 4.0 con EJB3 abilitato e sostituire MyFaces con l'implementazione di riferimento JSF 1.2. Una volta completati questi passi, le applicazioni Seam 2.0 possono essere deployate in JBoss 4.0.

2.9.1. Install JBoss 4.0

JBoss 4.0 non porta con sé una configurazione di default compatibile con Seam. Per eseguire Seam, occorre installare JBoss 4.0.5 usando l'installer JEMS 1.2 con il profile ejb3 selezionato.

Seam non funzionerà con un'installazione che non include il supporto EJB3. L'installer JEMS può essere scaricato da *http://labs.jboss.com/jemsinstaller/downloads*.

2.9.2. Installare JSF 1.2 RI

La configurazione web per JBoss 4.0 può essere trovata in `server/default/deploy/jbossweb-tomcat55.sar`. Occorre cancellare `myfaces-api.jar` e `myfaces-impl.jar` dalla directory `jsf-libs`. Poi occorre copiare `jsf-api.jar`, `jsf-impl.jar`, `el-api.jar`, e `el-ri.jar` in questa directory. I JAR JSF possono essere trovati nella directory `lib` di Seam. I JAR EL possono essere ottenuti dalla release Seam 1.2.

Occorre modificare il `conf/web.xml`, sostituendo `myfaces-impl.jar` con `jsf-impl.jar`.

Iniziare con Seam usando JBoss Tools

JBoss Tool è una collezione di plugin Eclipse. JBoss Tool è un wizard per la creazione di progetti Seam, Content Assist per Unified Expression Language (EL) sia in facelets e codice Java, un editor grafico per jPDL, un editor grafico per i file di configurazione di Seam, supporta l'esecuzione dei test di integrazione di Seam dall'interno di Eclipse, e molto altro.

In breve, se sei un utilizzatore di Eclipse, allora vorrai JBoss Tools!

JBoss Tools, come con seam-gen, funziona meglio con JBoss AS, ma è possibile con alcuni accorgimenti far girare l'applicazione in altri application server. I cambiamenti sono più o meno quelli descritti più avanti per seam-gen in questa guida.

3.1. Prima di iniziare

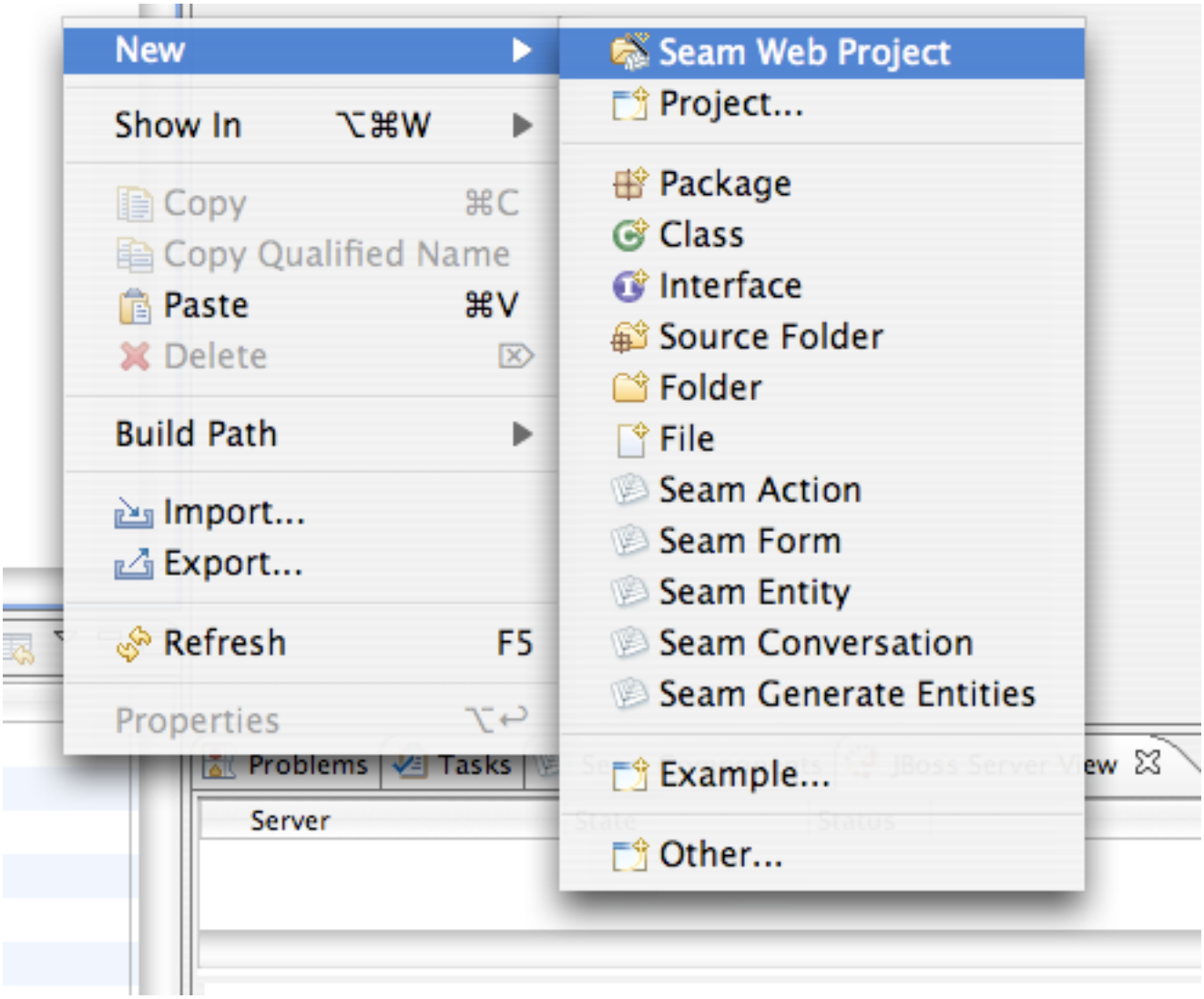
Assicurarsi di avere JDK 5, JBoss AS 4.2 o 5.0, Eclipse 3.3, i plugin di JBoss Tool (almeno Seam Tools, Visual Page Editor, jBPM Tools e JBoss AS Tools) ed il plugin TestNG per Eclipse correttamente installati prima di partire.

Vedere la pagina ufficiale *JBoss Tools installation* [http://www.jboss.org/tools/download/installation] per configurare velocemente JBoss Tools in Eclipse. Controllare anche la pagina *Installing JBoss Tools* [http://www.jboss.org/community/wiki/InstallingJBossTools] nella community wiki di JBoss per altri dettagli e per approcci di installazioni alternative.

3.2. Configurare un nuovo progetto Seam

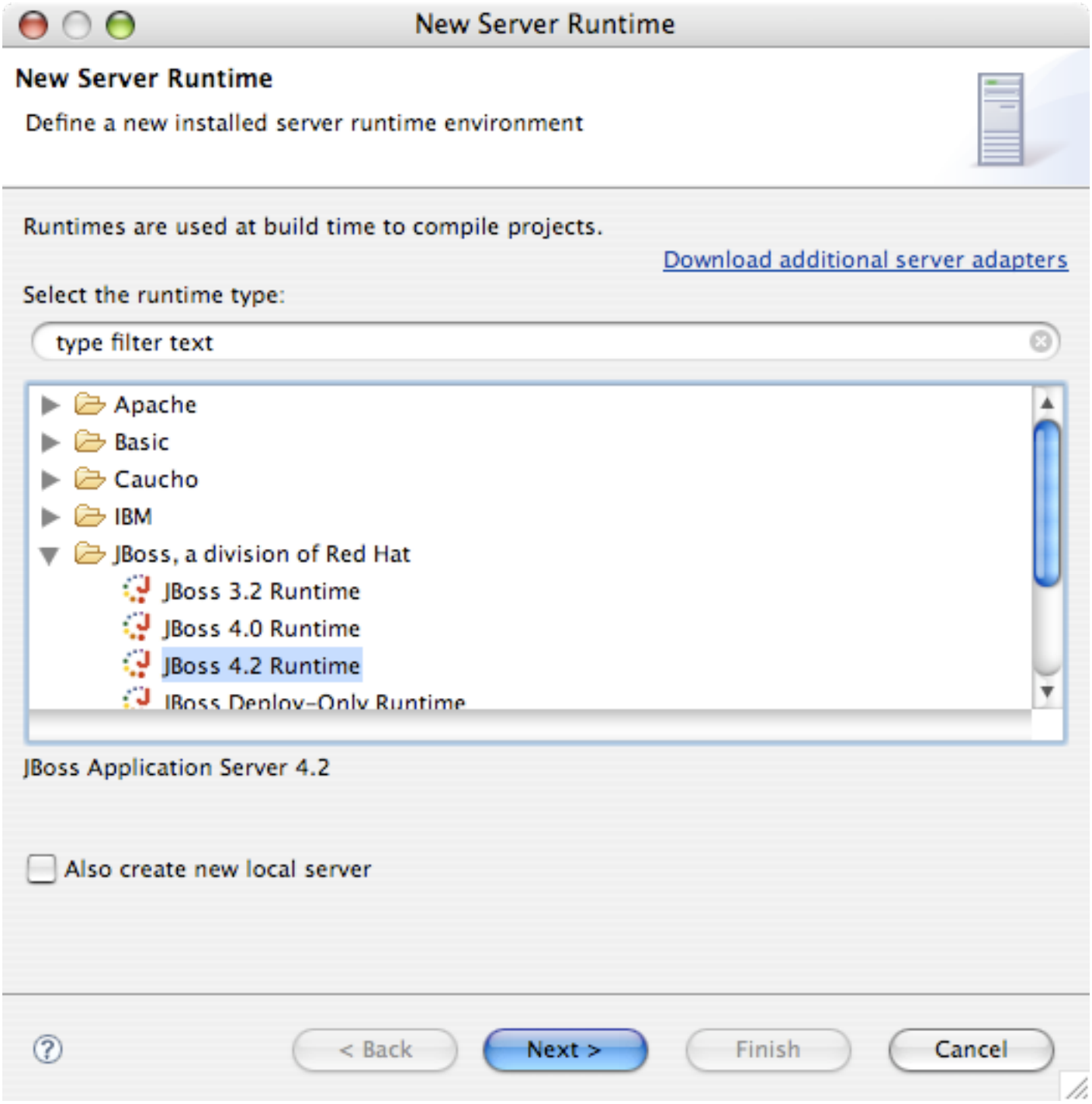
Avviare Eclipse e selezionare la prospettiva *Seam*.

Si vada in *File -> New -> Seam Web Project*.

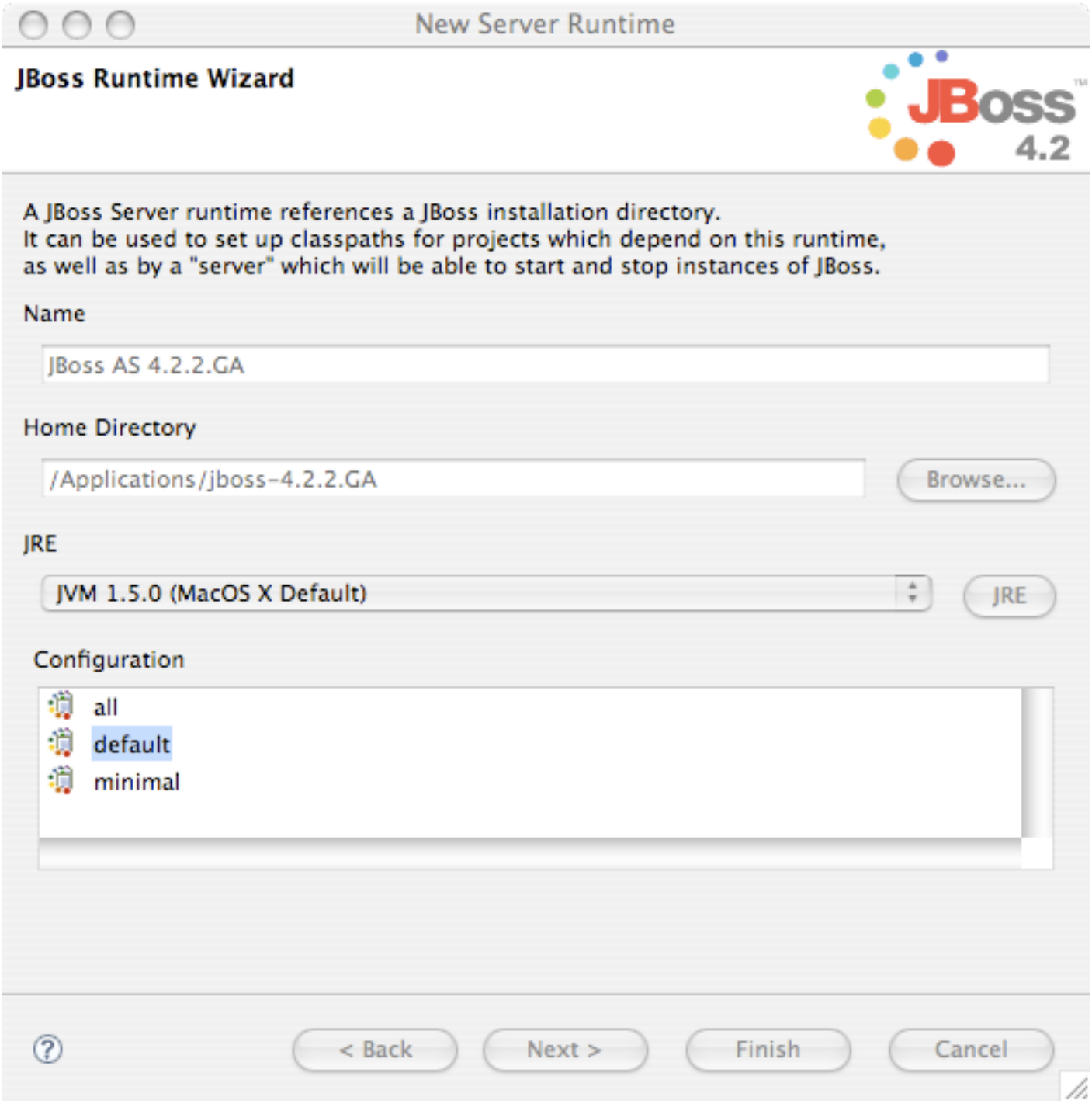


Primo, inserire un nome per il nuovo progetto. Per questo tutorial si userà `helloworld` .

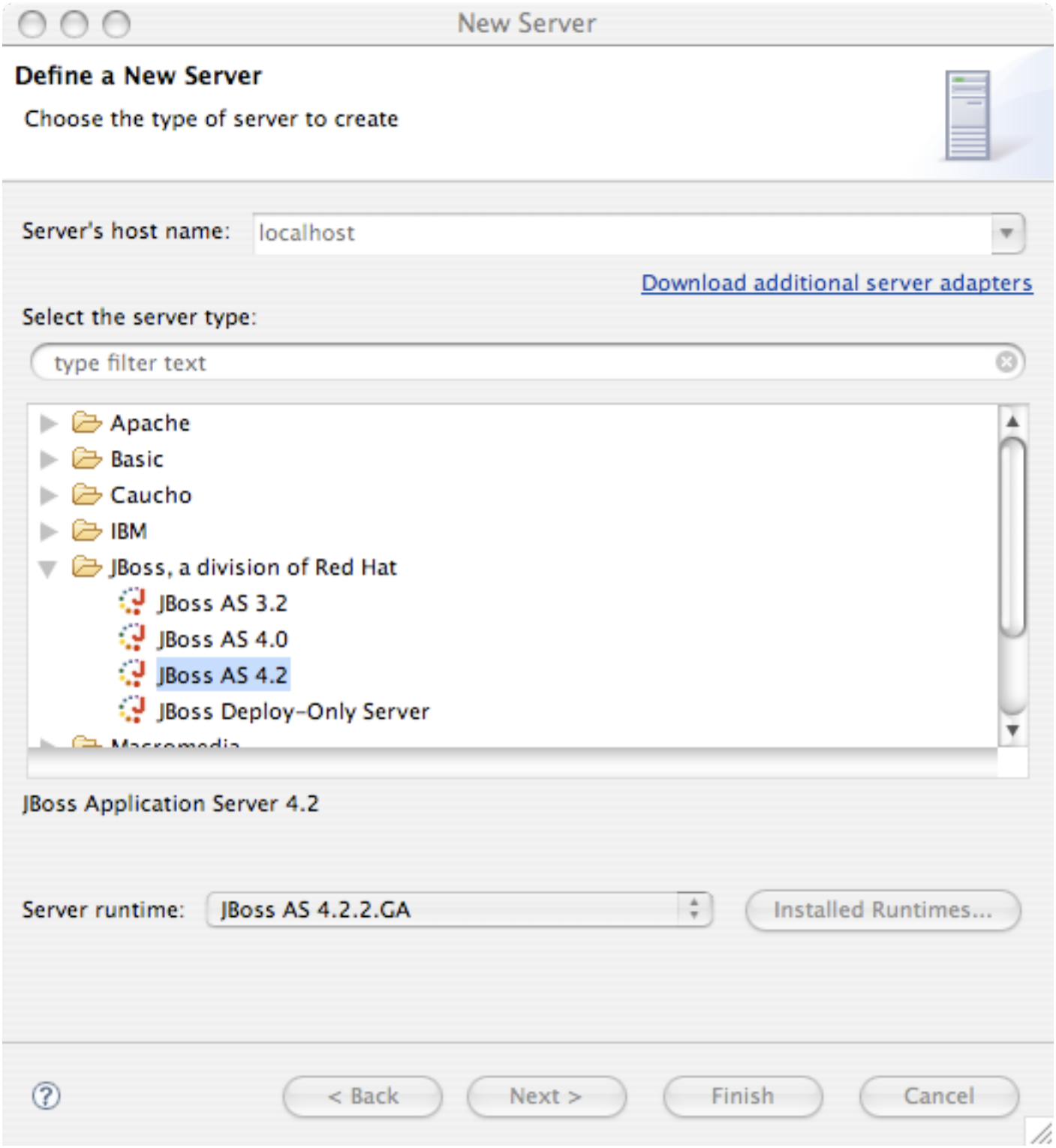
Ora, occorre dire a JBoss Tools dell'esistenza di JBoss AS. In questo esempio si usa JBoss AS 4.2, anche se è certamente possibile usare anche JBoss AS 5.0. Questo è un processo in due fasi, primo occorre definire un runtime, assicurarsi di selezionare JBoss AS 4.2:



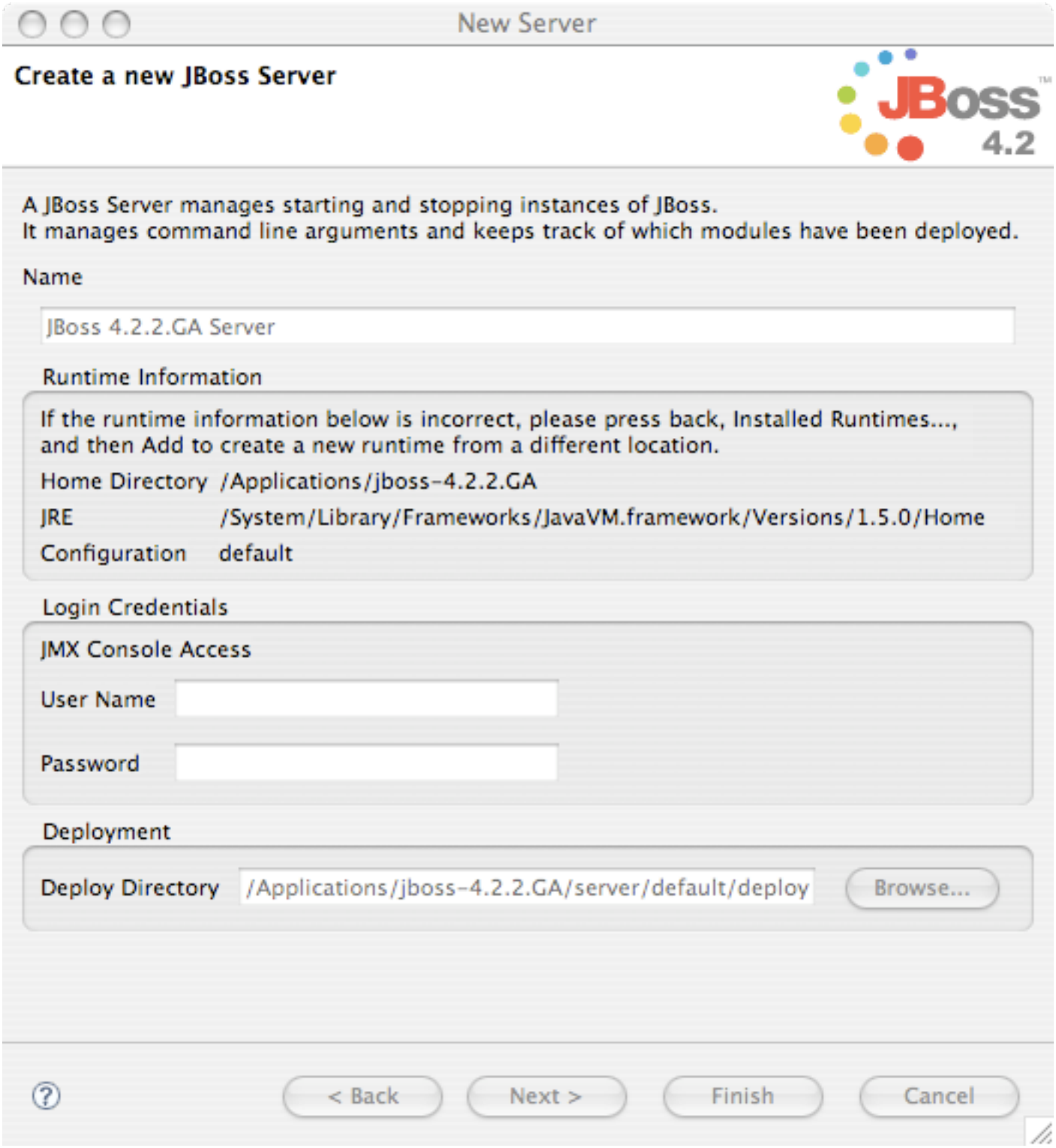
Inserire un nome per il runtime, e localizzarlo sul proprio hard disk:



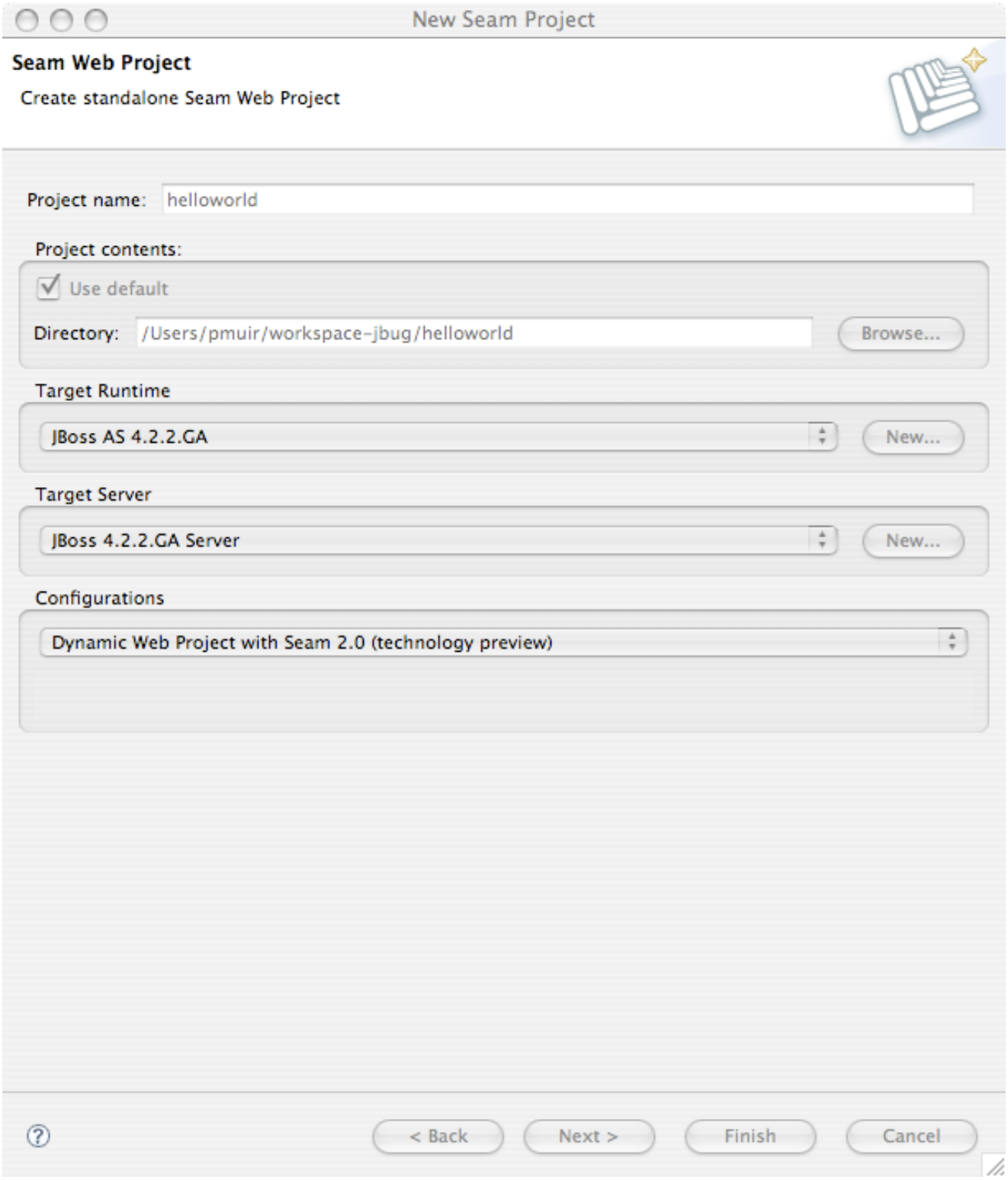
Poi, occorre definire un server in cui JBoss Tools possa fare il deploy. Assicurarsi di selezionare ancora JBoss AS 4.2, ed anche il runtime appena definito:



Alla successiva schermata si dia un nome al server e si prema *Finish*:

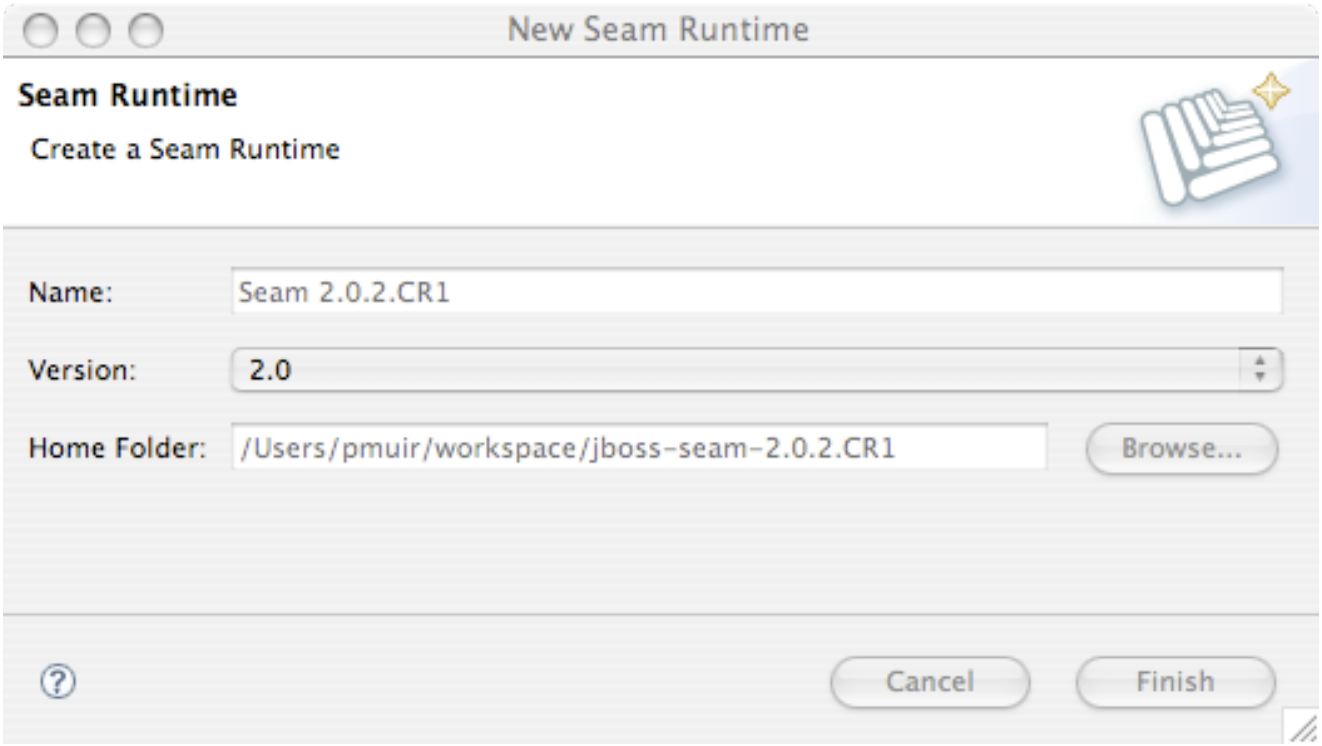


Assicurarsi che siano selezionati il runtime ed il server appena creati, selezionare *Dynamic Web Project with Seam 2.0 (technology preview)* e premere *Next*:



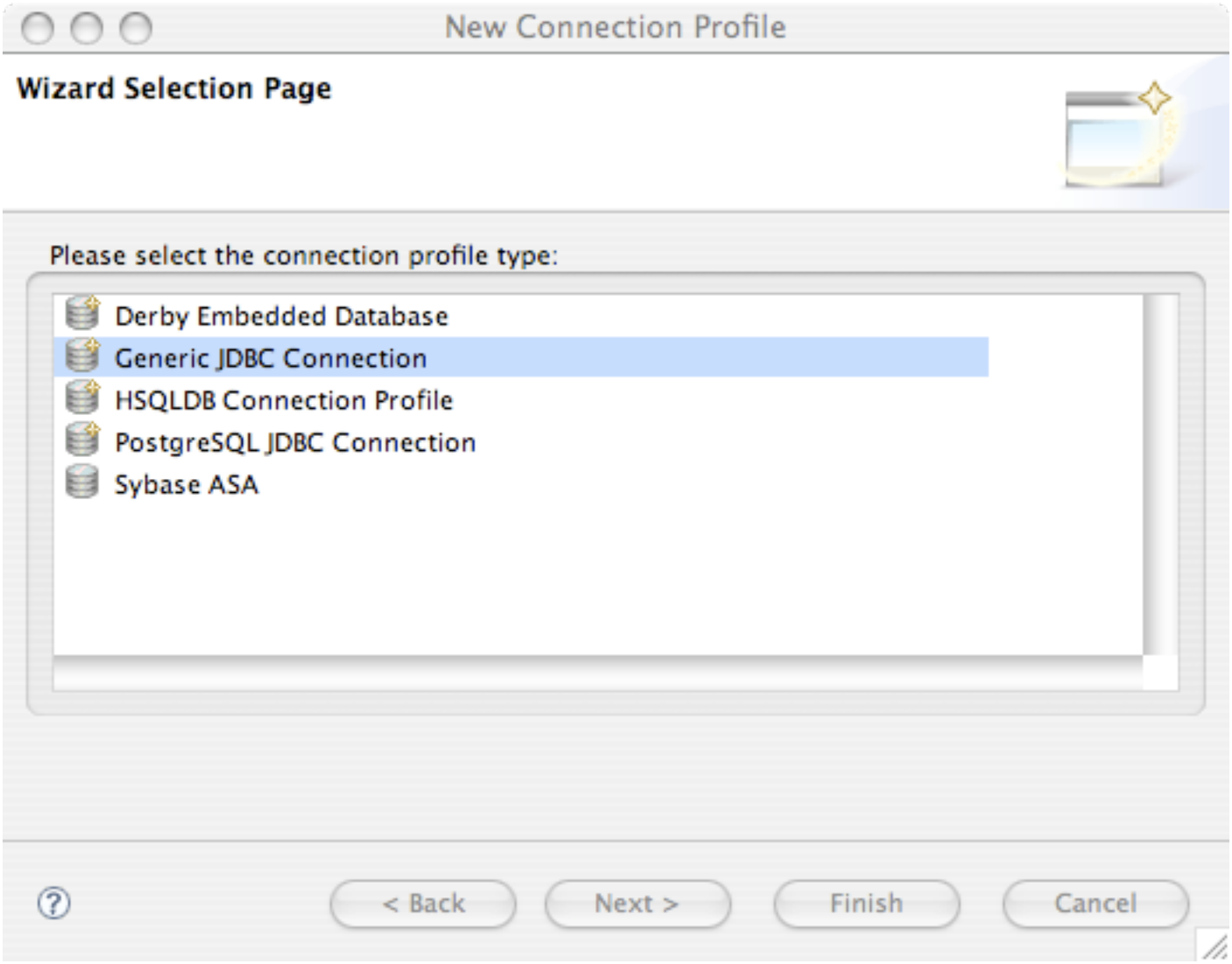
Le prossime 3 schermate consentono di personalizzare ulteriormente il proprio progetto, ma per noi i valori di default vanno bene. Si prema *Next* fino ad arrivare alla schermata finale.

Il primo passo è dire a JBoss Tools quale download di Seam si vuole usare. *Aggiungere* un nuovo *Seam Runtime* - assicurarsi di dare un nome, e selezionare *2.0* come versione:

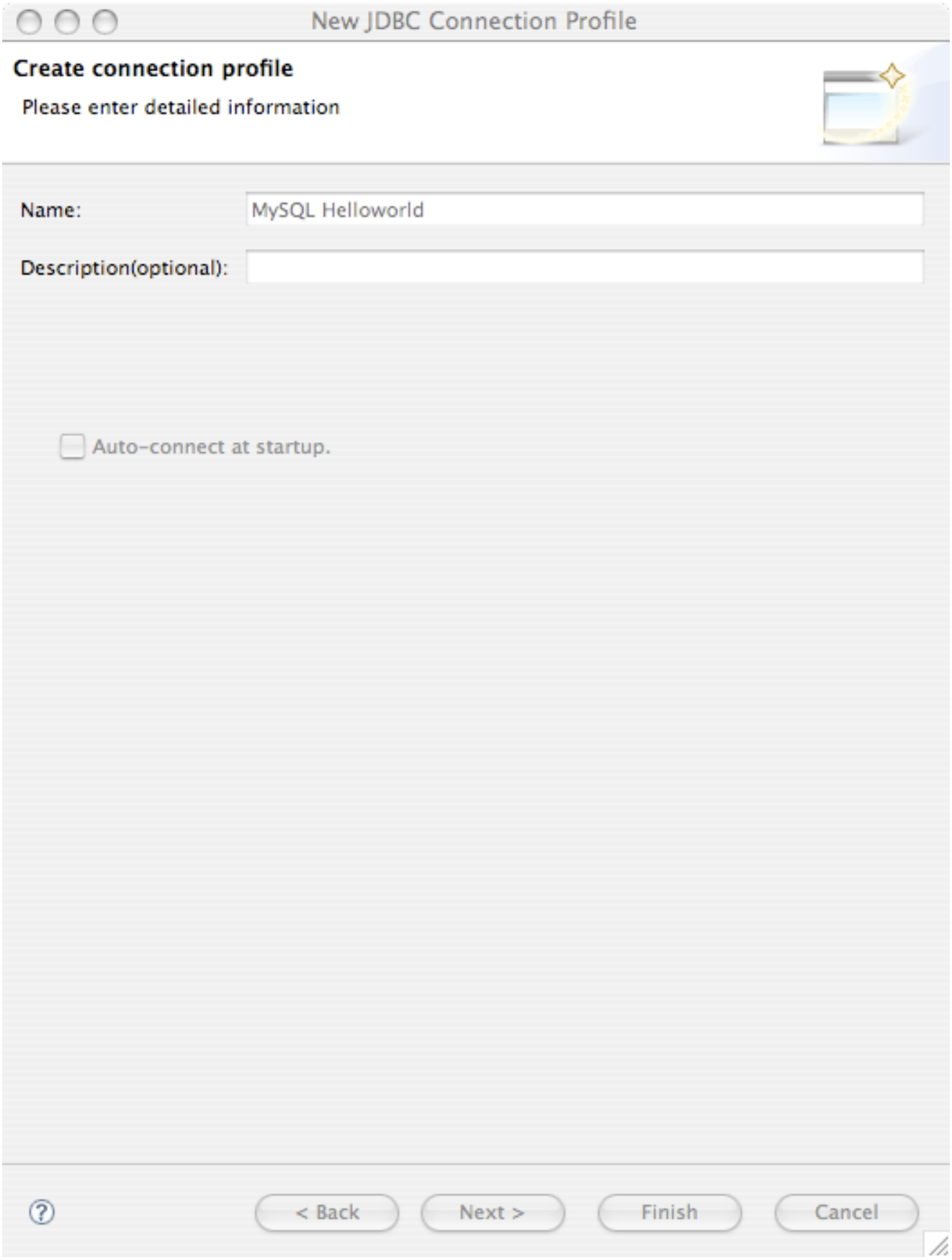


La scelta più importante da fare è tra deploy EAR e deploy WAR del proprio progetto. I progetti EAR supportano EJB 3.0 e richiede Java EE 5. I progetti WAR non supportano EJB 3.0, ma possono essere deployati in ambiente J2EE. Anche l'impacchettamento di un WAR è semplice da capire. Se si è installato un application server pronto per EJB3 come JBoss, si scelga *EAR*. Altrimenti, si scelga *WAR*. Assumeremo per il resto del tutorial che si sia scelto il deploy WAR, ma si possono seguire gli stessi passi per un deploy EAR.

Poi, si selezioni il tipo di database. Assumeremo di avere installato MySQL, con uno schema esistente. Occorrerà dire a JBoss Tools del database, selezionare *MySQL* come database, e creare un nuovo profilo di connessione. Selezionare *Generic JDBC Connection*:

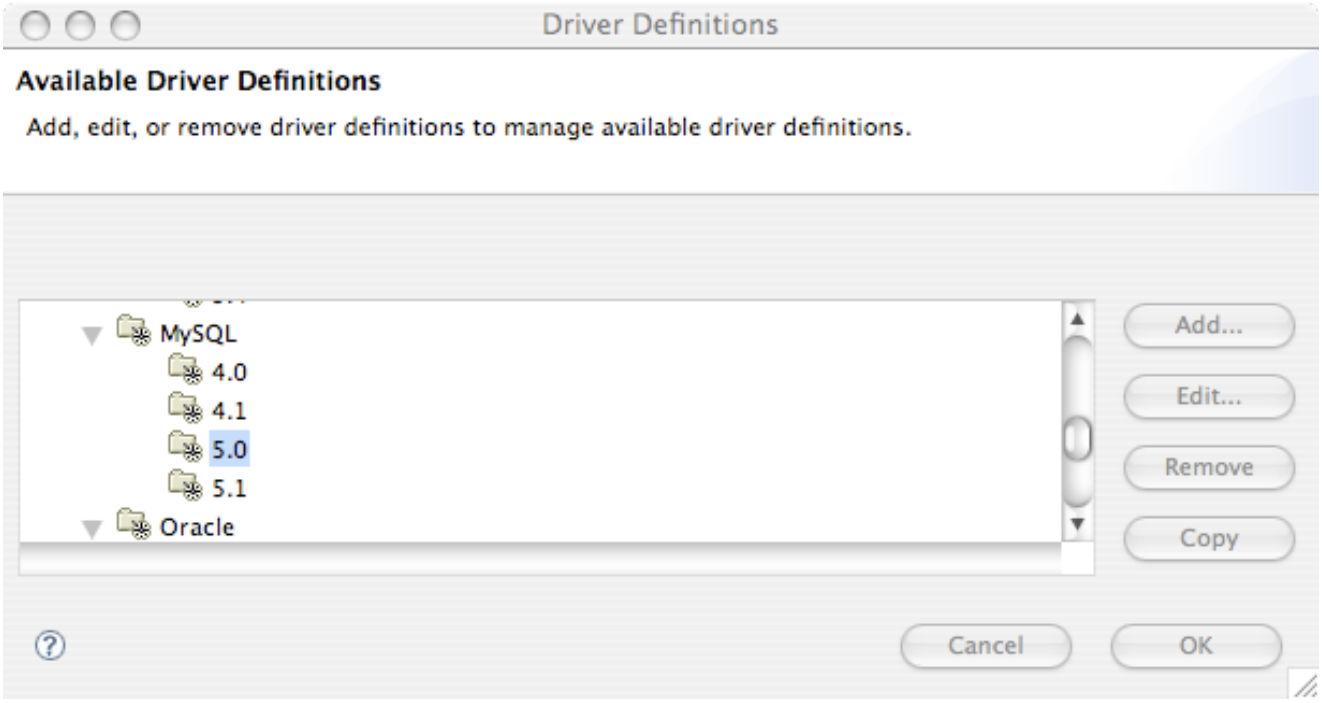


Si metta un nome:

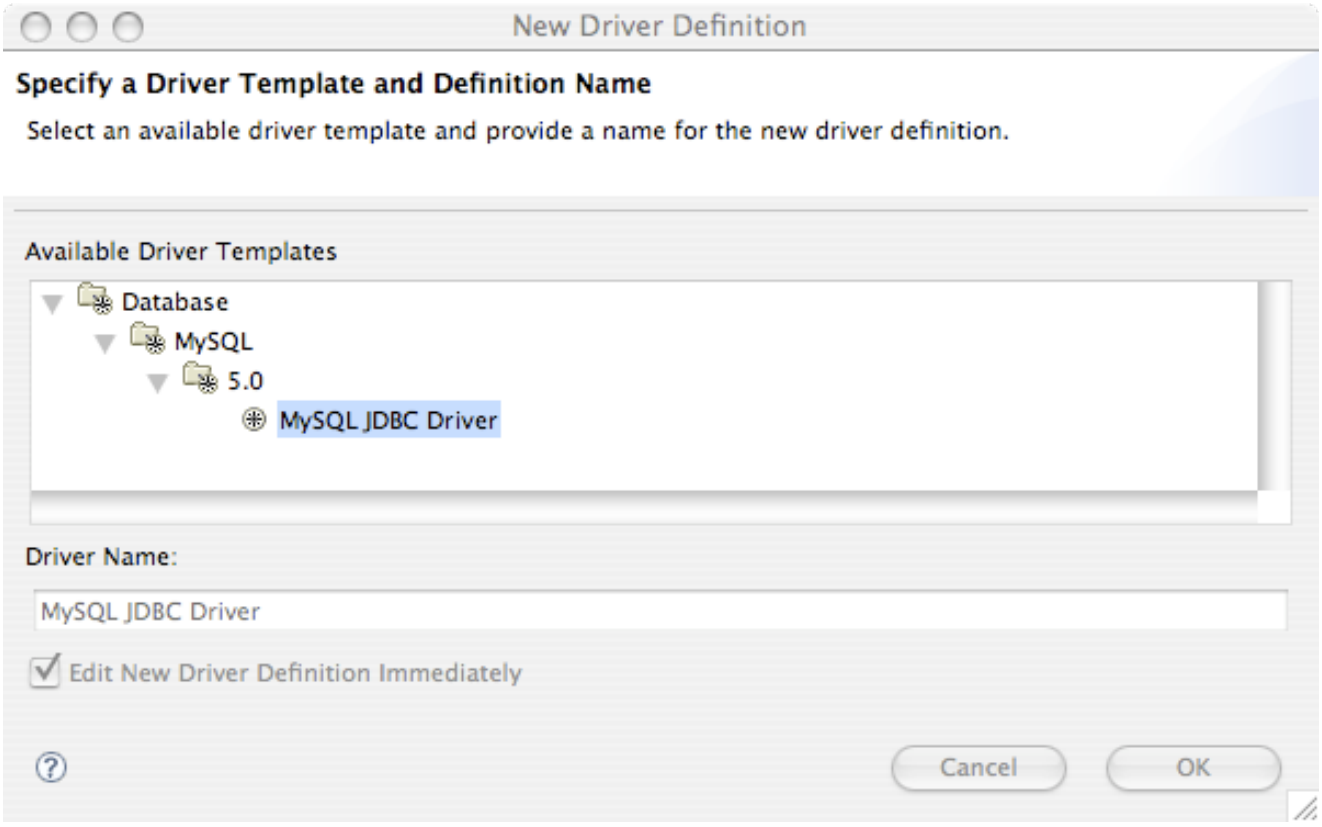

New JDBC Connection Profile
Create connection profile
Please enter detailed information
Name:
MySQL Helloworld
Description(optional):
Auto-connect at startup.
< Back
Next >
Finish
Cancel

JBoss Tools non è fornito assieme ai driver dei database, quindi occorre specificare dove è collocato il driver MySQL JDBC. Si prema il pulsante ....

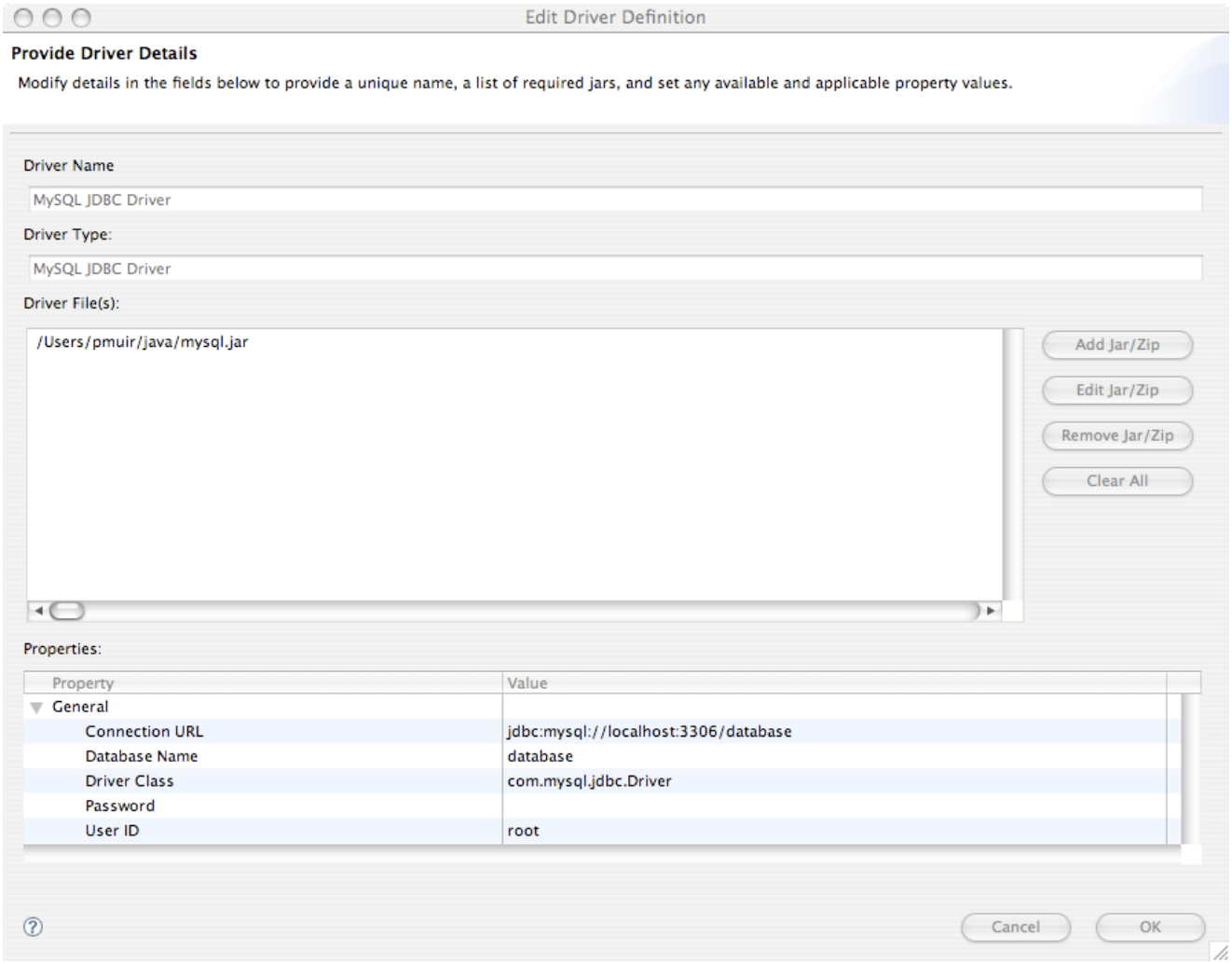
Trovare MySQL 5 e premere *Add...*:



Scegliere il template *MySQL JDBC Driver*:

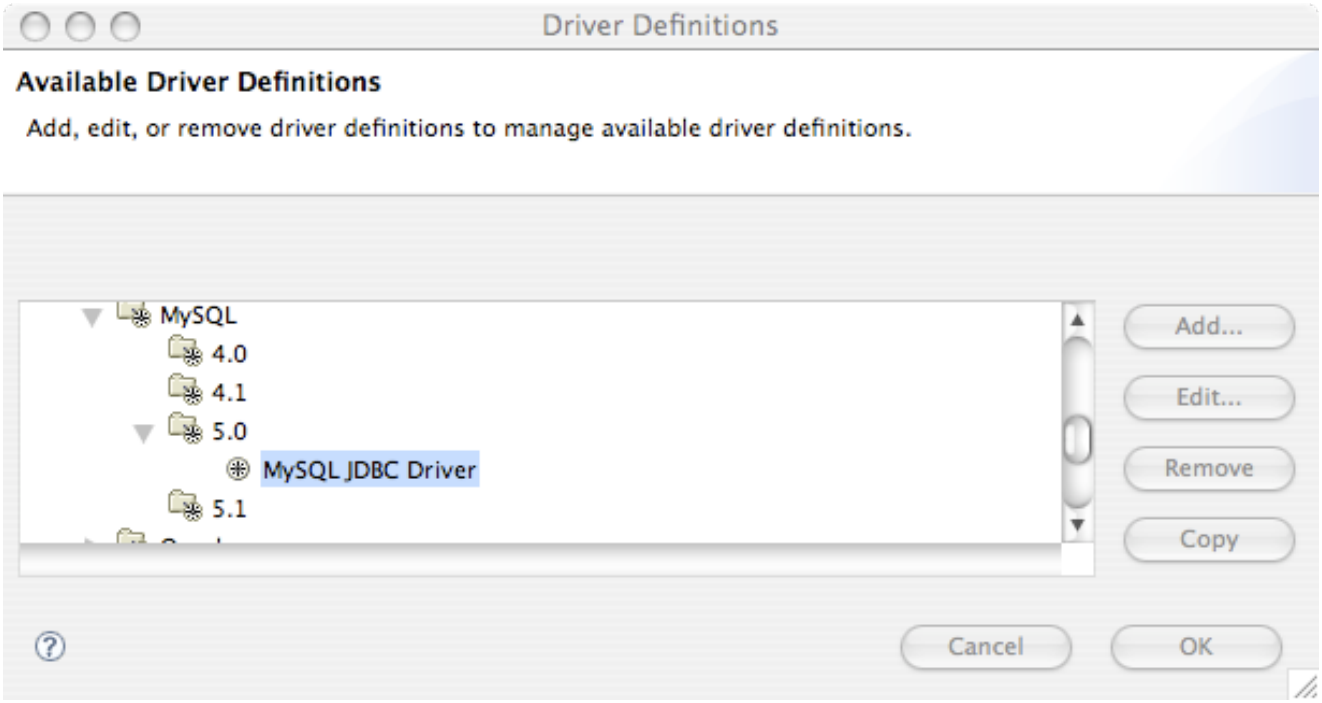


Trovare il jar sul proprio computer scegliendo *Edit Jar/Zip*:



Riguardare lo username e la password usate per la connessione, e se corretti, premere *Ok*.

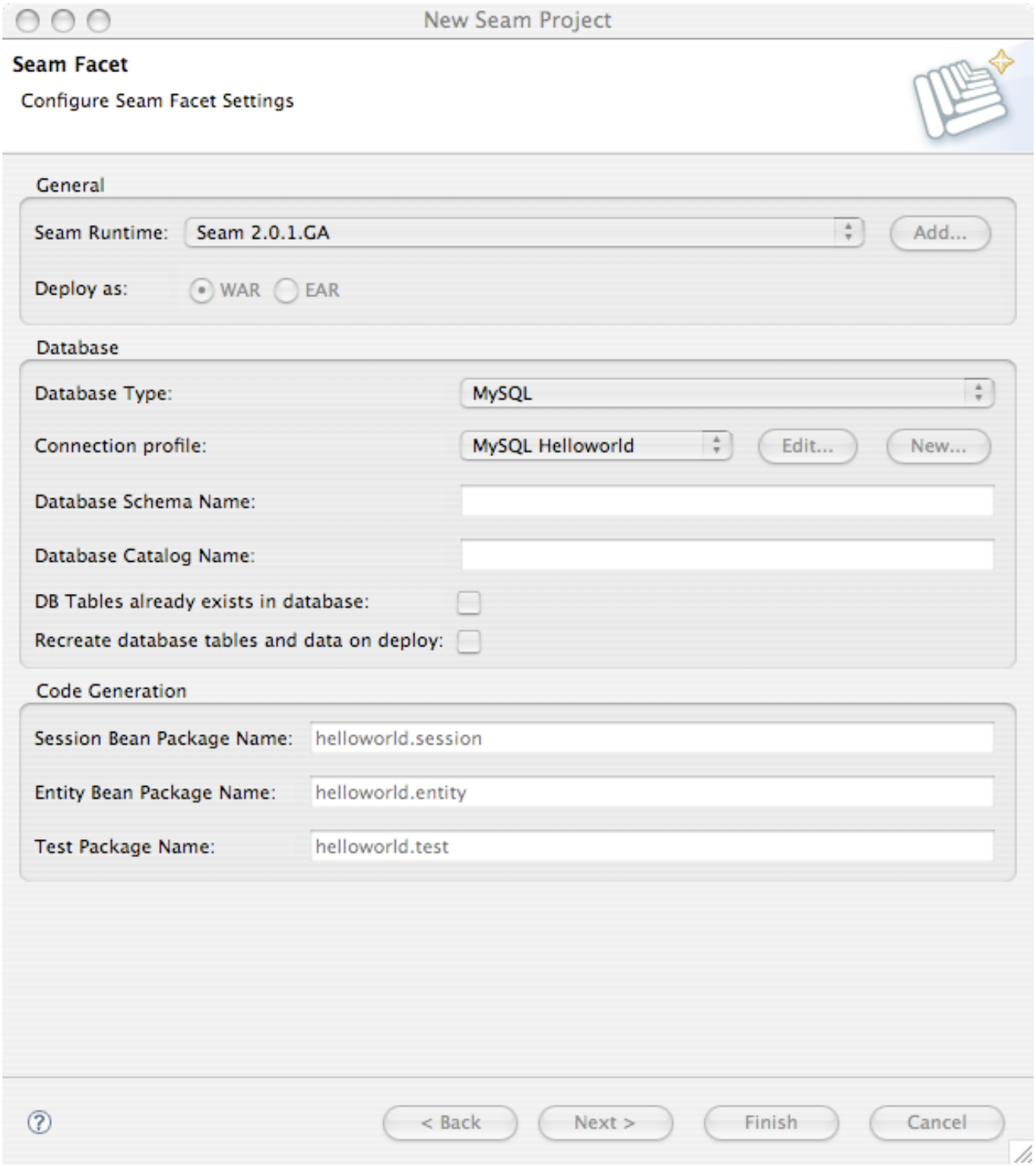
Infine scegliere il driver appena creato:



Se si sta lavorando con un modello di dati esistenti, assicurarsi di segnalare a JBoss Tools che le tabelle esistono già nel database.

Riguardare lo username e la password usate per la connessione, testare la connessione usando il pulsante *Test Connection*, e se funzionante, premere *Finish*:

Infine, rivedere i nomi dei pacchetti per i bean generati, e se si è soddisfatti, cliccare su *Finish*:



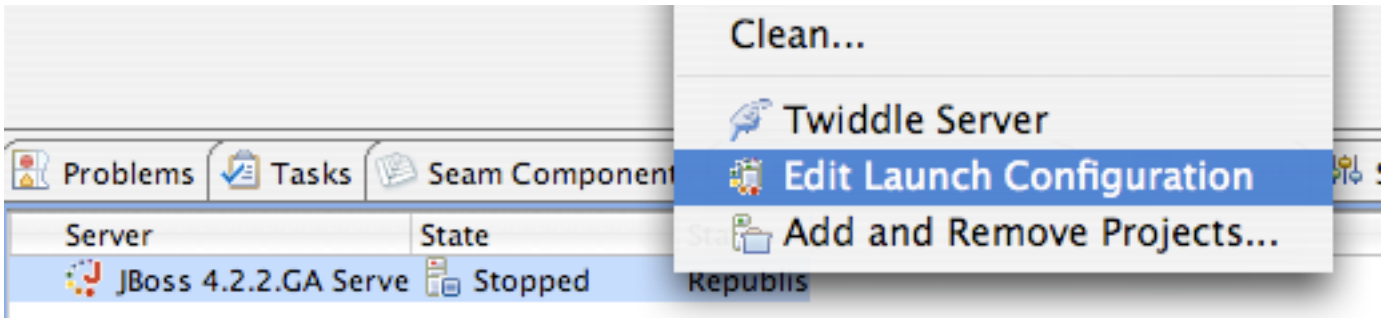
JBoss ha un supporto molto sofisticato per il redeploy a caldo (hot deploy) di WAR e EAR. Sfortunatamente, a causa di alcuni bug nella JVM, ripetuti redeploy di un EAR - comuni durante lo sviluppo - possono portare eventualmente ad errori di perm gen space nella JVM. Per questa ragione, raccomandiamo di eseguire JBoss in fase di sviluppo dentro una JVM con un ampio perm gen space. Suggeriamo i seguenti valori:

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

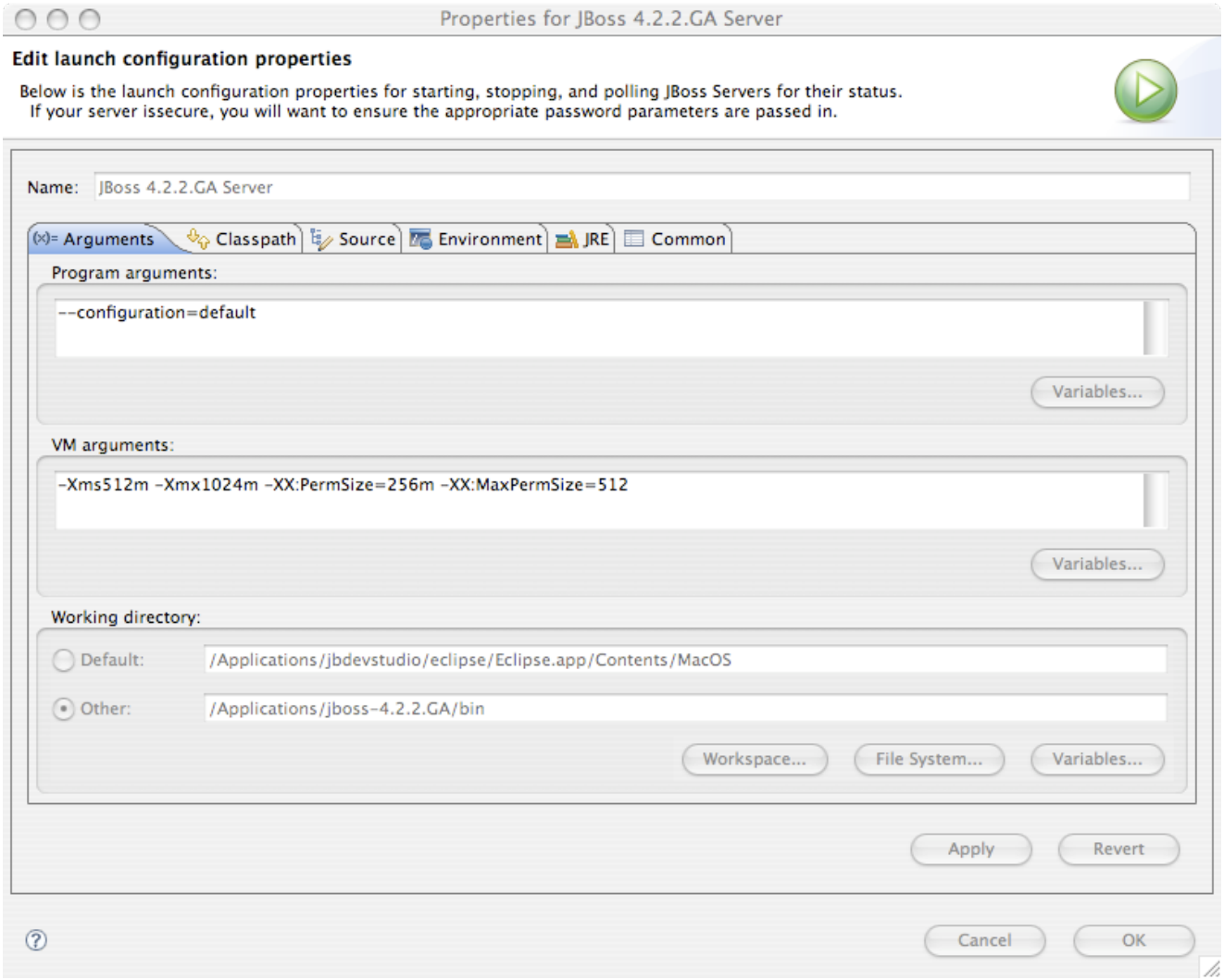
Se la memoria disponibile è poca, questa è la quantità minima suggerita:

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

Trovare il server in *JBoss Server View*, cliccare col tasto destro sul server e selezionare *Edit Launch Configuration*:

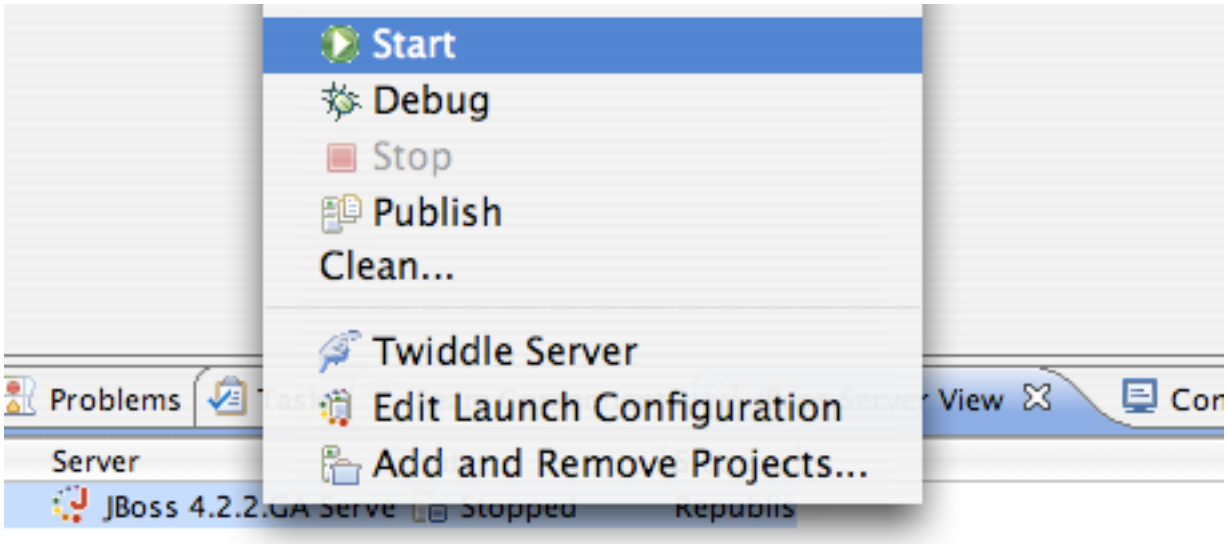


Poi si cambino gli argomenti VM:



Se non si vuole avere a che fare con queste configurazioni, non occorre farlo - si ritorni in questa sezione quando si vedrà la prima `OutOfMemoryException`.

Per avviare JBoss, e fare il deploy del progetto, cliccare col destro sul server creato e cliccare quindi *Start*, (o *Debug* per avviare in modalità Debug):

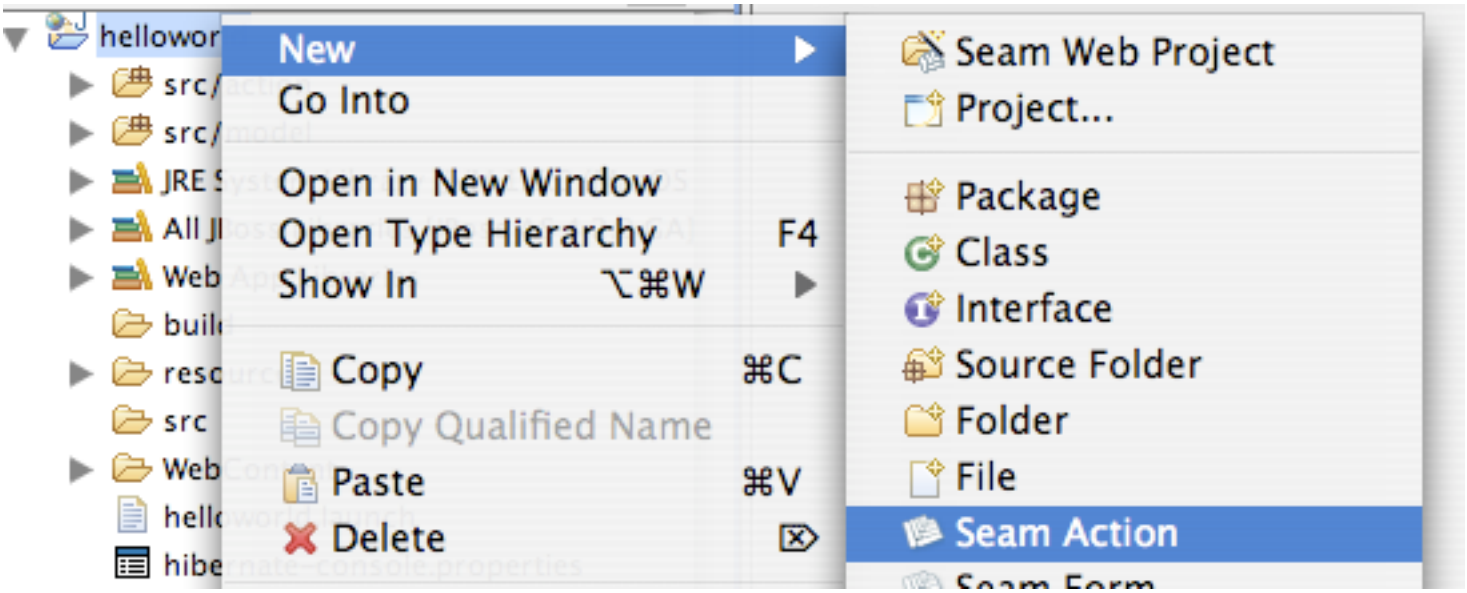


Non si abbia paura dei documenti di configurazione in XML che sono stati creati nella directory di progetto. La maggior parte riguardano Java EE, e sono creati la prima volta e poi non vengono più toccati. Al 90% sono sempre gli stessi nei vari progetti Seam.

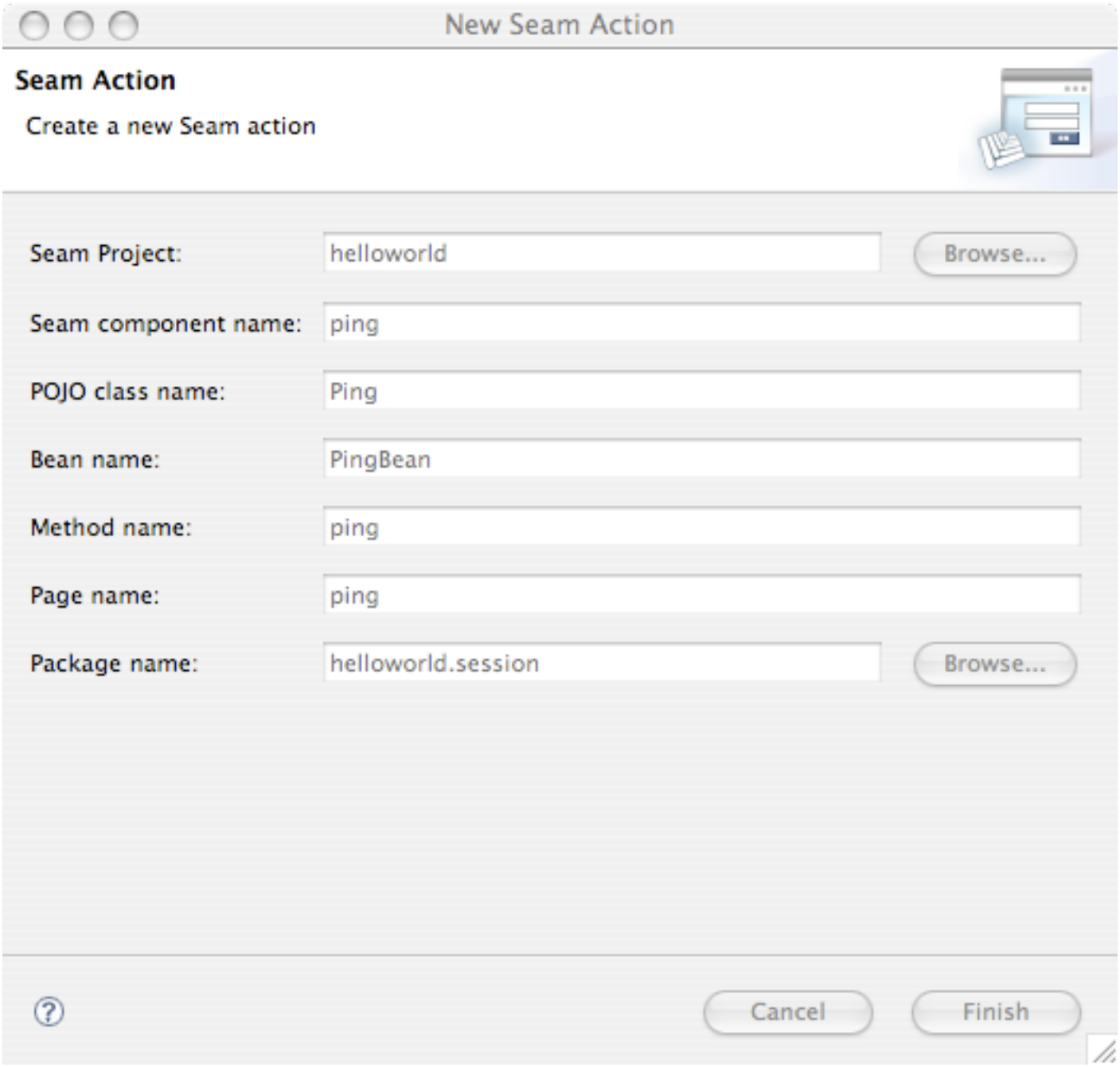
3.3. Creazione di una nuova azione

Se si è abituati ai tradizionali framework web action-style, probabilmente ci si sta domandando come si crea una semplice pagina web con un metodo d'azione stateless in Java.

Innanzitutto selezionare *New -> Seam Action*:



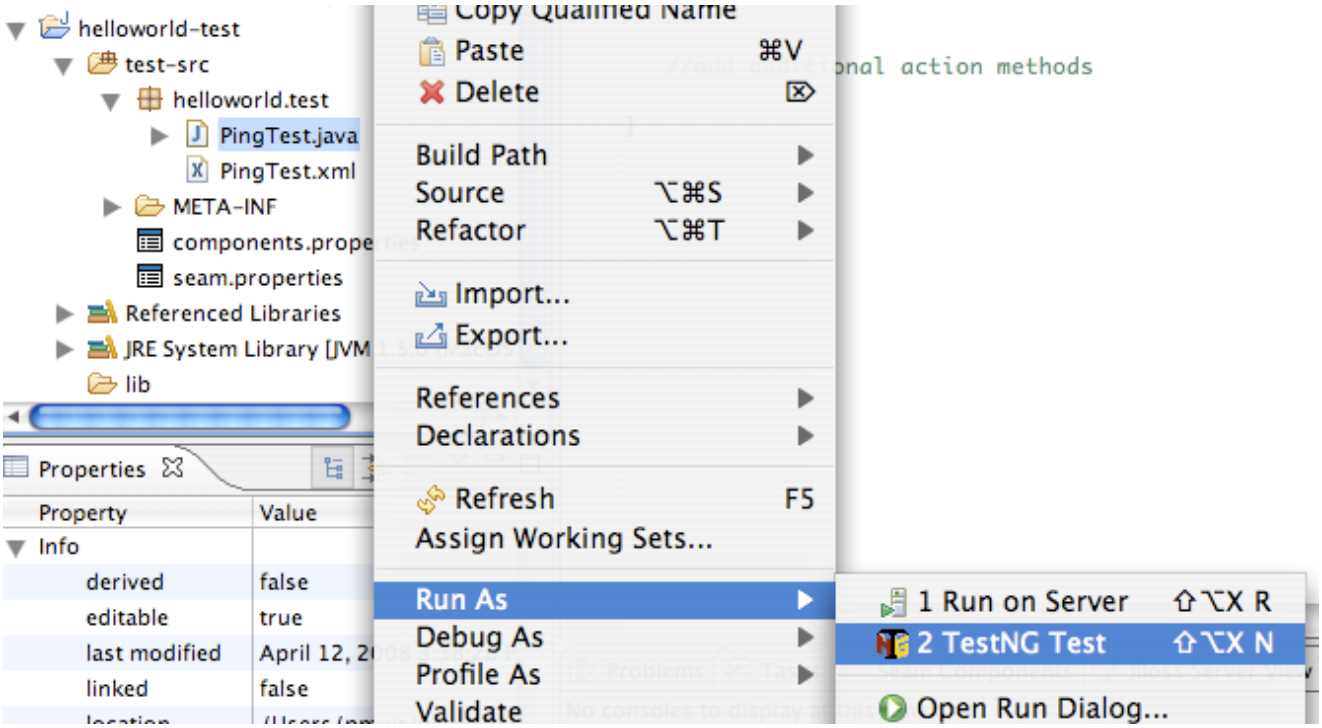
Ora, si inserisca il nome del componente Seam. JBoss Tools seleziona i valori di default sensibili per gli altri campi:



Infine di prema *Finish*.

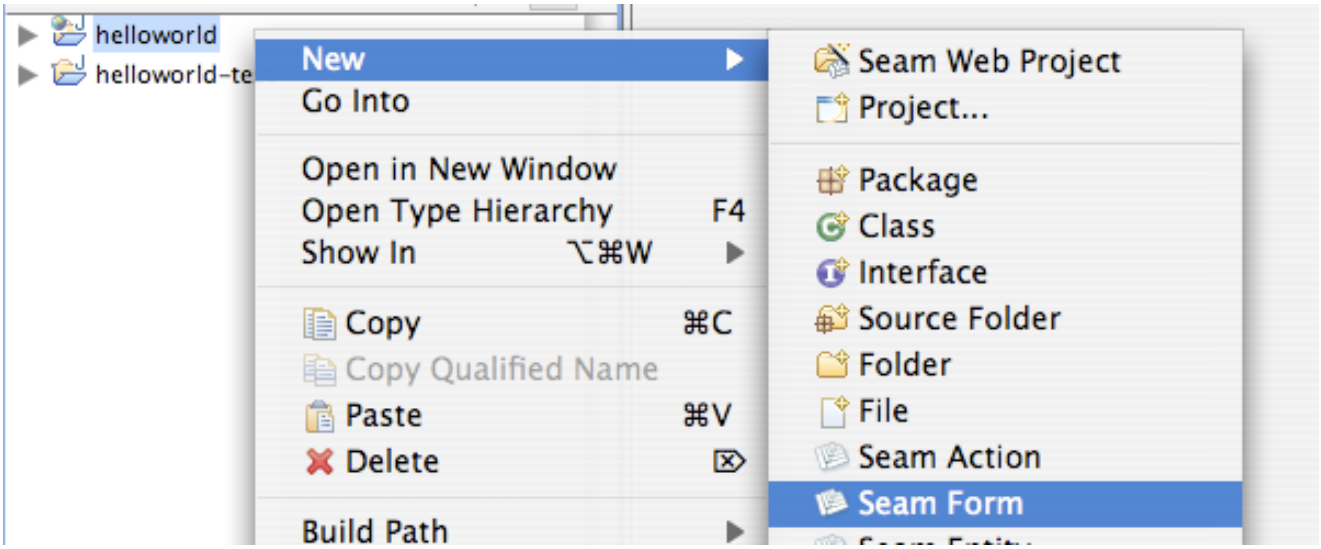
Ora si vada in `http://localhost:8080/helloworld/ping.seam` e si clicchi il pulsante. Si può vedere il codice dietro l'azione guardando nel progetto alla directory `src`. Si metta un breakpoint nel metodo `ping()`, e si clicchi di nuovo il pulsante.

Infine, si apra il progetto `helloworld-test`, si cerchi la classe `PingTest`, si clicchi su di essa col tasto destro e si scelga *Run As -> TestNG Test*:

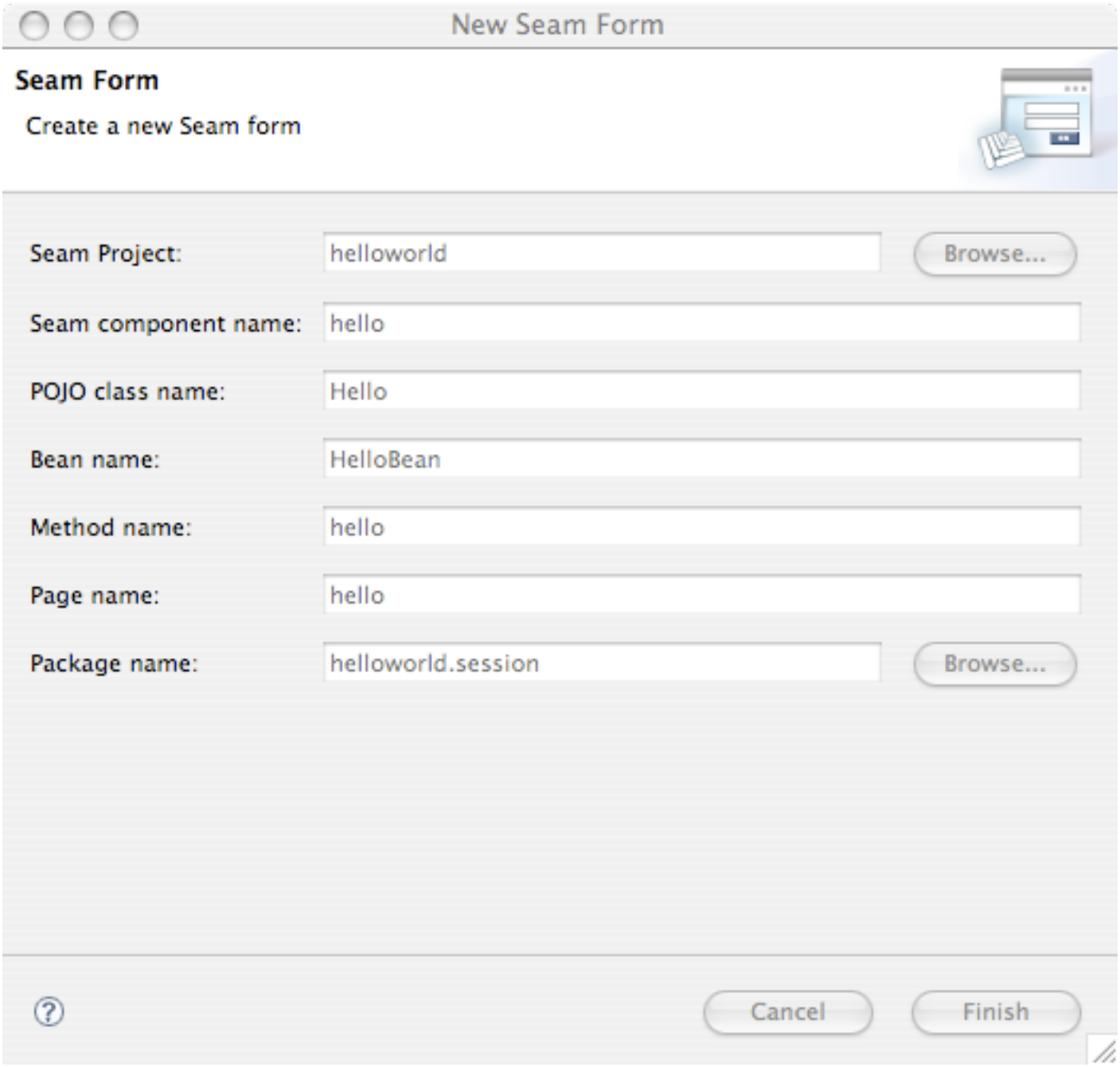


3.4. Creazione di una form con un'azione

Il primo passo è creare una form. Selezionare *New -> Seam Form*:



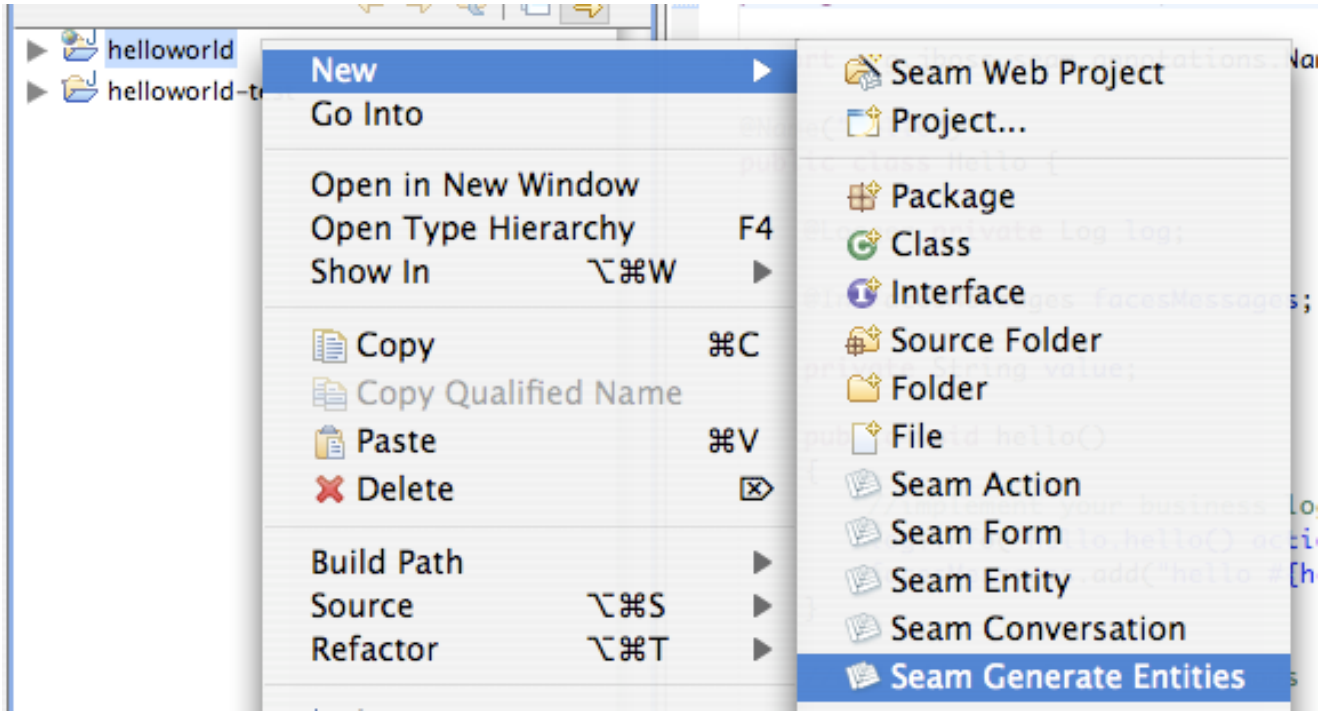
Ora, si inserisca il nome del componente Seam. JBoss Tools seleziona i valori di default sensibili per gli altri campi:



Si vada in `http://localhost:8080/helloworld/hello.seam`. Quindi si guardi il codice generato. Si esegua il test. Provare ad aggiungere alcuni nuovi campi nella form e nel componente Seam (si noti che non serve riavviare l'application server ogni volta che cambia il codice in `src/action` poiché Seam ricarica a caldo il componente per voi, vedere *Sezione 3.6, «Seam ed l'hot deploy incrementale con JBoss Tools»*).

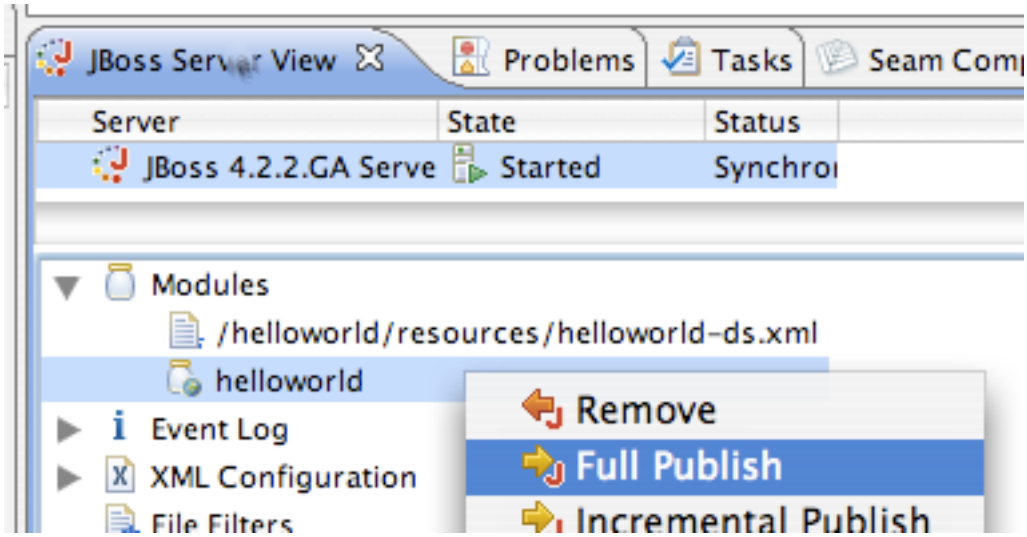
3.5. Generare un'applicazione da un database esistente

Si creino manualmente alcune tabelle nel database. (Se serve passare ad un database differente, si crei un nuovo progetto e si selezioni il database corretto). Poi, selezionare *New -> Seam Generate Entities*:



JBoss Tools fornisce l'opzione per eseguire il reverse engineering di entità, componenti, e viste da uno schema di database o da entità JPA esistenti. Ora eseguiremo un *Reverse engineering da database*.

Riavviare il deploy:



Andare in `http://localhost:8080/helloworld`. Si può sfogliare il database, modificare gli oggetti e creane di nuovi. Se si guarda il codice generato, probabilmente ci si meraviglierà della

sua semplicità! Seam è stato progettato per rendere semplice la scrittura del codice, anche per persone che non vogliono imbrogliare usando il reverse engineering.

3.6. Seam ed l'hot deploy incrementale con JBoss Tools

JBoss Tools supporta il deploy a caldo incrementale di:

- qualsiasi pagina facelets
- qualsiasi file `pages.xml`

out of the box.

Ma se si volesse cambiare il codice Java, servirebbe eseguire un riavvio completo dell'applicazione facendo un *Full Publish*.

Ma sesi vuole veramente un ciclo veloce modifica/compila/testa, Seam supporta il redeploy incrementale dei componenti JavaBean. Per usare questa funzionalità, occorre fare il deploy dei componenti JavaBean nella directory `WEB-INF/dev`, affinché vengano caricati da uno speciale classloader di Seam, anziché dal classloader dei WAR e EAR.

Occorre essere consapevoli delle seguenti limitazioni:

- i componenti devono essere componenti JavaBean, non possono essere EJB3 (stiamo lavorando per risolvere questa limitazione)
- gli entity non possono mai essere deployati a caldo
- i componenti deployati via `components.xml` non possono essere deployati a caldo
- i componenti hot-deployabili non saranno visibili alle classi deployate fuori da `WEB-INF/dev`
- La modalità debug di Seam deve essere abilitata e `jboss-seam-debug.jar` deve essere in `WEB-INF/lib`
- Occorre avere il filtro Seam installato in web.xml
- Si potrebbero vedere errori se il sistema è sotto carico ed è abilitato il debug.

Sesi crea un progetto WAR usando JBoss Tools, l'hot deploy incrementale è disponibile di default per le classi contenute nella directory sorgente `src/action`. Comunque, JBoss Tools non supporta il deploy incrementale a caldo per progetti EAR.

Il modello a componenti contestuali

I due concetti di base in Seam sono la nozione di *contesto* e la nozione di *componente*. I componenti sono oggetti stateful, solitamente EJB, e un'istanza di un componente viene associata al contesto, con un nome in tale contesto. La *bijection* fornisce un meccanismo per dare un alias ai nomi dei componenti interni (variabili d'istanza) associati ai nomi dei contesti, consentendo agli alberi dei componenti di essere dinamicamente assemblati e riassemblati da Seam.

Segue ora la descrizione dei contesti predefiniti in Seam.

4.1. Contesti di Seam

I contesti di Seam vengono creati e distrutti dal framework. L'applicazione non controlla la demarcazione dei contesti tramite esplicite chiamate dell'API Java. I contesti sono solitamente impliciti. In alcuni casi, comunque, i contesti sono demarcati tramite annotazioni.

I contesti base di Seam sono:

- contesto Stateless
- contesto Evento (cioè, richiesta)
- contesto Pagina
- contesto Conversazione
- contesto Sessione
- contesto processo di Business
- contesto Applicazione

Si riconosceranno alcuni di questi contesti dai servlet e dalle relative specifiche. Comunque due di questi potrebbero risultare nuovi: *conversation context*, e *business process context*. La gestione dello stato nelle applicazioni web è così fragile e propenso all'errore che i tre contesti predefiniti (richiesta, sessione ed applicazione) non sono significativi dal punto di vista della logica di business. Una sessione utente di login, per esempio, è praticamente un costrutto arbitrario in termini di workflow dell'applicazione. Quindi la maggior parte dei componenti Seam hanno scope nei contesti di conversazione e business process, poiché sono i contesti più significativi in termini di applicazione.

Ora si analizza ciascun contesto.

4.1.1. Contesto Stateless

I componenti che sono stateless (in primo luogo bean di sessione stateless) vivono sempre nel contesto stateless (che è sostanzialmente l'assenza di un contesto poiché l'istanza che Seam risolve non è memorizzata). I componenti stateless non sono molto interessanti e sono non molto

object-oriented. Tuttavia, essi vengono sviluppati e usati e sono quindi una parte importante di un'applicazione Seam.

4.1.2. Contesto Evento

Il contesto evento è il contesto stateful "più ristretto" ed è una generalizzazione della nozione del contesto di richiesta web per coprire gli altri tipi di eventi. Tuttavia, il contesto evento associato al ciclo di vita di una richiesta JSF è l'esempio più importante di un contesto evento ed è quello con cui si lavorerà più spesso. I componenti associati al contesto evento vengono distrutti alla fine della richiesta, ma il loro stato è disponibile e ben-definito per almeno il ciclo di vita della richiesta.

Quando si invoca un componente Seam via RMI, o Seam Remoting, il contesto evento viene creato e distrutto solo per l'invocazione.

4.1.3. Contesto Pagina

Il contesto pagina consente di associare lo stato con una particolare istanza di una pagina renderizzata. Si può inizializzare lo stato nell'event listener, o durante il rendering della pagina, e poi avere ad esso accesso da qualsiasi evento che ha origine dalla pagina. Questo è utile per funzionalità quali le liste cliccabili, dove dietro alla lista sono associati dati che cambiamo lato server. Lo stato è in verità serializzato al client, e quindi questo costrutto è estremamente robusto rispetto alle operazioni multi-finestra e al pulsante indietro.

4.1.4. Contesto Conversazione

Il contesto conversazione è un concetto fondamentale in Seam. Una *conversazione* è una unità di lavoro dal punto di vista dell'utente. Può dar vita a diverse interazioni con l'utente, diverse richieste, e diverse transazioni di database. Ma per l'utente, una conversazione risolve un singolo problema. Per esempio, "Prenota hotel", "Approva contratto", "Crea ordine" sono tutte conversazioni. Si può pensare alla conversazione come all'implementazione di un singolo "caso d'uso" o "user story", ma la relazione non è esattamente uguale.

Una conversazione mantiene lo stato associato a "cosa l'utente sta facendo adesso, in questa finestra". Un singolo utente potrebbe avere più conversazioni in corso in ogni momento, solitamente in più finestre. Il contesto conversazione assicura che lo stato delle diverse conversazioni non collida e non causi problemi.

Potrebbe volerci un pò di tempo prima di abituarsi a pensare applicazioni in termini di conversazione, ma una volta abituati, pensiamo che ci si appassionerà e non si riuscirà più a non pensare in altri termini!

Alcune conversazioni durano solo una singola richiesta. Le conversazioni che si prolungano attraverso più richieste devono essere marcate usando le annotazioni previste da Seam.

Alcune conversazioni sono anche *task*. Un task è una conversazione che è significativa in termini di processo di business long-running, ed ha il potenziale per lanciare una transizione di stato per il processo di business quando completa con successo. Seam fornisce uno speciale set di annotazioni per la demarcazione dei task.

Le conversazioni possono essere *inestate*, con una conversazione che ha posto "dentro" una conversazione più ampia. Questa è una caretteristica avanzata.

Solitamente lo stato della conversazione è mantenuto da Seam in una sessione servlet tra le richieste. Seam implementa dei *timeout di conversazione* configurabili, che automaticamente distruggono le conversazioni inattive, e quindi assicurano che lo stato mantenuto da una singola sessione utente non cresca senza limiti se l'utente abbandona le conversazioni.

Seam serializza il processo delle richieste concorrenti che prendono posto nello stesso contesto di conversazione long-running, nello stesso processo.

In alternativa Seam può essere configurato per mantenere lo stato conversazionale nel browser.

4.1.5. Contesto Sessione

Un contesto di sessione mantiene lo stato associato alla sessione utente. Mentre ci sono alcuni casi in cui è utile condividere lo stato tra più conversazioni, noi disapproviamo l'uso dei contesti di sessione per mantenere altri componenti diversi da quelli contenenti le informazioni globali sull'utente connesso.

In ambiente portal JSR-168 il contesto sessione rappresenta la sessione portlet.

4.1.6. Contesto processo di Business

Il contesto business process mantiene lo stato associato al processo di business long running. Questo stato è gestito e reso persistente dal motore BPM (JBoss jBPM). Il processo di business si prolunga attraverso più interazioni con diversi utenti, quindi questo stato è condiviso tra più utenti, ma in maniera ben definita. Il task corrente determina l'istanza corrente del processo di business, ed il ciclo di vita del processo di business è definito esternamente usando un *linguaggio di definizione di processo*, quindi non ci sono speciali annotazioni per la demarcazione del processo di business.

4.1.7. Contesto Applicazione

Il contesto applicazione è il familiare contesto servlet da specifiche servlet. Il contesto applicazione è principalmente utile per mantenere le informazioni statiche quali dati di configurazione, dati di riferimento i metamodelli. Per esempio, Seam memorizza la sua configurazione ed il metamodello nel contesto applicazione.

4.1.8. Variabili di contesto

Un contesto definisce un namespace, un set di *variabili di contesto*. Queste lavorano come gli attributi di sessione o richiesta nella specifica servlet. Si può associare qualsiasi valore si voglia alla variabile di contesto, ma solitamente si associano le istanze componenti di Seam alle variabili di contesto.

Quindi all'interno di un contesto un'istanza di componente è identificata dal nome della variabile di contesto (questo è solitamente, ma non sempre, lo stesso del nome del componente). Si

può accedere in modo programmatico all'istanza del componente con nome in un particolare scope tramite la classe `Contexts`, che fornisce accesso a parecchie istanze legate al thread dell'interfaccia `Context`:

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

Si può anche impostare o cambiare il valore associato al nome:

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

Solitamente, comunque, si ottengono i componenti da un contesto via injection e si mettono le istanza in un contesto via outjection.

4.1.9. Priorità di ricerca del contesto

A volte, come sopra, le istanze di un componente sono ottenute da un particolare scope noto. Altre volte, vengono ricercati tutti gli scope stateful in *ordine di priorità*. Quest'ordine è il seguente:

- Contesto Evento
- contesto Pagina
- contesto Conversazione
- contesto Sessione
- contesto processo di Business
- contesto Applicazione

Si può eseguire una ricerca prioritaria chiamando `Contexts.lookupInStatefulContexts()`. Quando si accede ad un componente via nome da una pagina JSF, serve una priorità di ricerca.

4.1.10. Modello di concorrenza

Né il servlet né le specifiche EJB definiscono dei modi per gestire le richieste correnti originate dallo stesso client. Il servlet container semplicemente lascia girare tutti i thread in modo concorrente e lascia la gestione della sicurezza dei thread al codice dell'applicazione. Il container EJB consente di accedere ai componenti stateless e lancia un'eccezione se dei thread multipli accedono ad un bean di sessione stateful.

Questo comportamento potrebbe risultare corretto nel vecchio stile delle applicazioni web, che erano basate su richieste sincrone con granularità fine. Ma per le moderne applicazioni che fanno ampio uso di molte richieste asincrone (AJAX) a granularità fine, la concorrenza è un fattore vitale e deve essere supportato dal modello di programmazione. Seam possiede un layer per la gestione della concorrenza nel suo modello di contesto.

I contesti Seam di sessione e applicazione sono multithread. Seam lascia le richieste concorrenti in un contesto che verrà processato in modo concorrente. I contesti evento e pagina sono per natura a singolo thread. Il contesto business è strettamente multithread, ma in pratica la concorenza è abbastanza rara e questo fatto può essere ignorato per la maggior parte delle volte. Infine Seam forza un modello a *singolo thread per conversazione per processo* per il contesto conversazione, serializzando le richieste concorrenti nello stesso contesto di conversazione long-running.

Poiché il contesto sessione è multithread, e spesso contiene uno stato volatile, i componenti con scope sessione sono sempre protetti da Seam verso accessi concorrenti fintantoché gli interceptor Seam non vengano disabilitati per quel componente. Se gli interceptor sono disabilitati, allora ogni sicurezza di thread che viene richiesta deve essere implementata dal componente stesso. Seam serializza di default le richieste a bean con scope sessione e JavaBean (e rileva e rompe ogni deadlock che sopravviene). Questo non è il comportamento di default per i componenti con scope applicazione, poiché tali componenti solitamente non mantengono uno stato volatile e poiché la sincronizzazione a livello globale è *estremamente* dispensiosa. Comunque si può forzare il modello di thread serializzato su un qualsiasi session bean o componente JavaBean aggiungendo l'annotazione `@Synchronized`.

Questo modello di concorrenza signifca che i client AJAX possono usare in modo sicuro le sessioni volatili e lo stato conversazionale, senza il bisogno di alcun lavoro speciale da parte dello sviluppatore.

4.2. Componenti di Seam

I componenti Seam sono POJO (Plain Old Java Objects). In particolare sono JavaBean o bean enterprise EJB 3.0. Mentre Seam non richiede che i componenti siano EJB e possono anche essere usati senza un container EJB 3.0, Seam è stato progettato con EJB 3.0 in mente ed realizza una profonda integrazione con EJB 3.0. Seam supporta i seguenti *tipi di componenti*.

- Bean di sessione stateless EJB 3.0
- Bean di sessione stateful EJB 3.0
- Bean entity EJB 3.0 (cioè classi entity JPA)
- JavaBeans
- EJB 3.0 message-driven beans
- Bean Spring (see *Capitolo 27, Integrazione con il framework Spring*)

4.2.1. Bean di sessione stateless

I componenti bean di sessione stateless non sono in grado di mantenere lo stato lungo le diverse invocazioni. Quindi solitamente lavorano operando sullo statto di altri componenti in vari contesti

Seam. Possono essere usati come action listener JSF, ma non forniscono proprietà ai componenti JSF da mostrare.

I bean di sessione stateless vivono sempre nel contesto stateless.

Si può accedere ai bean di sessione stateless in modo concorrente poiché viene usata una nuova istanza ad ogni richiesta. L'assegnazione di un'istanza alla richiesta è responsabilità del container EJB3 (normalmente le istanze vengono allocate da un pool riutilizzabile, ciò significa che si possono trovare variabili d'istanza contenenti dati da precedenti utilizzi del bean).

I bean di sessione stateless sono i tipi di componenti Seam meno interessanti.

I bean di sessione stateless Seam possono essere istanziati usando `Component.getInstance()` o `@In(create=true)`. Non dovrebbero essere istanziati direttamente tramite ricerca JNDI o tramite operatore `new`.

4.2.2. Bean di sessione stateful

I componenti bean di sessione stateful sono in grado di mantenere lo stato non solo lungo più invocazioni del bean, ma anche attraverso richieste multiple. Lo stato dell'applicazione che non appartiene al database dovrebbe essere mantenuto dai bean di sessione stateful. Questa è la grande differenza tra Seam e molti altri framework per applicazioni web. Invece di mettere informazioni sulla conversazione corrente direttamente nella `HttpSession`, si dovrebbe mantenerla nelle variabili d'istanza di un bean di sessione stateful, che è legato al contesto conversazione. Questo consente a Seam di gestire il ciclo di vita di questo stato per voi e di assicurare che non ci siano collisioni tra lo stato delle diverse conversazioni concorrenti.

I bean di sessione stateful sono spesso usati come action listener JSF ed in qualità di backing bean forniscono proprietà ai componenti JSF da mostrare o per l'invio di form.

Di default i bean di sessione stateful sono legati al contesto conversazione. Non possono mai essere legati ai contesti pagina o stateless.

Richieste concorrenti a bean di sessione stateful con scope sessione sono sempre serializzati da Seam fintantoché gli interceptor di Seam non vengono disabilitati per tale bean.

I bean di sessione stateful Seam possono essere istanziati usando `Component.getInstance()` o `@In(create=true)`. Non dovrebbero essere istanziati direttamente tramite ricerca JNDI o tramite operatore `new`.

4.2.3. Entity bean

Gli entity bean possono essere associati ad una variabile di contesto e funzionare come componenti Seam. Poiché gli entity hanno un'identità persistente in aggiunta alla loro identità contestuale, le istanze entity sono solitamente associate esplicitamente nel codice Java, piuttosto che essere istanziate implicitamente da Seam.

I componenti entity bean non supportano la bijection o la demarcazione di contesto. E neppure l'invocazione della validazione dell'entity bean (trigger).

Gli entity bean non sono solitamente usati come action listener JSF, ma spesso funzionano come backing bean che forniscono proprietà ai componenti JSF per la visualizzazione o la sottomissione di una form. In particolare è comune usare un entity come backing bean, assieme ad un action listener bean di sessione stateless per implementare funzionalità di tipo crea/aggiorna/cancella.

Di default gli entity bean sono associati al contesto conversazione. Non possono mai essere associati al contesto stateless.

Si noti che in un ambiente cluster è meno efficiente associare un entity bean direttamente ad una variabile di contesto con scope conversazione o sessione rispetto a come sarebbe mantenere un riferimento all'entity bean in un bean di sessione stateful. PEr questa ragione non tutte le applicazioni Seam definiscono entity bean come componenti Seam.

I componenti entity bean di Seam possono essere istanziati usando `Component.getInstance()`, `@In(create=true)` o direttamente usando l'operatore `new`.

4.2.4. JavaBeans

I Javabean possono essere usati solo come bean di sessione stateless o stateful. Comunque essi non forniscono la funzionalità di un session bean (demarcazione dichiarativa delle transazioni, sicurezza dichiarativa, replicazione clustered efficiente dello stato, persistenza EJB 3.0, metodi timeout, ecc.)

In un capitolo successivo si mostrerà come impiegare Seam ed Hibernate senza un container EJB. In questo caso i componenti sono JavaBean invece di session bean. Si noti comunque che in molti application server è talvolta meno efficiente clusterizzare la conversazione od i componenti Seam Javabean con scope sessione piuttosto che clusterizzare i componenti session bean stateful.

Di default i JavaBean sono legati al contesto evento.

Le richieste concorrenti a Javabean con scope sessione vengono sempre serializzate da Seam.

I componenti JavaBean di Seam possono essere istanziati usando `Component.getInstance()` o `@In(create=true)`. Non dovrebbero essere istanziati direttamente tramite operatore `new`.

4.2.5. Message-driven bean

I bean message-driven possono funzionare come componenti Seam. Comunque i bean message-driven sono chiamati in modo diverso rispetto agli altri componenti Seam - invece di invocarli tramite variabile di contesto, essi ascoltano i messaggi inviati ad una coda o topic JMS.

I bean message-driven possono non essere associati ad un contesto Seam. E possono non avere accesso allo stato di sessione o conversazione del loro "chiamante". Comunque essi non supportano la bijection e altre funzionalità di Seam.

I bean message-driven non vengono mai istanziati dall'applicazione. Essi vengono istanziati dal container EJB quando viene ricevuto un messaggio.

4.2.6. Intercettazione

Per eseguire le sue magie (bijection, demarcazione di contesto, validazione, ecc.) Seam deve intercettare le invocazioni dei componenti. Per i JavaBean, Seam è nel pieno controllo dell'istanziazione del componente e non ha bisogno di alcuna speciale configurazione. Per gli entity bean, l'intercettazione non è richiesta poiché bijection e demarcazione di contesto non sono definite. Per i session bean occorre registrare un interceptor EJB per il componente bean di sessione. Si può impiegare un'annotazione come segue:

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

Ma il miglio modo è definire l'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
>
```

4.2.7. Nomi dei componenti

Tutti i componenti Seam devono avere un nome. Si può assegnare un nome al componente usando l'annotazione `@Name`:

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

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

Questo nome è il *nome del componente Seam* e non è relazionato a nessun altro nome definito dalla specifica EJB. Comunque i nomi dei componenti Seam funzionano solo come nomi per i bean gestiti da JSF e si possono ritenere questi due concetti come identici.

`@Name` non è il solo modo per definire un nome di componente, ma occorre sempre specificare il nome *da qualche parte*. Altrimenti nessun'altra annotazione di Seam funzionerà.

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.

Per applicazioni estese e per i componenti predefiniti di seam, vengono spesso impiegati i nomi qualificati dei componenti per evitare conflitti di nome.

```
@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.com/products/seam/components">

   <import>org.jboss.seam.core</import>
   <import>org.jboss.seam.cache</import>
   <import>org.jboss.seam.transaction</import>
   <import>org.jboss.seam.framework</import>
   <import>org.jboss.seam.web</import>
   <import>org.jboss.seam.faces</import>
   <import>org.jboss.seam.international</import>
   <import>org.jboss.seam.theme</import>
   <import>org.jboss.seam.pageflow</import>
   <import>org.jboss.seam.bpm</import>
   <import>org.jboss.seam.jms</import>
   <import>org.jboss.seam.mail</import>
   <import>org.jboss.seam.security</import>
   <import>org.jboss.seam.security.management</import>
   <import>org.jboss.seam.security.permission</import>
   <import>org.jboss.seam.captcha</import>
   <import>org.jboss.seam.excel.exporter</import>
   <!-- ... --->
</components>
```

When attempting to resolve an unqualified name, Seam will check each of those namespaces, in order. You can include additional namespaces in your application's `components.xml` file for application-specific namespaces.

4.2.8. Definire lo scope di un componente

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` definisce un'enumeration dei possibili scope.

4.2.9. Componenti con ruoli multipli

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 {
   ...
}
```

L'annotazione `@Roles` consente di specificare tanti ruoli quanti se ne vuole.

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

4.2.10. Componenti predefiniti

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.

I componenti predefiniti possono essere iniettati, come ogni altro componente Seam, ma possono anche fornire dei metodi statici `instance()` di convenienza.

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

4.3. 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;
   ...
}
```

o nel metodo setter:

```
@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`.

Si può anche iniettare il valore di un'espressione:

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

I valori iniettati sono disiniettati (cioè impostati a `null`) immediatamente dopo il completamento del metodo e dell'outjection.

(There is much more information about component lifecycle and injection in the next chapter.)

L'annotazione `@Out` specifica che occorre eseguire l'outjection di un attributo, oppure da una variabile d'istanza:

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

```
    @Out User user;
    ...
}
```

o dal metodo getter:

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

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

    ...
}
```

Di un attributo si può fare sia l'injection sia l'outjection:

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

o:

```
@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;
   }

   ...
}
```

4.4. Metodi del ciclo di vita

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={....})`.

4.5. Installazione condizionale

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 precendence. There are some predefined precedence values (in ascending order):

1. `BUILT_IN` — i componenti con più bassa precedenza sono i componenti predefiniti in Seam.
2. `FRAMEWORK` — i componenti definiti da framework di terze parti possono sovrascrivere i componenti predefiniti, ma vengono sovrascritti dai componenti applicazione.
3. `APPLICATION` — la precedenza di default. Questo è appropriato per i componenti delle applicazioni più comuni.
4. `DEPLOYMENT` — per i componenti applicazione che sono specifici per un deploy.
5. `MOCK` — per gli oggetti mock usati in fase di test.

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 dependecies, 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.

4.6. Logging

Chi non è nauseato dal vedere codice incasinato come questo?

```
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 fornisce un'API per il logging che semplifica in modo significativo questo codice:

```
@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);
}
```

Non importa se si dichiara la variabile `log` statica o no — funzionerà in entrambi i modi, tranne per i componenti entity bean che richiedono la variabile `log` statica.

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.

4.7. L'interfaccia `Mutable` e `@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;
   }

   ...

}
```

O si può usare l'annotazione `@ReadOnly` per ottenere un effetto simile:

```
@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.

4.8. Componenti factory e manager

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 ... ;
}
```

Il secondo stile è un metodo di tipo `void` che associa il valore alla variabile di contesto stessa:

```
@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 visable 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.

Configurare i componenti Seam

La filosofia di minimizzare la configurazione basata su XML è estremamente forte in Seam. Tuttavia ci sono varie ragioni per voler configurare i componente Seam tramite XML: per isolare le informazioni specifiche del deploy dal codice Java, per abilitare la creazione di framework riutilizzabili, per configurare le funzionalità predefinite di Seam, ecc. Seam fornisce due approcci base per configurare i componenti: configurazione tramire impostazioni di proprietà in un file di proprietà o in `web.xml`, e configurazione tramite `components.xml`.

5.1. Configurare i componenti tramire impostazioni di proprietà

I componenti Seam possono essere accompagnati da proprietà di configurazioni o via parametri di contesto servlet oppure tramite un file di proprietà chiamato `seam.properties` collocato nella radice del classpath.

Il componente Seam configurabile deve esporre metodi setter di stile JavaBeans per gli attributi configurabili. Se un componente Seam chiamato `com.jboss.myapp.settings` ha un metodo setter chiamato `setLocale()`, si può scrivere una proprietà chiamata `com.jboss.myapp.settings.locale` nel file `seam.properties` o come parametro di contesto servlet, e Seam imposterà il valore dell'attributo `locale` quando istanzia il componente.

Lo stesso meccanismo viene usato per configurare lo stesso Seam. Per esempio, per impostare il timeout della conversazione, si fornisce un valore a `org.jboss.seam.core.manager.conversationTimeout` in `web.xml` oppure in `seam.properties`. (C'è un componente Seam predefinito chiamato `org.jboss.seam.core.manager` con metodo setter chiamato `setConversationTimeout()`.)

5.2. Configurazione dei componenti tramite `components.xml`

Il file `components.xml` è un poco più potente delle impostazioni di proprietà. Esso consente di:

- Configurare i componenti installati automaticamente — inclusi entrambi i componenti predefiniti ed i componenti di applicazione che sono stati annotati con l'annotazione `@Name` e rilevati dallo scanner di deploy di Seam.
- Installare le classi senza annotazione `@Name` come componenti Seam — questo è ancora più utile per alcuni tipi di componenti infrastrutturali che possono essere installati diverse volte con diversi nomi (per esempio i contesti di persistenza gestiti da Seam).
- Installare componenti che *hanno* un'annotazione `@Name`, ma non vengono installati di default poiché una annotazione `@Install` indica che non devono essere installati.
- Override dello scope di un componente

Un file `components.xml` può apparire in una delle tre seguenti posizioni:

- Nella directory `WEB-INF` di un file `war`.
- Nella directory `META-INF` di un file `jar`.
- In una qualsiasi directory di un file `jar` che contenga classi con annotazione `@Name`.

Solitamente i componenti Seam vengono installati quando lo scanner di deploy scopre una classe con una annotazione `@Name` collocata in un archivio con un file `seam.properties` o un file `META-INF/components.xml`. (Amenoché il componente abbia una annotazione `@Install` che indichi che non debba essere installato di default). Il file `components.xml` consente di gestire i casi speciali in cui occorra fare override delle annotazioni.

Per esempio, il seguente file `components.xml` installa jBPM:

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

Questo esempio fa la stessa cosa:

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

Questo installa e configura due differenti contesti di persistenza gestiti da Seam:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:persistence="http://jboss.com/products/seam/persistence"

   <persistence:managed-persistence-context name="customerDatabase"
            persistence-unit-jndi-name="java:/customerEntityManagerFactory"/>

   <persistence:managed-persistence-context name="accountingDatabase"
            persistence-unit-jndi-name="java:/accountingEntityManagerFactory"/>

</components
>
```

Ed anche questo fa lo stesso:

```
<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
>
```

Questo esempio crea un contesto di persistenza gestito da Seam con scope di sessione (questa non è una pratica raccomandata):

```
<components xmlns="http://jboss.com/products/seam/components"
         xmlns:persistence="http://jboss.com/products/seam/persistence"

  <persistence:managed-persistence-context name="productDatabase"
                                 scope="session"
               persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components
>
```

```
<components>

   <component name="productDatabase"
         scope="session"
         class="org.jboss.seam.persistence.ManagedPersistenceContext">
      <property name="persistenceUnitJndiName"
>java:/productEntityManagerFactory</property>
   </component>

</components
```

```
>
```

E' comune utilizzare l'opzione `auto-create` per gli oggetti infrastrutturali quali i contesti di persistenza, che risparmia dal dovere specificare esplicitamente `create=true` quando si usa l'annotazione `@In`.

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:persistence="http://jboss.com/products/seam/persistence"

 <persistence:managed-persistence-context name="productDatabase"
                        auto-create="true"
              persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components
>
```

```
<components>

   <component name="productDatabase"
      auto-create="true"
          class="org.jboss.seam.persistence.ManagedPersistenceContext">
      <property name="persistenceUnitJndiName"
>java:/productEntityManagerFactory</property>
   </component>

</components
>
```

La dichiarazione `<factory>` consente di specificare un valore o un'espressione di method binding che verrà valutata per inizializzare il valore di una variabile di contesto quando viene referenziata la prima volta.

```
<components>

                    <factory      name="contact"      method="#{contactManager.loadContact}"
 scope="CONVERSATION"/>

</components
>
```

Si può creare un "alias" (un secondo nome) per un componente Seam in questo modo:

```
<components>

   <factory name="user" value="#{actor}" scope="STATELESS"/>

</components
>
```

Si può anche creare un "alias" per un'espressione comunemente usata:

```
<components>

   <factory name="contact" value="#{contactManager.contact}" scope="STATELESS"/>

</components
>
```

E' comune vedere usato `auto-create="true"` con la dichiarazione `<factory>`:

```
<components>

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

</components
>
```

A volte si vuole riutilizzare lo stesso file `components.xml` con piccoli cambiamenti durante il deploy ed il testing. Seam consente di mettere dei wildcard della forma `@wildcard@` nel file `components.xml` che può essere rimpiazzato o dallo script Ant (a deployment time) o fornendo un file chiamato `components.properties` nel classpath (a development time). Si vedrà usato quest'ultimo approccio negli esempi di Seam.

5.3. File di configurazione a grana fine

Qualora si abbia un grande numero di componenti che devono essere configurati in XML, è più sensato suddividere l'informazione di `components.xml` in numerosi piccoli file. Seam consente di mettere la configurazione per una classe non anonima, per esempio, `com.helloworld.Hello` in una risorsa chiamata `com/helloworld/Hello.component.xml`. (Potresti essere familiare a

questo pattern, poiché è lo stesso usato da Hibernate.) L'elemento radice del file può essere o un elemento `<components>` oppure `<component>`.

La prima opzione lascia definire nel file componenti multipli:

```
<components>
   <component class="com.helloworld.Hello" name="hello">
      <property name="name"
>#{user.name}</property>
   </component>
   <factory name="message" value="#{hello.message}"/>
</components
>
```

La seconda opzione lascia definire o configurare un solo componente, ma è meno rumorosa:

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

Nella seconda opzione, il nome della classe è implicato nel file in cui appare la definizione del componente.

In alternativa, si può mettere una configurazione per tutte le classi nel pacchetto `com.helloworld` in `com/helloworld/components.xml`.

5.4. Tipi di proprietà configurabili

Le proprietà dei tipi stringa, primitivi o wrapper primitivi possono essere configurati solo come atteso:

```
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
>
```

Anche array, set e liste di stringhe o primitivi sono supportati:

```
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
>
```

Anche le mappe con chiavi associate a stringhe oppure valori stringa o primitivi sono supportati:

```
<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
>
```

Quando si configurano le proprietà multivalore, Seam preserverà di default l'ordine in cui vengono messi gli attributi in `components.xml` (amenoché venga usato `SortedSet`/`SortedMap` allora Seam userà `TreeMap`/`TreeSet`). Se la proprietà ha un tipo concreto (per esempio `LinkedList`) Seam userà quel tipo.

Si può anche fare l'override del tipo specificando un nome di classe pienamente qualificato:

```
<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
>
```

Infine si può unire assieme i componenti usando un'espressione value-binding. Si noti che è diverso dall'usare l'iniezione con `@In`, poiché avviene al momento dell'istanziamento del componente invece che al momento dell'invocazione. E' quindi molto più simile alle strutture con dependency injection offerte dai tradizionali IoC container come JSF o 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 risolve anche un'espressione stringa EL prima di assegnare il valore iniziale alla proprietà del bean del componente. Quindi si possono iniettare alcuni dati di contesto nei componenti.

```
<component name="greeter" class="com.example.action.Greeter">
   <property name="message"
>Nice to see you, #{identity.username}!</property>
</component
>
```

C'è un'importante eccezione. Se un tipo di proprietà a cui il valore iniziale assegnato è o una `ValueExpression` di Seam o una `MethodExpression`, allora la valutazione di EL è rimandata. Invece il wrapper dell'espressione appropriata viene creato e assegnato alla proprietà. I modelli di messaggi nel componente Home dell'Applicazione Framework di Seam servono da esempio.

```
<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."/>
```

Dentro il componente sipuò accedere all'espressione di stringa chiamando `getExpressionString()` sulla `ValueExpression` o `MethodExpression`. Se la proprietà è una `ValueExpression`, si può risolvere il valore usando `getValue()` e se la proprietà è un `MethodExpression`, si può invocare il metodo usando `invoke(Object args...)`. Ovviamente per assegnare un valore alla proprietà `MethodExpression`, l'intero valore iniziale deve essere una singola espressione EL.

5.5. Uso dei namespace XML

Attraverso gli esempi ci sono stati due modi per dichiarare i componenti: con e senza l'uso di namespace XML. Il seguente mostra un tipico file `components.xml` senza namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xsi:schemaLocation="http://jboss.com/products/seam/components http://jboss.com/
products/seam/components-2.1.xsd">

   <component class="org.jboss.seam.core.init">
      <property name="debug"
>true</property>
      <property name="jndiPattern"
>@jndiPattern@</property>
   </component>

</components
>
```

Come si può vedere, è abbastanza prolisso. Ancor peggio, i nomi del componente e dell'attributo non possono essere validati a development time.

La versione con namespace appare come:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
         xmlns:core="http://jboss.com/products/seam/core"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
            "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.1.xsd
                  http://jboss.com/products/seam/components http://jboss.com/products/seam/
components-2.1.xsd">

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

</components
>
```

Anche se le dichiarazioni di schema sono lunghe, il contenuto vero di XML è piatto e facile da capire. Gli schemi forniscono informazioni dettagliate su ogni componente e sugli attributi disponibili, consentendo agli editor XML di offrire un autocompletamento intelligente. L'uso di elementi con namespace semplifica molto la generazione ed il mantenimento in uno stato corretto dei file `components.xml`.

Questo funziona bene per i componenti predefiniti di Seam, ma per i componenti creati dall'utente? Ci sono due opzioni. La prima, Seam supporta un misto dei due modelli, consentendo l'uso

delle dichiarazioni generiche `<component>` for i componenti utente, assieme alle dichiarazioni con namespace dei componenti predefiniti. Ma ancor meglio, Seam consente di dichiarare in modo veloce i namespace per i propri componenti.

Qualsiasi pacchetto Java può essere associato ad un namespace XML annotando il pacchetto con l'annotazione `@Namespace`. (Le annotazioni a livello pacchetto vengono dichiarate in un file chiamato `package-info.java` nella directory del pacchetto.) Ecco un esempio tratto dalla demo seampay:

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

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

Questo è tutto ciò che bisogna fare per utilizzare lo stile namespace in `components.xml`! Adesso si può scrivere:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:pay="http://jboss.com/products/seam/examples/seampay"
        ... >

   <pay:payment-home new-instance="#{newPayment}"
                created-message="Created a new payment to #{newPayment.payee}" />

   <pay:payment name="newPayment"
            payee="Somebody"
            account="#{selectedAccount}"
            payment-date="#{currentDatetime}"
            created-date="#{currentDatetime}" />
    ...
</components
>
```

Oppure:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:pay="http://jboss.com/products/seam/examples/seampay"
        ... >

   <pay:payment-home>
      <pay:new-instance
```

```
>"#{newPayment}"</pay:new-instance>
      <pay:created-message
>Created a new payment to #{newPayment.payee}</pay:created-message>
   </pay:payment-home>

   <pay:payment name="newPayment">
      <pay:payee
>Somebody"</pay:payee>
      <pay:account
>#{selectedAccount}</pay:account>
      <pay:payment-date
>#{currentDatetime}</pay:payment-date>
      <pay:created-date
>#{currentDatetime}</pay:created-date>
    </pay:payment>
    ...
</components
>
```

Questi esempi illustrano i due usi modi d'uso di un elemento con namespace. Nella prima dichiarazione, `<pay:payment-home>` referenzia il componente `paymentHome`:

```
package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController
   extends EntityHome<Payment>
{
   ...
}
```

Il nome dell'elemento è una forma con trattino d'unione del nome del componente. Gli attributi dell'elemento sono la forma con trattino dei nomi delle proprietà.

Nella seconda dichiarazione, l'elemento `<pay:payment>` fa riferimento alla classe `Payment` nel pacchetto `org.jboss.seam.example.seampay`. In questo caso `Payment` è un entity dichiarato come componente Seam:

```
package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController
```

```
   extends EntityHome<Payment>
{
   ...
}
```

Se si vuole far funzionare la validazione e l'autocompletamento per i componenti definiti dall'utente, occorre uno schema. Seam non fornisce ancora un meccanismo per generare automaticamente uno schema per un set di componenti, così è necessario generarne uno manualmente. Come guida d'esempio si possono usare le definizioni di schema dei pacchetti standard di Seam.

Seam utilizza i seguenti namespace:

- components — `http://jboss.com/products/seam/components`
- core — `http://jboss.com/products/seam/core`
- drools — `http://jboss.com/products/seam/drools`
- framework — `http://jboss.com/products/seam/framework`
- jms — `http://jboss.com/products/seam/jms`
- remoting — `http://jboss.com/products/seam/remoting`
- theme — `http://jboss.com/products/seam/theme`
- security — `http://jboss.com/products/seam/security`
- mail — `http://jboss.com/products/seam/mail`
- web — `http://jboss.com/products/seam/web`
- pdf — `http://jboss.com/products/seam/pdf`
- spring — `http://jboss.com/products/seam/spring`

Eventi, interceptor e gestione delle eccezioni

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.

6.1. Eventi di Seam

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:

- Eventi JSF
- Eventi di transizione jBPM
- Azioni di pagina Seam
- Eventi guidati da componenti Seam
- Eventi contestuali Seam

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.jsp">
   <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.

6.2. Azioni di pagina

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.jsp" action="#{helloWorld.sayHello}"/>
</pages
>
```

Or we can use a `*` wildcard as a suffix to the `view-id` to specify an action that applies to all view ids that match 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.jsp">
      <action execute="#{helloWorld.sayHello}" if="#{not validation.failed}"/>
      <action execute="#{hitCount.increment}"/>
   </page>
</pages
>
```

Page actions are executed on both an initial (non-faces) request and a postback (faces) request. If you 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.

6.3. Parametri di pagina

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.

6.3.1. Mapping request parameters to the model

Seam lets us provide a value binding that maps a named request parameter to an attribute of a model object.

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

The `<param>` declaration is bidirectional, 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.jsp` (or from `/hello.jsp` back to `/hello.jsp`), 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).

6.4. Parametri di richiesta che si propagano

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.jsp" 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.

6.5. 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 *Sezione 30.1.4.3, «Riscrittura dell'URL»*.

6.6. Conversione e validazione

Si può specificare un convertitore JSF per proprietà di modelli complessi:

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
                  <param    name="op"    converterId="com.my.calculator.OperatorConverter"
 value="#{calculator.op}"/>
  </page>
</pages
>
```

In alternativa:

```
<pages>
```

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

possono essere usati anche i validatori JSF e `required="true"`:

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

In alternativa:

```
<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`.

6.7. Navigazione

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:

- Non è possibile specificare i parametri di richiesta da usare in caso di redirection.
- Non è possibile iniziare o terminare le conversazione da una regola.
- 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`.

Questa regola di navigazione JSF:

```
<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
>
```

Può essere riscritto come segue:

```
<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
>
```

Od anche:

```
<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
>
```

6.8. 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.jsp` 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
>
```

6.9. Eventi guidati da componenti

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*.

Si specificano gli event listener (observer) in `components.xml`.

```
<components>
   <event type="hello">
      <action execute="#{helloListener.sayHelloBack}"/>
      <action execute="#{logger.logHello}"/>
   </event>
</components
>
```

Dove il *tipo di evento* è solo una stringa arbitraria.

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");
   }
}
```

Oppure si può usare un'annotazione.

```
@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);
   }
}
```

6.10. Eventi contestuali

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` — chiamato quando fallisce la validazione JSF
- `org.jboss.seam.noConversation` — chiamato quando non c'è alcuna conversazione long-running mentre questa è richiesta
- `org.jboss.seam.preSetVariable.<name>` — chiamato quando la variabile di contesto <name> è impostata
- `org.jboss.seam.postSetVariable.<name>` — chiamato quando la variabile di contesto <name> è impostata
- `org.jboss.seam.preRemoveVariable.<name>` — chiamato quando la variabile di contesto <name> non è impostata
- `org.jboss.seam.postRemoveVariable.<name>` — chiamato quando la variabile di contesto <name> non è impostata
- `org.jboss.seam.preDestroyContext.<SCOPE>` — chiamato prima che il contesto <SCOPE> venga distrutto
- `org.jboss.seam.postDestroyContext.<SCOPE>` — chiamato prima che il contesto <SCOPE> venga distrutto
- `org.jboss.seam.beginConversation` — chiamato quando inizia una conversazione long-running
- `org.jboss.seam.endConversation` — chiamato quando finisce una conversazione long-running
- `org.jboss.seam.conversationTimeout` — chiamato quando avviene un timeout di conversazione. L'id di conversazione viene passato come parametro.
- `org.jboss.seam.beginPageflow` — chiamato quando inizia un pageflow
- `org.jboss.seam.beginPageflow.<name>` — chiamato quando inizia il pageflow <name>
- `org.jboss.seam.endPageflow` — chiamato quando finisce il pageflow
- `org.jboss.seam.endPageflow.<name>` — chiamato quando finisce il pageflow <name>
- `org.jboss.seam.createProcess.<name>` — chiamato quando viene creato il processo <name>
- `org.jboss.seam.endProcess.<name>` — chiamato quando finisce il processo <name>
- `org.jboss.seam.initProcess.<name>` — chiamato quando il processo <name> è associato alla conversazione

- `org.jboss.seam.initTask.<name>` — chiamato quando il task <name> è associato alla conversazione
- `org.jboss.seam.startTask.<name>` — chiamato quando il task <name> viene avviato
- `org.jboss.seam.endTask.<name>` — chiamato quando il task <name> viene terminato
- `org.jboss.seam.postCreate.<name>` — chiamato quando il componente <name> viene creato
- `org.jboss.seam.preDestroy.<name>` — chiamato quando il componente <name> viene distrutto
- `org.jboss.seam.beforePhase` — chiamato prima dell'inizio di una fase JSF
- `org.jboss.seam.afterPhase` — chiamato dopo la fine di una fase JSF
- `org.jboss.seam.postInitialization` — chiamato quando Seam ha inizializzato e avviato tutti i componenti
- `org.jboss.seam.postReInitialization` — chiamato quando Seam ha reinizializzato ed avviato tutti i componenti dopo un redeploy
- `org.jboss.seam.exceptionHandled.<type>` — chiamato quando viene gestita da Seam un'eccezione non catturata del tipo <type>
- `org.jboss.seam.exceptionHandled` — chiamato quando viene gestita da Seam un'eccezione non catturata
- `org.jboss.seam.exceptionNotHandled` — chiamato quando non c'è alcun handler per eccezioni non catturate
- `org.jboss.seam.afterTransactionSuccess` — chiamato quando ha successo una transazione nel Seam Application Framework
- `org.jboss.seam.afterTransactionSuccess.<name>` — chiamato quando ha successo una transazione nel Seam Application Framework che gestisce un'entità chiamata `<name>`
- `org.jboss.seam.security.loggedOut` — chiamato quando un utente si disconnette
- `org.jboss.seam.security.loginFailed` — chiamato quando fallisce un tentativo di autenticazione utente
- `org.jboss.seam.security.loginSuccessful` — chiamato quando un utente si autentica con successo
- `org.jboss.seam.security.notAuthorized` — chiamato quando fallisce un controllo di autorizzazione
- `org.jboss.seam.security.notLoggedIn` — chiamato quando non è autenticato alcun utente mentre l'autenticazione è richiesta

- `org.jboss.seam.security.postAuthenticate.` — chiamato dopo che un utente viene autenticato
- `org.jboss.seam.security.preAuthenticate` — chiamato prima del tentativo di autenticazione di un utente

I componenti Seam possono osservare uno di questi eventi così come osservano qualsiasi altro evento guidato da componente.

6.11. Interceptor Seam

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).

6.12. Gestione delle eccezioni

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.

6.12.1. Eccezioni e transazioni

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.

6.12.2. Abilitare la gestione delle eccezioni di Seam

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
>
```

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

6.12.3. Uso delle annotazioni per la gestione delle eccezioni

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 { ... }
```

Nota

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 { ... }
```

6.12.4. Uso di XML per la gestione delle eccezioni

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`.

6.12.4.1. 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`.

6.12.5. Alcune eccezioni comuni

Se si usa 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
>
```

Se si usa il Seam Application Framework:

```
<exception class="org.jboss.seam.framework.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message
>Not found</message>
  </redirect>
</exception
>
```

Se si usa 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
>
```

E per 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. `no-conversation-view-id` and `conversation-required` give you finer grained control over session expiration if you are inside a conversation.

6.13. conversation-required

When specified as the attribute of a `page` element in `pages.xml`, this setting controls whether a page requires an active long-running or nested conversation before being rendered. If there is not an active long-running or nested conversation when trying to access the page, you will be redirected to the `no-conversation-view-id` view (which is specified in the root `pages` element) instead.

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

Conversazioni e gestione del workspace

E' ora di capire il modello di conversazione di Seam con maggior dettaglio.

Storicamente la nozione di "conversazione" Seam si presenta come unificatrice di tre differenti idee:

- L'idea di uno *spazio di lavoro (workspace)*, che ho incontrato in un progetto per il governo Vittoriano nel 2002. In questo progetto fui obbligato ad implementare la gestione del workspace sopra Struts, un'esperienza che prego di non ripetere mai più.
- L'idea di una *transazione per l'applicazione* con semantica ottimista, e il convincimento che i framework esistenti basati su un'architettura stateless non potessero fornire una gestione efficiente dei contesti di persistenza estesa. (La squadra di Hibernate è veramente stanca di sentire lamentele per le `LazyInitializationException`, che non è in verità colpa di Hibernate, ma piuttosto colpa di un modello di contesto di persistenza estremamente limitato delle architetture stateless come il framework Spring o il tradizionale (anti)pattern *stateless session facade* in J2EE.)
- L'idea di un *task* a workflow.

Unificando queste idee e fornendo un supporto profondo nel framework, si ha un costrutto potente che consente di creare applicazioni più ricche e più efficienti con meno codice di prima.

7.1. Il modello di conversazioni di Seam

Gli esempi visti finora usano un modello di conversazione molto semplice che segue queste regole:

- C'è sempre un contesto di conversazione attivo durante le fasi di apply request values, process validations, update model values, invoke application e render response del ciclo di vita della richiesta JSF.
- Alla fine della fase restore view del ciclo di vita della richiesta JSF, Seam tenta di ripristinare ogni precedente contesto di conversazione long-running. Se non ne esiste nessuno, Seam crea un nuovo contesto di conversazione temporanea.
- Quando viene incontrato un metodo `@Begin`, il contesto della conversazione temporanea è promosso a conversazione long-running.
- Quando viene incontrato un metodo `@End`, il contesto della conversazione long-running è ridotto a conversazione temporanea.
- Alla fine della fase di render response del ciclo di vita della richiesta JSF, Seam memorizza i contenuti del contesto di conversazione long-running o distrugge i contenuti del contesto di conversazione temporanea.

- Qualsiasi richiesta faces (un postback JSF) si propagherà nel contesto della conversazione. Di default le richieste non-faces (richieste GET, per esempio) non si propagano nel contesto di conversazione, si veda sotto per maggiori informazioni.
- Se il ciclo di vita della richiesta JSF viene accorciato da un redirect, Seam memorizza e ripristina in modo trasparente il contesto dell'attuale conversazione — amenoché la conversazione sia già stata terminata tramite `@End(beforeRedirect=true)`.

Seam propaga in modo trasparente il contesto della conversazione (includendo il contesto della conversazione temporanea) lungo i postback JSF e i redirect. Se non si fa niente di speciale, una *richiesta non-faces* (per esempio una richiesta GET) non verrà propagata nel contesto di conversazione e non verrà processata in una conversazione temporanea. Questo è solitamente - ma non sempre - il comportamento desiderato.

Se si vuole propagare una conversazione Seam lungo una richiesta non-faces, non occorre esplicitamente codificare l'*id della conversazione* come parametro di richiesta:

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

O in stile più JSF:

```
<h:outputLink value="main.jsf">
   <f:param name="#{manager.conversationIdParameter}" value="#{conversation.id}"/>
   <h:outputText value="Continue"/>
</h:outputLink
>
```

Se si utilizza la libreria di tag Seam, questo è l'equivalente:

```
<h:outputLink value="main.jsf">
   <s:conversationId/>
   <h:outputText value="Continue"/>
</h:outputLink
>
```

Se si desidera disabilitare la propagazione del contesto di conversazione per il postback, viene usato un simile trucchetto:

```
<h:commandLink action="main" value="Exit">
```

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

Se si utilizza la libreria di tag Seam, questo è l'equivalente:

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

Si noti che disabilitando la propagazione del contesto della conversazione non è assolutamente la stessa cosa che terminare la conversazione:

Il parametro di richiesta `conversationPropagation`, o il tag `<s:conversationPropagation>` possono anche essere usati per iniziare e terminare la conversazione, o iniziare una conversazione innestata.

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

```
<h:commandLink action="main" value="Select Child">
   <s:conversationPropagation type="nested"/>
</h:commandLink
>
```

```
<h:commandLink action="main" value="Select Hotel">
   <s:conversationPropagation type="begin"/>
</h:commandLink
>
```

```
<h:commandLink action="main" value="Select Hotel">
   <s:conversationPropagation type="join"/>
</h:commandLink
>
```

Il modello di conversazione rende semplice costruire applicazioni che si comportano in modo corretto in presenza di operazioni con finestre multiple. Per molte applicazioni, questo è quello che serve. Alcune applicazioni complesse hanno uno ed entrambi dei seguenti requisiti aggiuntivi:

- Una conversazione propaga diverse unità più piccole di interazione utente, che vengono eseguite in modo seriale o anche concorrente. Le più piccole *conversazioni innestate* hanno il proprio stato isolato di conversazione, ed hanno anche accesso allo stato della conversazione più esterna.
- L'utente è in grado di passare tra più conversazioni dentro la stessa finestra del browser. Questa caratteristica è chiamata *gestione del workspace*.

7.2. Conversazioni innestate

Una conversazione innestata viene creata invocando un metodo marcato con `@Begin(nested=true)` dentro lo scope di una conversazione esitente. Una conversazione innestata ha un proprio contesto di conversazione, ma può leggere i valori dal contesto della conversazione più esterna. Il contesto della conversazione più esterna è in sola lettura in una conversazione innestata, ma poiché gli oggetti sono ottenuti per riferimento, i cambiamenti agli oggetti stessi si rifletteranno nel contesto più esterno.

- Innestando una conversazione si inizializza un contesto che viene messo sullo stack del contesto della conversazione originale, o più esterna. La conversazione più esterna è considerata padre.
- Ogni valore messo in outjection o direttamente impostato nel contesto della conversazione innestata non influenza gli oggetti accessibili nel contesto della conversazione padre.
- L'iniezione o la ricerca nel contesto da un contesto di conversazione per prima cosa cercherà il valore nell'attuale contesto e, se non viene trovato alcun valore, procederà lungo lo stack della conversazione se questa è innestata. Come si vedrà, questo comportamento può essere ridefinito.

Quando si incontra una `@End`, la conversazione innestata verrà distrutta, togliendola dallo stack (pop), e la conversazione più esterna verrà ripristinata. Le conversazioni possono essere annidate con gradi di profondità arbitrari.

Certe attività utente (gestione del workspace, o il pulsante indietro) possono causare il ripristino della conversazione più esterna prima che venga terminata la conversazione innestata. In questo caso è possibile avere più conversazioni innestate concorrenti che appartengono alla stessa conversazione più esterna. Se la conversazione più esterna finisce prima che termini la conversazione innestata, Seam distrugge tutti i contesti delle conversazioni innestate assieme a quella più esterna.

La conversazione alla fine dello stack delle conversazioni è la conversazione radice. Distruggendo questa conversazione si distruggono sempre tutti i suoi discendenti. Si può ottenere questo in modo dichiarativo specificando `@End(root=true)`.

Una conversazione può essere pensata come uno *stato continuo*. Le conversazioni innestate consentono all'applicazione di catturare lo stato continuo consistente in vari punti durante l'interazione utente, quindi assicurando un comportamento corretto rispetto al pulsante indietro ed alla gestione del workspace.

Come menzionato in precedenza, se un componente si trova in una conversazione padre dell'attuale conversazione innestata, la conversazione innestata userà la stessa istanza. Occasionalmente, è utile avere diverse istanze in ciascuna conversazione innestata, cosicché l'istanza del componente che si trova nella conversazione padre sia invisibile alle sue conversazioni figlie. Si può ottenere questo comportamento annotando il componente `@PerNestedConversation`.

7.3. Avvio di conversazioni con richieste GET

JSF non definisce alcun tipo di action listener da lanciare quando una pagina viene acceduta tramite una richiesta non-faces (per esempio, una richiesta HTTP GET). Questo può succedere se l'utente memorizza la pagina come segnalibro, o se si naviga nella pagina tramite un `<h:outputLink>`.

A volte si vuole immediatamente iniziare una conversazione all'accesso della pagina. Poiché non c'è alcun metodo d'azione JSF, non si può risolvere il problema nel consueto modo, annotando l'azione con `@Begin`.

Sorge un altro problema se la pagina ha bisogno di recuperare uno stato da una variabile di contesto. Si sono già visti due modi per risolvere questo problema. Se lo stato è mantenuto in un componente Seam, si può recuperare lo stato in un metodo `@Create`. Se non lo è, si può definire un metodo `@Factory` per la variabile di contesto.

Se nessuna di queste opzioni funziona, Seam permette di definire una *pagina d'azione* nel file `pages.xml`.

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

Il metodo d'azione viene chiamato all'inizio della fase di render response, ogni volta che la pagina sta per essere generata. Se l'azione della pagina ritorna un esito non-null, Seam processerà ogni opportuna regola di JSF e Seam, e genererà un'altra pagina.

Se *tutto* ciò che si vuole fare prima di generare una pagina è iniziare una conversazione, si può utilizzare un metodo d'azione predefinito che fa questo:

```
<pages>
```

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

Si noti che si può chiamare quest'azione ridefinita da un controllo JSF, ed in modo simile si può usare `#{conversation.end}` per terminare le conversazioni.

Se si vuole più controllo, per unirsi a conversazioni esistenti od iniziare una conversazione innestata, per iniziare un pageflow od una conversazione atomica, occorre usare l'elemento `<begin-conversation>`.

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

C'è anche un elemento `<end-conversation>`.

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

Per risolvere il primo problema, si hanno cinque opzioni:

- Annotare il metodo `@Create` con `@Begin`
- Annotare il metodo `@Factory` con `@Begin`
- Annotare il metodo d'azione della pagina Seam con `@Begin`
- Usare `<begin-conversation>` in `pages.xml`.
- Usare `#{conversation.begin}` come metodo d'azione della pagina Seam

7.4. Usando <s:link> e <s:button>

I comandi link JSF eseguono sempre un invio di form tramite JavaScript, che rompe le caratteristiche dei browser "Apri in nuova finestra" o "Apri in nuova scheda". Nel semplice JSF occorre usare un <h:outputLink> se si vuole questa finzionalità. Ma ci sono due grandi limitazioni in <h:outputLink>

- JSF non fornisce un modo per agganciare un action listener a <h:outputLink>.
- JSF non propaga la riga selezionata di un DataModel poiché non c'è alcun invio di form.

Seam fornisce la nozione di *pagina d'azione* per aiutare a risolvere il primo problema, ma questo non aiuta per niente il secondo problema. Si *può* aggirare questo usando l'approccio RESTful di passare un parametro di richiesta e di riottenere l'oggetto selezionato lato server. In alcuni casi — come nell'esempio Seam del Blog — questo è il migliore approccio. Lo stile RESTful supporta i segnalibri, poiché non richiede uno stato lato server. In altri casi, dove non interessanno i segnalibri, l'uso di un @DataModel e di @DataModelSelection è conveniente e trasparente!

Per riempiere questa mancanza di funzionalità e rendere semplicela propagazione delle conversazioni da gestire, Seam fornisce il tag JSF <s:link>.

Il link può specificare solo l'id della vista JSF:

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

Oppure può spcificareil metodo d'azione (nel qual caso l'esito dell'azione determina la pagina di destinazione):

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

Se si specificano *entrambi* l'id della vista JSF ed il metodo d'azione, verrà usata la 'vista' *amenoché* il metodo d'azione ritorni un esito non-null:

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

Il link propaga automaticamente la riga selezionata del DataModel usando all'interno <h:dataTable>:

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}" value="#{hotel.name}"/>
```

Si può lasciare lo scope di una conversazione esistente:

```
<s:link view="/main.xhtml" propagation="none"/>
```

Si può iniziare, terminare, o innestare le conversazioni:

```
<s:link action="#{issueEditor.viewComment}" propagation="nest"/>
```

Se il link inizia una conversazione, si può anche specificare il pageflow da usare:

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

L'attributo `taskInstance` è per l'uso nelle liste di task jBPM:

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

(Si veda l'applicazione demo di Negozio DVD come esempio.)

Infine se il "link" deve essere visualizzato come pulsante, si usi `<s:button>`:

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

7.5. Messaggi di successo

E' abbastanza comune visualizzare all'utente un messaggio indicante il successo od il fallimento di un'azione. E' conveniente usare `FacesMessage` di JSF per questo scopo. Sfortunatamente un'azione di successo spesso richiede un redirect del browser, e JSF non propaga i messaggi faces lungo i redirect. Questo rende difficile visualizzare i messaggi nel semplice JSF.

Il componente Seam predefinito con scope conversazione chiamato `facesMessages` risolve questo problema. (Occorre avere installato il filtro redirect di Seam.)

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

Qualsiasi messaggio aggiunto a `facesMessages` viene usato nella fase render response più prossima per la conversazione corrente. Questo funziona anche quando non ci sono conversazioni long-running poiché Seam preserva anche i contesti di conversazioni temporanee lungo i redirect.

Si possono anche includere espressioni JSF EL in un sommario di messaggi faces:

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

Si possono visualizzare i messaggi nella solita maniera, per esempio:

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

7.6. Id di una conversazione naturale

Lavorando con le conversazioni che trattano oggetti persistenti, può essere desiderabile utilizzare la chiave naturale di business dell'oggetto invece dello standard, id di conversazione "surrogato":

Redirect facile verso conversazioni esistenti

Può essere utile redirigersi verso una conversazione esistente se l'utente richiede la stessa operazione due volte. Si prenda quest'esempio: «Sei su eBay, stai per pagare un oggetto che hai scelto come regalo per i tuoi genitori. Diciamo che lo stai per inviare a loro - stai per inserire i dettagli di pagamento ma non ti ricordi il loro indirizzo. Accidentalmente riutilizzi la stessa finestra del browser per cercare il loro indirizzo. Ora devi ritornare al pagamento di quell'oggetto.»

Con una conversazione naturale è molto facile riunire l'utente alla conversazione esistente, e riprendere dove aveva lasciato - riunirsi alla conversazione pagaOggetto con l'idOggetto come id di conversazione.

URL user friendly

Questo si concretizza in una gerarchia navigabile (si può navigare editando l'url) e in URL significativi (come mostrato in Wiki - quindi non si identifichino gli oggetti con id casuali). Per alcune applicazioni gli URL user friendly sono sicuramente meno importanti.

Con una conversazione naturale, quando si costruisce un sistema di prenotazione hotel (od una qualsiasi applicazione) si può generare un URL del tipo `http://seam-hotels/`

`book.seam?hotel=BestWesternAntwerpen` (sicuramente un qualsiasi parametro `hotel`, che mappi sul modello di dominio, deve essere univoco) e con URLRewrite si può facilmente trasformare questo in http://seam-hotels/book/BestWesternAntwerpen.

Molto meglio!

7.7. Creazione di una conversazione naturale

Le conversazioni naturali sono definite in `pages.xml`:

```
<conversation name="PlaceBid"
              parameter-name="auctionId"
              parameter-value="#{auction.auctionId}"/>
```

La prima cosa da notare dalla definizione di cui sopra è che la conversazione ha un nome, in questo caso `PlaceBid`. Questo nome identifica univocamente questa particolare conversazione e viene usato dalla definizione di `pagina` per identificare una conversazione con nome in cui partecipare.

Il prossimo attributo `parameter-name` definisce il parametro di richiesta che conterrà l'id della conversazione naturale, al posto del parametro dell'id della conversazione di default. In quest'esempio, il `parameter-name` è `auctionId`. Questo significa che invece di un parametro di conversazione come `cid=123` che appare nell'URL della pagina, conterrà invece `auctionId=765432`.

L'ultimo attributo della configurazione di cui sopra, `parameter-value`, definisce un'espressione EL usata per valutare il valore della chiave naturale di business da usare come id di conversazione. In quest'esempio, l'id della conversazione sarà il valore della chiave primaria dell'istanza `auction` attualmente nello scope.

Poi si definirà quali pagine parteciperanno nella conversazione con nome. Questo è fatto specificando l'attributo `conversation` per una definizione di `pagina`:

```
 <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
```

```
>
```

7.8. Redirezione alla conversazione naturale

Avviando o facendo redirect verso una conversazione naturale ci sono un numero di opzioni possibili per specificare il nome della conversazione naturale. Si guardi alla seguente definizione di pagina:

```
 <page view-id="/auction.xhtml">
  <param name="id" value="#{auctionDetail.selectedAuctionId}"/>

  <navigation from-action="#{bidAction.placeBid}">
   <redirect view-id="/bid.xhtml"/>
  </navigation>
 </page
>
```

Da qua si può vedere che invocando l'azione `#{bidAction.placeBid}` dalla vista auction (comunque, tutti questi esempio sono presi dall'esempio seamBay), che verrà rediretta a `/bid.xhtml`, che come si è visto, è configurata con la conversazione naturale `PlaceBid`. La dichiarazione del metodo d'azione appare come:

```
  @Begin(join = true)
  public void placeBid()
```

Quando le conversazioni con nome vengono specificate nell'elemento `<page/>`, la redirezione alla conversazione con nome avviene come parte delle regole, dopo che il metodo d'azione è già stato invocato. Questo è un problema quando ci si redirige ad una conversazione esistente, poiché la redirezione deve avvenire prima che sia invocato il metodo d'azione. Quindi è necessario specificare il nome della conversazione quando si invoca l'azione. Un metodo per fare questo è usare il tag `s:conversationName`:

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

Un'altra alternativa è specificare l'attributo `conversationName` quando si usano o `s:link` o `s:button`:

```
<s:link value="Place Bid" action="#{bidAction.placeBid}" conversationName="PlaceBid"/>
```

7.9. Gestione del workspace

La gestione del workspace è la capacità di "cambiare" le conversazioni all'interno di una singola finestra. Seam rende la gestione del workspace completamente trasparente a livello di codice Java. Per abilitare la gestione del workspace, occorre fare questo:

- Fornire un testo di *descrizione* ad ogni id di vista (quando si usa JSF o le regole di navigazione JSF) od il nodo della pagina (quando si usano i pageflow jPDL). Questo testo di descrizione viene mostrato all'utente attraverso lo switcher di workspace.
- Includere uno o più switcher JSP standard di workspace oppure frammenti facelets nelle proprie pagine. I frammenti standard supportano la gestione del workspace attraverso un menu dropdown, una lista di conversazioni, oppure breadcrumb.

7.9.1. Gestione del workspace e navigazione JSF

Quando si usano JSF o le regole di navigazione di Seam, Seam cambia la conversazione ripristinando l'attuale `view-id` per quella conversazione. Il testo descrittivo per il workspace viene definito in un file chiamato `pages.xml` che Seam si aspetta di trovare nella directory `WEB-INF`, giusto dove si trova `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
>
```

Si noti che se questo file manca, l'applicazione Seam continuerà a funzionare perfettamente! La sola cosa che mancherà sarà la possibilità di cambiare workspace.

7.9.2. Gestione del workspace e pageflow jPDL

Quando si usa una definizione di pageflow jPDL, Seam cambia la conversazione ripristinando il corrente stato del processo jBPM. Questo è un modello più flessibile poiché consente allo stesso `view-id` di avere descrizioni differenti a seconda del nodo corrente `<page>`. Il testo di descrizione è definito dal nodo `<page>`:

```
<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
>
```

7.9.3. Lo switcher delle conversazioni

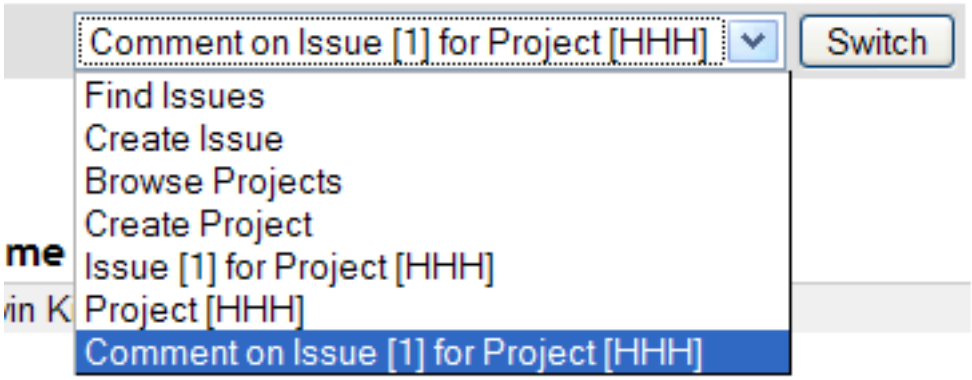
Includere il seguente frammento nella pagina JSP o facelets per ottenere un menu dropdown che consenta di cambiare la conversazione, o la pagina dell'applicazione:

```
<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 quest'esempio si ha un menu che include un item per ogni conversazione, assieme a due item aggiuntivi che consentono all'utente di iniziare una nuova conversazione.

Solo le conversazioni con una descrizione (specificata in `pages.xml`) verranno incluse nel menu dropdown.



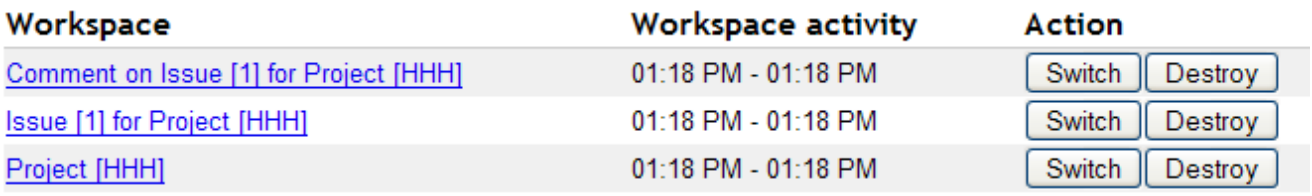
7.9.4. La lista delle conversazioni

La lista delle conversazioni è molto simile allo switcher delle conversazioni, tranne che viene mostrata come tabella:

```
<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
>
```

Si immagini di voler personalizzare questo per la propria applicazione.



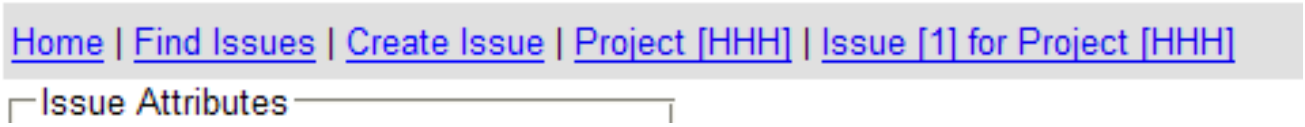
Solo le conversazioni con una descrizione verranno incluse nella lista.

Si noti che la lista delle conversazioni consente all'utente di distruggere i workspace.

7.9.5. Breadcrumbs

I breadcrumb sono utili nelle applicazioni che usano il modello di conversazioni innestato. I breadcrumb sono una lista di link alle conversazioni nell'attuale stack di conversazioni:

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



7.10. Componenti conversazionali ed associazione ai componenti JSF

I componenti conversazionali hanno una piccola limitazione: non possono essere usati per mantenere un riferimento ai componenti JSF. (In generali si preferisce non usare questa funzionalità di JSF amenoché sia assolutamente necessario, poiché questo crea una dipendenza stretta della logica dell'applicazione con la vista.) Sulle richieste postback, i binding dei componenti vengono aggiornati durante la fase restore view, prima che il contesto della conversazione di Seam venga ripristinato.

Per aggirare questo si usa un componente con scope evento per memorizzare i binding dei componenti ed per iniettarlo nel componente a scope conversazione che lo richiede.

```
@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;

   ...
}
```

Inoltre non si può iniettare un componente con scope conversazione in un componente con scope evento a cui associare il controllo JSF. Questo include i componenti predefiniti Seam come `facesMessages`.

In alternativa si può accedere all'albero del componente JSF attraverso l'handle implicito `uiComponent`. Il seguente esempio accede a `getRowIndex()` del componente `UIData` che è retrostante alla tabella dei dati durante l'interazione, e stampa il numero della riga corrente:

```
<h:dataTable id="lineItemTable" var="lineItem" value="#{orderHome.lineItems}">
  <h:column>
    Row: #{uiComponent['lineItemTable'].rowIndex}
  </h:column>
  ...
</h:dataTable
>
```

I componenti JSF UI sono disponibili in questa mappa con il loro identificatore di client.

7.11. Chiamare concorrenti ai componenti conversazionali

Una discussione generale sulle chiamate concorrenti ai componenti Seam può essere trovata in *Sezione 4.1.10, «Modello di concorrenza»*. Qua si discuterà la situazione più comune in cui si troverà la concorrenza — accedendo a componenti conversazionali da richieste AJAX. Si dicuteranno le opzioni che la libreria client Ajax dovrebbe fornire per controllare gli eventi originati nel client — e si vedranno le opzioni che fornisce RichFaces.

I componenti conversazionali non consentono una vero accesso concorrente, e quindi Seam accoda ogna richiesta per poi processarla in modo seriale. Questo consente ad ogni richiesta di venir eseguita in un modo deterministico. Comunque, una semplice cosa non è cosa ottima — in primo luogo, se un metodo per qualche ragione impiega molto tempo a terminare, eseguirlo più volte quando il client genera una richiesta, è una cattiva idea (potenziale per attacchi di tipo Denial of Service), ed in secondo luogo, AJAX spesso viene usato per fornire un veloce aggiornamento dello stato all'utente, e così continuare ad eseguire l'azione per lungo tempo non è utile.

Quindi quando si lavora all'interno di una conversazione long-running, Seam accoda l'evento azione per un periodo di tempo (il timeout della richiesta concorrente); se non si può processare l'evento in tempo, si crea una conversazione temporanea e si stampa un messaggio all'utente per rendergli noto cosa sta succedendo. E' quindi molto importante non inondare il server con eventi AJAX!

Si può impostare un valore di default sensibile per il timeout delle richieste concorrenti (in ms) dentro il file components.xml:

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

Si può anche perfezionare questo timeout a livello di ogni singola pagina:

```
<page view-id="/book.xhtml"
```

```
        conversation-required="true"
        login-required="true"
        concurrent-request-timeout="2000" />
```

Finora si è discusso delle richieste AJAX che appaiono in serie all'utente - il client dice al server quale evento avviene, e quindi rigenera parte della pagina a seconda del risultato. Questo approccio è ottimo quando la richiesta AJAX è leggera (i metodi chiamati sono semplici, per esempio il calcolo della somma di una colonna di numeri): Ma cosa succede se occorre un calcolo più complesso che dura diversi minuti?

Per una computazione pesante occorre usare un approccio basato sull'interrogazione — il client spedisce una richiesta AJAX al server, che causa un'azione asincrona sul server (la risposta al client è immediata) ed il client quindi interroga il server per gli aggiornamenti. Questo è un buon approccio quando si ha un'azione long-running per la quale è importante che ciascuna azione venga eseguita (non si vuole che qualcuna vada in timeout).

7.11.1. Come si può progettare la nostra applicazione AJAX conversazionale?

In primo luogo occorre decidere se si vuole usare una richiesta semplice "seriale" e se si vuole l'approccio con interrogazione.

Nel caso di richiesta "seriale" occorre stimare quando tempo occorrerà alle richieste per completarsi - è più breve del timeout della richiesta concorrente? Altrimenti si potrebbe volere probabilmente modificare il timeout per questa pagina (come discusso sopra). Si vuole che la coda lato client prevenga l'inondazione di richiesta al server. Se l'evento si verifica spesso (es. keypress, onblur di un campo d'input) e l'aggiornamento immediato del client non è una priorità si deve ritardare la richiesta lato client. Quando si lavora sul ritardo delle richieste, si tenga presente che l'evento può venire accodato anche lato server.

Infine la libreria del client può fornire un'opzione su come abbandonare le richieste duplicate non terminate a favore di quelle più recenti.

Usando un design di tipo interrogazione richiede un minor fine-tuning. Basta marcare il metodo d'azione con `@Asynchronous` e decidere l'intervallo di interrogazione:

```
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;
}
```

7.11.2. Gestione degli errori

Comunque per quanto in modo attento si progetti la propria applicazione in modo che accodi le richieste concorrenti nel componente conversazionale, c'è il rischio che il server venga sovraccaricato e sia incapace di processare tutte le richieste prima che la richiesta debba aspettare più a lungo del `concurrent-request-timeout`. In questo caso Seam lancerà una `ConcurrentRequestTimeoutException` che potrà venir gestita in `pages.xml`. Si raccomanda di inviare un errore HTTP 503:

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

503 Service Unavailable (HTTP/1.1 RFC)

Il server è incapace di gestire la richiesta a causa di un temporaneo sovraccarico o di una manutenzione al server. L'implicazione è che questa è una condizione temporanea che verrà risolta dopo un qualche ritardo.

In alternativa si può fare il redirect ad una pagina d'errore:

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException" log-level="trace">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message
>The server is too busy to process your request, please try again later</message>
  </redirect>
</exception
>
```

ICEfaces, RichFaces e Seam Remoting possono tutti gestire i codici d'errore HTTP. Seam Remoting mostrerà una finestra di dialogo con l'errore HTTP e ICEfaces indicherà l'errore nel suo componente di stato. RichFaces fornisce il supporto più completo per la gestione degli errori

HTTP e chiamate callback definibili dall'utente. Per esempio, per mostrare il messaggio d'errore all'utente:

```
<script type="text/javascript">
  A4J.AJAX.onError = function(req,status,message) {
    alert("An error occurred");
  };
</script
>
```

Invece di un codice d'errore, il server riporta che la vista è scaduta, forse per un timeout di sessione, si usi una funzione callback separata in RichFaces per gestire questo scenario.

```
<script type="text/javascript">
  A4J.AJAX.onExpired = function(loc,message) {
    alert("View expired");
  };
</script
>
```

In alternativa si può consentire a RichFaces di gestire quest'errore, nel qual caso all'utente verrà presentato un prompt che chiede "Lo stato della vista non può essere ripristinato - ricaricare la pagina?" Si può personalizzare questo messaggio impostando la seguente chiave in un resource bundle dell'applicazione.

```
AJAX_VIEW_EXPIRED=View expired. Please reload the page.
```

7.11.3. RichFaces (Ajax4jsf)

RichFaces (Ajax4jsf) è la libreria AJAX più usata in Seam e fornisce tutti i controlli discussi sopra:

- `eventsQueue` — fornisce una coda in cui vengono messi gli eventi. Tutti gli eventi vengono accodati e le richieste vengono inviate al server in modo seriale. Questo è utile se la richiesta al server può prendersi un pò di tempo per essere eseguita (es. computazione pesante, recupero di informazioni da una sorgente lenta) ed il server non viene inondato.
- `ignoreDupResponses` — ignora la risposta prodotta dalla richiesta se una recente richiesta simile è già in coda. ignoreDupResponses="true" *non cancella* l'elaborazione della richiesta lato server — previene solamente aggiornamenti non necessari lato client.

  Quest'opzione dovrebbe essere usata con cautela nelle conversazioni Seam poiché consente di eseguire richeste concorrenti multiple.

- `requestDelay` — definisce il tempo (in ms) in cui la richiesta rimane in coda. Se la richiesta non è stata processata dopo questo tempo, la richiesta verrà inviata (anche se la risposta è stata ricevuta) o scartata (se c'è in coda una richiesta recente simile).

  Quest'opzione dovrebbe essere usata con cautela nelle conversazioni Seam poiché consente richieste concorrenti multiple- Occorre accertarsi che il ritardo impostato (in combinazione con il timeout delle richieste concorrenti) sia più lungo dell'azione da eseguire.

- `<a:poll reRender="total" interval="1000" />` — interroga il server, e rigenera un'area come occorre

Pageflows e processi di business

JBoss jBPM è un motore di gestione dei processi di business per ambiente Java SE o EE. jBPM ti consente di rappresentare un processo di business o un'interazione utente come un grafo di nodi, raffiguranti stati d'attesa, decisioni, compiti (task), pagine web, ecc. Il grafo viene definito usando un dialetto XML semplice, molto leggibile, chiamato jPDL, che può essere editato e visualizzato graficamente usando un plugin di eclipse. jPDL è un linguaggio estendibile ed è adatto per un range di problemi, dalla definizione di un flusso di pagine dell'applicazione web alla gestione tradizionale del workflow, fino all'orchestrazione di servizi in un ambiente SOA.

Le applicazioni Seam utilizzano jBPM per due tipi di problemi:

- Complesse interazioni da parte dell'utente comportano la definizione un pageflow (flusso di pagina). Una definizione di un processo con jPDL stabilisce il flusso delle pagine per una singola conversazione. Una conversazione in Seam è considerata un'interazione di breve durata con un singolo utente.
- Definizione del processo di business sottostante. Il processo di business può comportare una serie di conversazioni con più utenti. Il suo stato viene persistito nel database jBPM, divenendo così di lunga durata. Il coordinamento delle attività di più utenti è un problema molto più complesso che descrivere l'interazione di un singolo utente, cosicché jBPM offre dei modi sofisticati per la gestione dei compiti (task) e per la gestione di più percorsi concorrenti di esecuzione.

Non confondere le due cose! Queste operano a livelli molto diversi e con diverso grado di granularità. *Pageflow*, *conversazione* e *task* si riferiscono tutti alla singola interazione con il singolo utente. Un processo di business comporta più task. Quindi le due applicazione di jBPM sono totalmente ortogonali. Possono essere usate assieme, in modo indipendente, o si può non usarle affatto.

Non serve conoscere jPDL per usare Seam. Se ci si trova bene nel definire un pageflow con JSF o con le regole di navigazione di Seam, e se l'applicazione è guida più dai dati che dal processo, probabilmente non serve usare jBPM. Ma noi pensiamo che strutturare l'interazione dell'utente in termini di rappresentazione grafical ben definita aiuti a costruire applicazioni più robuste.

8.1. Pageflow in Seam

Ci sono due modi per definire il pageflow in Seam:

- Uso di JSF o delle regole di navigazione di Seam - il *modello di navigazione stateless*
- Utilizzo di jPDL - il *modello di navigazione stateful*

Applicazioni molto semplici richiedono soltanto un modello di navigazione stateless. Invece applicazioni molto complesse impiegano entrambi i modelli in differenti punti. Ciascun modello ha i suoi punti di forza e le sue debolezze!

8.1.1. I due modelli di navigazione

Il modello stateless definisce una mappatura tra un set di esiti di un evento e la pagina della vista. Le regole di navigazione sono interamente senza memoria rispetto allo stato mantenuto dall'applicazione oltre che alla pagina origine dell'evento. Questo significa che i metodi dell'action listener devono di tanto in tanto prendere decisioni sul pageflow, poiché solo loro hanno accesso allo stato corrente dell'applicazione.

Ecco ora un esempio di definizione di pageflow usando le regole di navigazione JSF:

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

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

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

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

</navigation-rule
>
```

Ecco lo stesso esempio di definizione di pageflow usando le regole di navigazione di Seam:

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

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

</page
>
```

Se ritieni che le regole di navigazione siano troppo lunghe, si può restituire l'id della vista direttamente dai metodi dell'action listener:

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

Si noti che questo comporta un redirect. Si possono persino specificare i parametri da usare nel redirect:

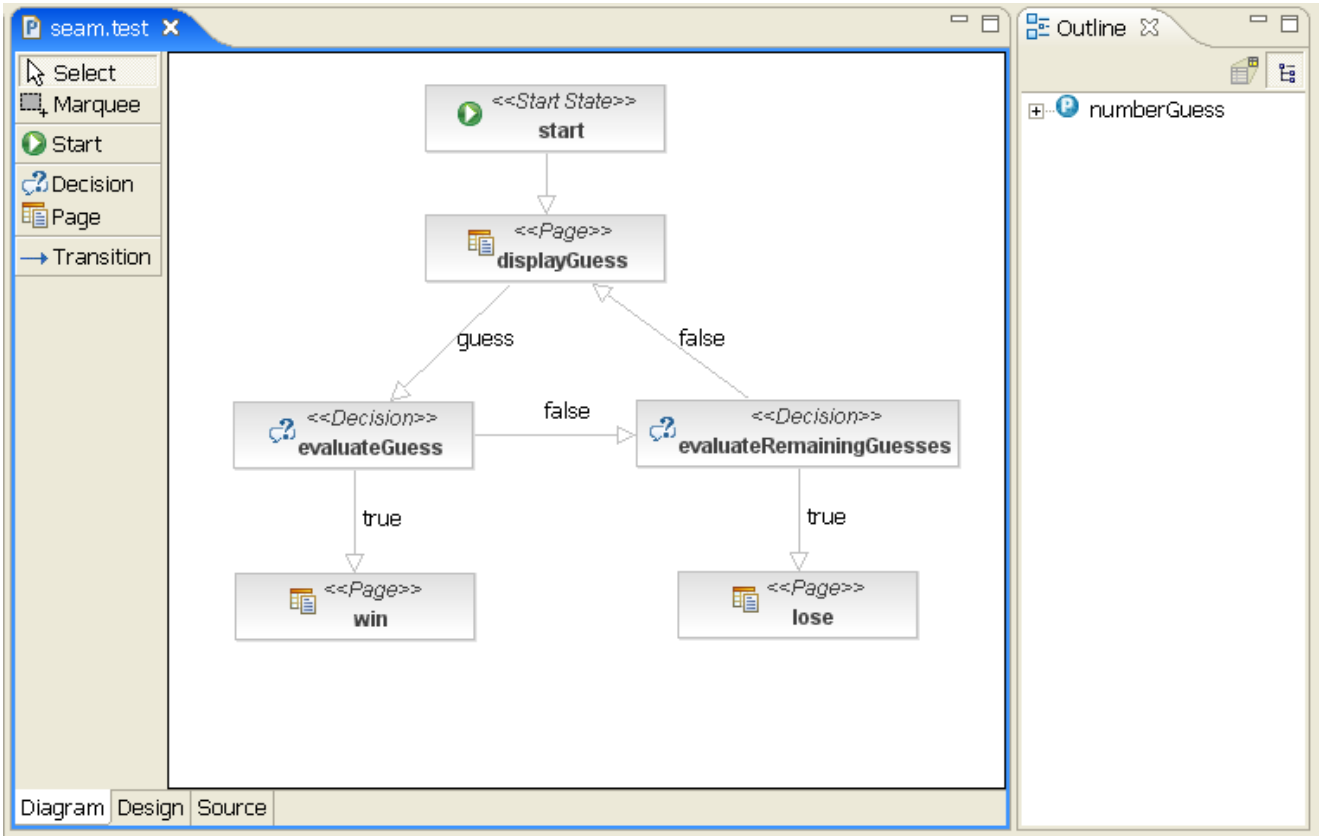
```
public String search() {
   return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}
```

Il modello stateful definisce un set di transizioni tra gli stati dell'applicazione. In questo modello è possibile esprimere il flusso di qualsiasi interazione utente interamente nella definizione jPDL di pageflow, e scrivere i metodi action listener completamente slegati dal flusso dell'interazione.

Ecco ora un esempio di definizione di pageflow usando jPDL:

```
<pageflow-definition name="numberGuess">
```

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

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

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

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

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

</pageflow-definition
>
```



Ci sono due cose da notare immediatamente:

- Le regole di navigazione JSF/Seam sono *molto* più semplici. (Comunque questo nasconde il fatto che il codice Java sottostante è molto complesso.)
- jPDL rende l'interazione utente immediatamente comprensibile senza dover guardare il codice JSP o Java.

In aggiunta il modello stateful è più *vincolato*. Per ogni stato logico (ogni passo del pageflow) c'è un set vincolato di possibili transizioni verso altri stati. Il modello stateless è un modello *ad hoc* adatto ad una navigazione libera e senza vincoli in cui l'utente decide dove andare, non l'applicazione.

La distinzione di navigazione stateful/stateless è abbastanza simile alla tradizionale vista di interazione modale/senza modello. Ora le applicazioni Seam non sono solitamente modali nel semplice senso della parola - infatti, evitare il comportamento modale dell'applicazione è uno delle principali ragioni per usare le conversazioni! Comunque le applicazioni Seam possono essere, e spesso lo sono, modali a livello di una particolare conversazione. E' noto che il comportamento modale è qualcosa da evitare il più possibile; è molto difficile predire l'ordine in cui gli utenti vogliono fare le cose! Comunque non c'è dubbio che il modello stateful ha un suo utilizzo.

Il maggior contrasto fra i due modelli è nel comportamento col pulsante indietro.

8.1.2. Seam ed il pulsante indietro

Quando le regole di navigazione di JSF o Seam vendono impiegate, Seam consente all'utente di navigare liberamente avanti ed indietro e di usare il pulsante aggiorna. E' responsabilità dell'applicazione assicurare che lo stato conversazionale rimanga internamente consistente quando questo avviene. L'esperienza con la combinazione di framework web come Struts o WebWork - che non supportano un modello conversazionale - e modelli a componenti stateless come session bean EJB stateless o il framework Spring ha insegnato a molti sviluppatori che questo è quasi praticamente impossibile da realizzare! Comunqur la nostra esperienza è che il contesto di Seam, dove c'è un modello conversazionale ben definito, agganciato a session bean stateful, è in verità abbastanza semplice. E' tanto semplice quanto combinare l'uso di `no-conversation-view-id` con controlli nulli all'inizio di metodi action listener. Riteniamo che il supporto alla navigazione libera sia quasi sempre desiderabile.

In questo casola dichiarazione `no-conversation-view-id` va in `pages.xml`. Questa dice a Seam di reindirizzare ad una pagina differente se la richiesta proviene da una pagina generata durante una conversazione, e questa conversazione non esiste più:

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

Dall'altro lato, nel modello stateful, il pulsante indietro viene interpretato come una transizione indietro ad un precedente stato. Poiché il modello stateful costringe ad un set di transizioni dallo stato corrente, il pulsante indietro viene di default disabilitato nel modello stateful! Seam rileva in modo trasparente l'uso del pulsante indietro e blocca qualsiasi tentativo di eseguire un'azione da una pagina precedente "in stallo", e reindirizza l'utente alla pagina "corrente" (mostrando un messagio faces). Se si consideri questa una funzionalità oppure una limitazione del modello stateful dipende dal proprio punto di vista: come sviluppatore è una funzionalità; come utente può essere frustrante! Si può abilitare la navigazione da un particolare nodo di pagina con il pulsante indietro impostando `back="enabled"`.

```
<page name="checkout"
      view-id="/checkout.xhtml"
      back="enabled">
   <redirect/>
   <transition to="checkout"/>
   <transition name="complete" to="complete"/>
</page
>
```

Questo permette l'uso del pulsante indietro *dallo* stato `checkout` a *qualsiasi altro stato!*

Nota

Se una pagina viene impostata per il redirect dopo una transazione, non è possibile usare il pulsante indietro per ritornare a tale pagina anche quando è abilitato su una pagina più avanti nel flow. La ragione è che Seam memorizza le informazioni sul pageflow nello scope pagina ed il pulsante indietro deve risultare in un PST affinché tale informazione sia ripristinata (cioè, richiesta Faces). Un redirect fornisce questo collegamento.

Occorre ancora definire cosa succede se una richiesta ha origine da una pagina generata durante un pageflow mentre la conversazione con il pageflow non esiste più. In questo caso la dichiarazione `no-conversation-view-id` va dentro la definizione del pageflow:

```
<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 pratica entrambi i modelli di navigazione hanno la loro utilità ed imparerai presto a riconoscere quando impiegare uno o l'altro.

8.2. Utilizzo dei pageflow jPDL

8.2.1. Installazione dei pageflow

Vanno installati i componenti di Seam relativi a jBPM e vanno collocate le definizione dei pageflow (usando l'estensione standard `.jpdl.xml`) dentro l'archivio Seam (un archivio che contiene un file `seam.properties`):

```
<bpm:jbpm />
```

Si può anche comunicare esplicitamente a Seam dove trovare le definizioni dei pageflow. Questo viene specificato dentro `components.xml`:

```
<bpm:jbpm>
   <bpm:pageflow-definitions>
```

```
      <value
>pageflow.jpdl.xml</value>
   </bpm:pageflow-definitions>
</bpm:jbpm
>
```

8.2.2. Avvio dei pageflow

Si "inizia" un pageflow basato su jPDLspecificando il nome della definizione del processo usando un'annotazione `@Begin`, `@BeginTask` oppure `@StartTask`:

```
@Begin(pageflow="numberguess")
public void begin() { ... }
```

In alternativa si può iniziare un pageflow usando pages.xml:

```
<page>
      <begin-conversation pageflow="numberguess"/>
   </page
>
```

Se il pageflow viene iniziato durante la fase `RENDER_RESPONSE` — durante un metodo `@Factory` o `@Create`, per esempio — si presume di essere già nella pagina da generare, e si usa un nodo `<start-page>` come primo nodo nel pageflow, come nell'esempio sopra.

Ma se il pageflow viene iniziato come risultato di un'invocazione di un action listener, l'esito dell'action listener determina quale è la prima pagina da generare. In questo caso si usa un `<start-state>` come primo nodo del pageflow, e si dichiara una transizione per ogni possibile esito:

```
<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
>
```

8.2.3. Nodi e transizioni di pagina

Ogni nodo `<page>` rappresenta uno stato in cui il sistema aspetta input da parte dell'utente:

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

`view-id` è l'id della vista JSF. L'elemento `<redirect/>` ha lo stesso effetto di `<redirect/>` in una regola di navigazione JSF: cioè un comportamento post-then-redirect per superare i problemi del pulsante aggiorna del browser. (Si noti che Seam propaga i contesti di conversazioni assieme a questi redirect. Quindi in Seam non serve alcun construtto "flash" dello stile di Ruby on Rails!)

Il nome della transizione è il nome dell'esito JSF lanciato cliccando un command button od un command link in `numberGuess.jsp`.

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

Quando cliccando il pulsante verrà invocata la transizione, jBPM attiverà l'azione della transizione chiamando il metodo `guess()` del componente `numberGuess`. Si noti che la sintassi usata per specificare le azioni in jPDL non è che un'espressione JSF EL già familiare, e che l'action handler della transizione è solo un metodo di un componente Seam negli attuali contesti. Così per gli eventi jBPM si ha esattamente lo stesso modello ad eventi visto per gli eventi JSF! (Il principio *Unico tipo di "cosa"*.)

Nel caso di esito nullo (per esempio un pulsante di comando senza la definizione di `action`), Seam segnalerà la transizione senza nome se ne esiste una, oppure rivisualizzerà la pagina se tutte

le transizioni hanno un nome. Così è possibile semplificare leggermente l'esempio del pageflow e questo pulsante:

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

Esegue la seguente transizione senza nome:

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

E' anche possibile che il pulsante chiami un action method, nel qual caso l'esito dell'azione determinerà la transizione da prendere:

```
<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
>
```

Comunque questo è considerato uno stile inferiore, poiché sposta la responsabilità del controllo del flusso fuori dalla definizione del pageflow e la mette negli altri componenti. E' molto meglio centralizzare questo concern nel pageflow stesso.

8.2.4. Controllo del flusso

Di solito non occorrono le funzionalità più potenti di jPDL nella definizione dei pageflow. Serve comunque il nodo `<decision>`:

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

```
>
```

Una decisione viene presa valutando un'espressione EL JSF nei contesti di Seam.

8.2.5. Fine del flusso

Si termina una conversazione usando `<end-conversation>` oppure `@End`. (Infatti per chiarezza si consiglia l'uso di *entrambi*.)"

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

Opzionalmente è possibile terminare un task specificando un nome alla `transition` jBPM. In questo modo Seam segnalerà la fine del task corrente al processo di business sottostante.

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

8.2.6. Composizione dei pageflow

E' possibile comporre pageflow ed avere una pausa in un pageflow mentre viene eseguito un altro pageflow. Il nodo `<process-state>` effettua una pausa nel pageflow più esterno e comincia l'esecuzione di un pageflow con nome:

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

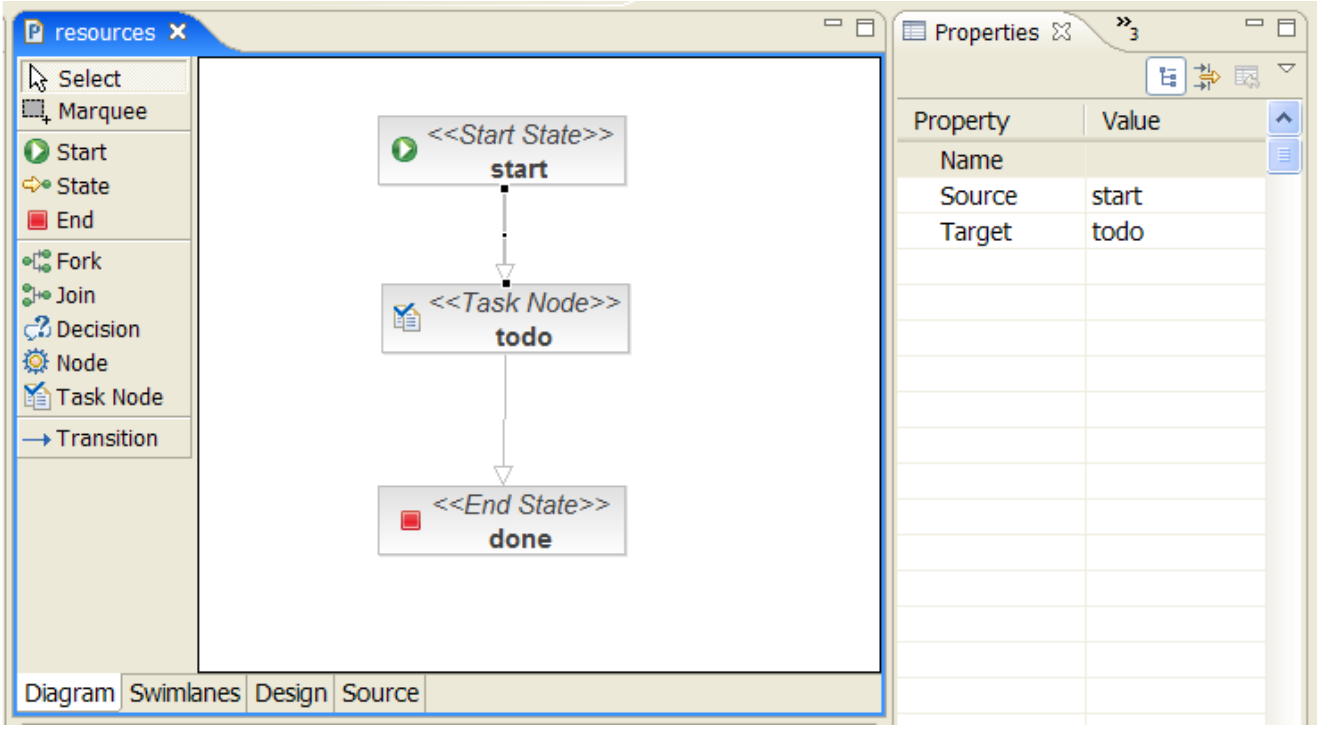
Il flusso più interno comincia l'esecuzione al nodo `<start-state>`. Quando raggiunge un nodo `<end-state>`, l'esecuzione del flusso più interno termina, e riprende quella del flusso più esterno con la transizione definita dall'elemento `<process-state>`.

8.3. La gestione del processo di business in Seam

Un processo di business è un set di task ben-definiti che deve essere eseguito dagli utenti o dai sistemi software secondo regole ben-definite riguardo *chi* può eseguire un task, e *quando*deve essere eseguito. L'integrazione jBPM di Seam facilita la visione della lista di task agli utenti e consente loro di gestire questi task. Seam consente anche all'applicazione di memorizzare lo stato associato al processo di business nel contesto `BUSINESS_PROCESS`, ed rendere questo stato persistente tramite variabili jBPM.

Una semplice definizione di processo di business appare più o meno come una definizione di pageflow (*Unico tipo di cosa*), tranne che invece dei nodi `<page>`, si hanno i nodi `<task-node>`. In un processo di business long-running, gli stati di attesa si verificano quando il sistema aspetta che un qualche utente entri ed esegua un 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
>
```



E' perfettamente possibile avere entrambi le definizioni di processo di business jPDL e le definizioni di pageflow jPDL nello stesso progetto. Se questo è il caso, la relazione tra i due è che un singolo `<task>` in un processo di business corrisponde ad un intero pageflow `<pageflow-definition>`

8.4. Uso di jPDL nella definizione del processo di business

8.4.1. Installazione delle definizioni di processo

Occorre installare jBPM ed indicare dove si trovano le definizioni dei processi di business:

```
<bpm:jbpm>
   <bpm:process-definitions>
      <value
>todo.jpdl.xml</value>
   </bpm:process-definitions>
</bpm:jbpm
>
```

Poichéi processi jBPM sono persistenti nel corso dei riavvii dell'applicazione, quando si usa Seam in ambiente di produzione non si vorrà sicuramente installare ogni volta le definizioni dei processi adogni riavvio. Quindi, in ambiente di produzione, occorre fare il deploy del processo in jBPM fuori

da Seam. In altre parole, si installano le definizioni dei processi dal file `components.xml` durante lo sviluppo dell'applicazione.

8.4.2. Inizializzazione degli actor id

Bisogna sempre sapere quale utente è attualmente loggato. jBPM "riconosce" gli utenti dal loro *actor id* e dai *group actor id*. Specificheremo gli attuali actor id usando il componente interno di Seam chiamato `actor`:

```
@In Actor actor;

public String login() {
   ...
   actor.setId( user.getUserName() );
   actor.getGroupActorIds().addAll( user.getGroupNames() );
   ...
}
```

8.4.3. Iniziare un processo di business

Per iniziare un'istanza di processo di business, si usa l'annotazione `@CreateProcess`:

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

In alternativa è possibile iniziare un processo di business usando pages.xml:

```
<page>
   <create-process definition="todo" />
</page
>
```

8.4.4. Assegnazione task

Quando un processo raggiunge un task node, vengono create istante di compiti. Queste devono essere assegnate a utenti o gruppi di utenti. Si può codificare gli id degli attori (actor id) o delegare ad un componente Seam:

```
<task name="todo" description="#{todoList.description}">
   <assignment actor-id="#{actor.id}"/>
```

```
</task
>
```

In questo caso si è semplicemente assegnato il task all'utente corrente. Possiamo anche assegnare compiti ad un pool (gruppo):

```
<task name="todo" description="#{todoList.description}">
   <assignment pooled-actors="employees"/>
</task
>
```

8.4.5. Liste di task

Parecchi componenti incorporati in Seam consentono facilmente di visualizzare liste di compiti. `pooledTaskInstanceList` è una lista di compiti che gli utenti possono assegnare a se stessi:

```
<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
>
```

Notare che invece di `<s:link>` avremmo potuto usare un semplice JSF`<h:commandLink>`:

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}"
>
   <f:param name="taskId" value="#{task.id}"/>
</h:commandLink
>
```

Il componente `pooledTask` è un componente incorporato che semplicemente assegna il task all'utente corrente.

Il componente `taskInstanceListForType` include i task di un particolare tipo che sono stati assegnati all'utente corrente:

```
<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
>
```

8.4.6. Esecuzione di un task

Per iniziare a lavorare ad un compito, si può usare o `@StartTask` o `@BeginTask` sul metodo listener.

```
@StartTask
public String start() { ... }
```

In alternativa si può iniziare a lavorare su un task usando pages.xml:

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

Queste annotazioni iniziano uno speciale tipo di conversazione che ha significato in termini di processo di business. Il lavoro fatto da questa conversazione ha accesso allo stato mantenuto nel contesto di business process.

Se si termina una conversazione usando `@EndTask`, Seam segnalerà il completamento del task:

```
@EndTask(transition="completed")
public String completed() { ... }
```

In alternativa si usa pages.xml:

```
<page>
   <end-task transition="completed" />
</page
>
```

Si può anche usare EL per specificare la transizione in pages.xml.

A questo punto jBPM assume il controllo e continua l'esecuzione della definizione del processo di business. (In processi più complessi, parecchi task potrebbero aver bisogno di essere completati prima che l'esecuzione del processo possa riprendere.)

Fare riferimento alla documentazione jBPM per una panoramica delle funzionalità che jBPM fornisce per la gestione di processi complessi di business.

Seam e Object/Relational Mapping

Seam fornisce un supporto esteso alle due maggiori e più popolari architetture per la persistenza in Java: Hibernate3 e Java Persistence API introdotta con EJB 3.0. L'architettura unica di Seam per la gestione dello stato consente l'integrazione dei più sofisticati ORM di ogni framework per applicazioni web.

9.1. Introduzione

Seam è nato dalla frustrazione del team di Hibernate per l'assenza di contesto stateless tipica delle precedenti generazioni di architetture nelle applicazioni Java. L'architettura della gestione dello stato di Seam è stata originariamente progettata per risolvere problemi relativi alla persistenza — in particolare i problemi associati all'*elaborazione ottimistica delle transazioni*. Le applicazioni online scalabili usano sempre transazioni ottimistiche. Una transazione atomica di livello (database/JTA) non dovrebbe propagare l'interazione dell'utente amenoché l'applicazione sia progettata per supportare solo un piccolo numero di client concorrenti. Ma quasi tutto il lavoro interessante coinvolge in primo luogo la visualizzazione dei dati all'utente, e poi, immediatamente dopo, l'aggiornamento dei dati stessi. Quindi Hibernate è stato progettato per supportare l'idea del contesto di persistenza che propaga una transazione ottimistica.

Sfortunatamente le cosiddette architetture "stateless" che precedettero Seam e EJB 3.0 non avevano alcun costrutto per rappresentare una transazione ottimistica. Quindi, invece, queste architetture fornivano contesti di persistenza con scope a livello di transazione atomica. Sicuramente questo portava diversi problemi agli utenti ed è la causa numero uno per le lamentele riguardanti Hibernate: la temuta `LazyInitializationException`. Ciò di cui si ha bisogno è un costrutto per rappresentare una transazione ottimistica a livello applicazione.

EJB 3.0 riconosce il problema e introduce l'idea di componente stateful (un bean di sessione stateful) con un *contesto di persistenza esteso* con scope legato al ciclo di vita del componente. Questa è una soluzione parziale al problema (ed è un utile costrutto), comunque ci sono due problemi:

- Il ciclo di vita del bean di sessione stateful deve essere gestito manualmente via codice a livello web (risulta che questo è un problema sottile e molto più difficile in pratica di quanto sembri).
- La propagazione del contesto di persistenza tra componenti stateful nella stessa transazione ottimistica è possibile, ma pericolosa.

Seam risolve il primo problema fornendo conversazioni, componenti bean di sessione stateful con scope di conversazione. (La maggior parte delle conversazioni in verità rappresentano transazioni ottimistiche a livello dei dati). Questo è sufficiente per molte semplici applicazioni (quali la demo prenotazione di Seam) dove non serve la propagazione del contesto di persistenza. Per applicazioni più complesse, con molti componenti interagenti in modo stretto in ciascuna conversazione, la propagazione del contesto di persistenza tra componenti diventa un problema importante. Quindi Seam estende il modello di gestione del contesto di persistenza di EJB 3.0 per fornire contesti di persistenza estesi e con scope di conversazione.

9.2. Transazioni gestite da Seam

I bean di sessione EJB includono la gestione dichiarativa delle transazioni. Il container EJB è capace di avviare una transazione in modo trasparente quando viene invocato il bean, e terminarla quando termina l'invocazione. Se si scrive un metodo di un bean di sessione che agisce come action listener JSF, si può fare tutto il lavoro associato all'azione in una transazione, ed essere sicuri che venga eseguito il commit od il rollback quando l'azione viene terminata. Questa è grande funzionalità ed è tutto ciò che serve ad alcune applicazioni Seam.

Comunque c'è un problema con tale approccio. Un'applicazione Seam potrebbe non eseguire l'accesso a tutti i dati per una richiesta da una chiamata di un singolo metodo a un bean di sessione.

- La richiesta può comportare di essere processata da diversi componenti poco accoppiati, ciascuno dei quali viene chiamato indipendentemente dal layer web. E' comune vedere parecchie chiamate per richiesta dal layer web ai componenti EJB in Seam.
- La generazione della vista può richiedere il lazy fetching delle associazioni.

Più transazioni per richiesta ci sono, più è probabile che si incontrino problemi di atomicità e isolamento quando l'applicazione processa molte richieste concorrenti. Certamente tutte le operazioni di scrittura devono avvenire nella stessa transazione!

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 `LazyInitializationExceptions` 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 vaues 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.

La gestione delle transazioni di Seam è utile anche se vengono usati contesti di persistenza gestiti da un container EJB 3.0. Ma in particolare essa è utile quando Seam è usato fuori dall'ambiente Java EE 5, o in ogni altro caso dove si usi un contesto di persistenza gestito da Seam.

9.2.1. Disabilitare le transazioni gestite da Seam

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 />
```

9.2.2. Configurazione di un gestore di transazioni Seam

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 5 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* .

9.2.3. Sincronizzazione delle transazioni

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 5 environment the `<transaction:ejb-transaction/>` component should be 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.

9.3. Contesti di persistenza gestiti da Seam

If you're using Seam outside of a Java EE 5 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 persisence 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.)

9.3.1. 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.

9.3.2. Uso delle sessioni Hibernate gestite da Seam

Le sessioni Hibernate gestite da Seam sono simili. In `components.xml`:

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

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

Dove `java:/bookingSessionFactory` è il nome della session factory specificata in `hibernate.cfg.xml`.

```
<session-factory name="java:/bookingSessionFactory">
   <property name="transaction.flush_before_completion"
>true</property>
   <property name="connection.release_mode"
>after_statement</property>
   <property name="transaction.manager_lookup_class"
>org.hibernate.transaction.JBossTransactionManagerLookup</property>
   <property name="transaction.factory_class"
>org.hibernate.transaction.JTATransactionFactory</property>
   <property name="connection.datasource"
>java:/bookingDatasource</property>
   ...
</session-factory
>
```

Note 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;
```

9.3.3. 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 `FlushModeType`s 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" />
```

Si può impostare qualsiasi Contesto di Persistenza Gestito da Seam alla modalità flush manuale:

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core">
  <core:manager conversation-timeout="120000" default-flush-mode="manual" />
</components
>
```

9.4. Usare il 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}"/>
```

Ora si può iniettare la sessione direttamente:

```
@In Session session;

@Create
public void init() {
   session.enableFilter("currentVersions");
}
```

9.5. Uso di 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();
```

è equivalente a:

```
User user = em.createQuery("from User where username=:username")
        .setParameter("username", user.getUsername())
        .getSingleResult();
```

Certamente non si dovrà mai e poi mai scrivere qualcosa del tipo:

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

(è inefficiente e vulnerabile ad attacchi di SQL injection.)

9.6. Uso dei filtri Hibernate

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.

I contesti di persistenza gestiti da Seam possono avere una lista di filtri definiti, che verrà abilitata quando viene creato un `EntityManager` od una `Session` di Hibernate. (Certamente possono essere utilizzati solo quando Hibernare è il provider di persistenza sottostante.)

```
<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
>
```

Validazione delle form JSF in Seam

Nel puro JSF la validazione è definita nella vista:

```
<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 pratica quest'approccio di norma viola il principio DRY (Don't Repeat Yourself = Non ripeterti), poiché la maggior parte della "validazione" forza i vincoli che sono parte del modello di dati, e che esistono lungo tutta la definizione dello schema di database. Seam fornisce supporto ai vincoli del modello definiti, utilizzando Hibernate Validator.

Si cominci col definire i vincoli sulla classe `Location`:

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

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

   @NotNull
```

```
    @Length(max=6)
    @Pattern("^\d*$")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

Bene, questa è una prima buona riduzione, ma in pratica è possibile essere più eleganti utilizzando dei vincoli personalizzati invece di quelli interni a Hibernate Validator:

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

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

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

Qualsiasi strada si prenda non occorre più specificare il tipo di validazione da usare nelle pagine JSF. Invece è possibile usare `<s:validate>` per validare il vincolo definito nell'oggetto modello.

```
<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
>
```

Nota: specificare `@NotNull` nel modello *non* elimina la necessità di `required="true"` per farlo apparire nel controllo! Questo è dovuto ad una limitazione nell'architettura di validazione in JSF.

Quest'approccio *definisce* i vincoli sul modello e *presenta* le violazioni al vincolo nella vista — di gran lunga un miglior design.

Comunque non è molto meno corto di quanto lo era all'inizio, quindi proviamo `<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
>
```

Questo tag semplicemente aggiunge un `<s:validate>` ad ogni input nella form. Per form grandi questo fa risparmiare molto!

Adesso occorre fare qualcosa per mostrare all'utente un messaggio quando la validazione fallisce. Ora vengono mostrati tutti i messaggi in cima alla form. Per consentire all'utente di associare il

messaggio al singolo input, occorre definire un'etichetta e usare l'attributo standard `label` sul componente d'input.

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

Si può iniettare questo valore nella stringa del messaggio usando il placeholder {0} (il primo ed unico parametro passato al messaggio JSF a causa di una restrizione in Hibernate Validator). Vedere la sezione Internazionalizzazione per ulteriori informazioni sulla definizione dei messaggi.

```
validator.length={0} la lunghezza deve essere tra {min} e {max}
```

Ciò che si vuole fare, tuttavia, è mostrare il messaggio vicino al campo con l'errore (questo è possibile nel semplice JSF), ed evidenziare il campo e l'etichetta (questo non è possibile) ed, eventualmente, mostrare un'immagine vicino al campo (anche questo non è possibile). Si vuole anche mostrare un piccolo asterisco colorato vicino all'etichetta per ciascun campo richiesto. Utilizzando quest'approccio non è più necessario identificare l'etichetta.

Si ha quindi bisogno di parecchia funzionalità per ogni singolo campo della form. Ma non si vuole essere costretti a specificare per ogni campo della form come evidenziare, come disporre l'immagine, quale messaggio scrivere e quale è il campo d'input da associare.

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

   <div>

      <s:label styleClass="#{invalid?'error':''}">
         <ui:insert name="label"/>
         <s:span styleClass="required" rendered="#{required}"
>*</s:span>
      </s:label>

      <span class="#{invalid?'error':''}">
         <h:graphicImage value="/img/error.gif" rendered="#{invalid}"/>
         <s:validateAll>
```

```
            <ui:insert/>
          </s:validateAll>
        </span>

        <s:message styleClass="error"/>

    </div>

</ui:composition
>
```

Si può includere questo template per ciascun campo della form utilizzando `<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
>
```

Infine è possibile utilizzare RichFaces Ajax per mostrare i messaggi di validazione mentre l'utente naviga nella form:

```
<h:form>

    <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
```

```
      <ui:define name="label"
>Country:</ui:define>
      <h:inputText value="#{location.country}" required="true">
         <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
      </h:inputText>
   </s:decorate>

   <s:decorate id="zipDecoration" template="edit.xhtml">
      <ui:define name="label"
>Zip code:</ui:define>
      <h:inputText value="#{location.zip}" required="true">
         <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
      </h:inputText>
   </s:decorate>

   <h:commandButton/>

</h:form
>
```

E' meglio definire esplicitamente gli id per i controlli importanti all'interno di una pagina, specialmente in caso di un test automatico della UI, utilizzando dei toolkit quale Selenium. Se non vengono forniti gli id in modo esplicito, JSF li genererà, ma i valori generati cambieranno se si cambia qualcosa nella pagina.

```
<h:form id="form">

   <h:messages globalOnly="true"/>

   <s:decorate id="countryDecoration" template="edit.xhtml">
      <ui:define name="label"
>Country:</ui:define>
      <h:inputText id="country" value="#{location.country}" required="true">
         <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
      </h:inputText>
   </s:decorate>

   <s:decorate id="zipDecoration" template="edit.xhtml">
      <ui:define name="label"
>Zip code:</ui:define>
      <h:inputText id="zip" value="#{location.zip}" required="true">
         <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
      </h:inputText>
```

```
    </s:decorate>

    <h:commandButton/>

</h:form
>
```

E se si vuole specificare un messaggio diverso quando la validazione fallisce? Si può utilizzare il message bundle di Seam con Hibernate Validator (e tutte le altre finezze come le espressioni EL dentro il messaggio ed i message bundle per ogni singola vista).

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

    // Getters and setters for name

    @NotNull
    @Length(max=6)
    @ZipCode(message="#{messages['location.zipCode.invalid']}")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

```
location.zipCode.invalid = Codice ZIP non valido per #{location.name}
```

Integrazione con Groovy

Uno degli aspetti di JBoss Seam è la caratteristica RAD (Rapid Application Development). Benché i linguaggi dinamici non siano un sinonimo di RAD, in questo ambito essi sono uno degli aspetti più interessanti. Fino a poco tempo fa scegliere un linguaggio dinamico richiedeva anche di scegliere una piattaforma di sviluppo completamente differente (una piattaforma di sviluppo con un insieme di API ed un ambiente runtime così conveniente da non voler più tornare ad usare le vecchie API Java, con la fortuna di essere costretti ad usare in ogni caso quelle API proprietarie). I linguaggi dinamici costruiti sulla Java Virtual Machine, e *Groovy* [http://groovy.codehaus.org] in particolare, hanno rotto questo approccio alla grande.

Oggi JBoss Seam unisce il mondo dei linguaggi dinamici con il mondo Java EE integrando perfettamente sia i linguaggi statici che quelli dinamici. JBoss Seam lascia che lo sviluppatore scelga il migliore strumento per ciò che deve fare, senza cambiare contesto. Scrivere componenti Seam dinamici è esattamente come scrivere componenti Seam normali. Si usano le stesse annotazioni, le stesse API, lo stesso di tutto.

11.1. Introduzione a Groovy

Groovy è un linguaggio dinamico agile basato sul linguaggio Java, ma con alcune caratteristiche addizionali ispirate da Python, Ruby e Smalltalk. Il punto di forza di Groovy è duplice:

- La sintassi Java è supportata in Groovy: il codice Java è codice Groovy, e ciò rende il processo di apprendimento molto semplice.
- Gli oggetti Groovy sono oggetti Java e le classi Groovy sono classi Java: Groovy si integra semplicemente con le librerie e i framework Java esistenti.

TODO: scrivere un breve riassunto delle caratteristiche specifiche della sintassi Groovy.

11.2. Scrivere applicazioni Seam in Groovy

Non c'è molto da dire su questo. Poiché un oggetto Groovy è un oggetto Java, è virtualmente possibile scrivere qualsiasi componente Seam, così come qualsiasi altra classe del resto, in Groovy e metterla in funzione. E' anche possibile fare un misto di classi Groovy e classi Java nella stessa applicazione.

11.2.1. Scrivere componenti Groovy

Come è stato possibile notare finora, Seam usa pesantemente le annotazioni. Assicurarsi di usare Groovy 1.1 o una versione successiva per avere il supporto delle annotazioni. Di seguito ci sono alcuni esempi di codice Groovy utilizzato in una applicazione Seam.

11.2.1.1. Entità

```
@Entity
```

```
@Name("hotel")
class Hotel implements Serializable
{
    @Id @GeneratedValue
    Long id

    @Length(max=50) @NotNull
    String name

    @Length(max=100) @NotNull
    String address

    @Length(max=40) @NotNull
    String city

    @Length(min=2, max=10) @NotNull
    String state

    @Length(min=4, max=6) @NotNull
    String zip

    @Length(min=2, max=40) @NotNull
    String country

    @Column(precision=6, scale=2)
    BigDecimal price

    @Override
    String toString()
    {
        return "Hotel(${name},${address},${city},${zip})"
    }
}
```

Groovy supporta nativamente il concetto di proprietà (getter/setter), perciò non c'è bisogno di scrivere esplicitamente il ripetitivo codice dei getter e setter: nell'esempio precedente, la classe hotel può essere utilizzata da Java come `hotel.getCity()`, i metodi getter e setter vengono generati dal compilatore Groovy. Questo tipo di facilitazione sintattica rende il codice delle entità molto conciso.

11.2.1.2. Componenti Seam

Scrivere componenti Seam in Groovy non è diverso da farlo in Java: le annotazioni sono utilizzate per marcare la classe come un componente Seam.

```
@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])
   }
}
```

11.2.2. seam-gen

Seam gen ha una integrazione trasparente rispetto a Groovy. E' possibile scrivere codice Groovy in un progetto strutturato da seam-gen senza alcun requisito infrastrutturale addizionale. Se vengono scritte entità in Groovy è sufficiente posizionare i file `.groovy` in `src/main`. Allo stesso modo, quando vengono scritte delle azioni, è sufficiente posizionare i file `.groovy` in `src/hot`.

11.3. Esecuzione

Eseguire classi Groovy è molto simile ad eseguire classi Java (soprendentemente non c'è bisogno di scrivere o di essere compatibili con qualche complessa specifica a 3 lettere per supportare più linguaggi nei componenti di framework).

Al di là della modalità standard di esecuzione, JBoss Seam ha l'abilità, durante lo sviluppo, di sostituire componenti Seam JavaBeans senza bisogno di riavviare l'applicazione, risparmiando molto tempo nel ciclo di sviluppo e test. Lo stesso supporto è fornito per i componenti Seam GroovyBeans quando i file `.groovy` vengono eseguiti.

11.3.1. Eseguire il codice Groovy

Una classe Groovy *è* una classe Java, con una rappresentazione bytecode esattamente come una classe Java. Per eseguire un'entità Groovy, un Session Bean Groovy o un componente Seam Groovy, è necessario un passaggio di compilazione. Un approccio diffuso è quello di usare il task ant `groovyc`. Una volta compilata, una classe Groovy non presenta alcuna differenza rispetto ad una classe Java e l'application server le tratterà nello stesso modo. Notare che questo consente di utilizzare un misto di codice Groovy e Java.

11.3.2. Esecuzione di file .groovy durante lo sviluppo

JBoss Seam supporta l'esecuzione diretta di file `.groovy` (cioè senza compilazione) nella modalità di hot deployment incrementale (solo per lo sviluppo). Ciò consente un ciclo di modifica/test molto rapido. Per impostare l'esecuzione dei file .groovy, seguire la configurazione indicata in *Sezione 2.8, «Seam e hot deploy incrementale»* ed eseguire il codice Groovy (i file `.groovy`) nella cartella `WEB-INF/dev`. I componenti GroovyBean verranno presi in modo incrementale senza bisogno di riavviare l'applicazione (e ovviamente neanche l'application server).

Fare attenzione al fatto che l'esecuzione diretta dei file .groovy soffre delle stesse limitazioni del normale hot deployment di Seam:

- I componenti devono essere JavaBeans o GroovyBeans. Non possono essere componenti EJB3.
- Le entità non possono essere eseguite in modalità hot deploy.
- I componenti da eseguire in modalità hotdeploy non saranno visibili ad alcuna classe posizionata al di fuori di `WEB-INF/dev`
- La modalità debug di Seam deve essere attivata

11.3.3. seam-gen

Seam-gen supporta la compilazione e l'esecuzione dei file Groovy in modo trasparente. Questo include l'esecuzione diretta dei file `.groovy` durante lo sviluppo (senza compilazione). Creando un progetto seam-gen di tipo WAR, le classi Java e Groovy posizionate in `src/hot` saranno automaticamente candidate per l'esecuzione incrementale in modalità hot deploy. In modalità di produzione, i file Groovy saranno semplicemente compilati prima dell'esecuzione.

In `examples/groovybooking` è possibile trovare un esempio della demo Booking scritto completamente in Groovy con il supporto per l'esecuzione incrementale in modalità hot deploy

Scrivere la parte di presentazione usando Apache Wicket

Seam supporta Wicket come uno strato di presentazione alternativo a JSF. Si guardi l'esempio `wicket` in Seam che illustra l'esempio Booking riscritto per Wicket.

Nota

Il supporto Wicket è nuovo in Seam, perciò alcune caratteristiche che sono disponibili in JSF non sono ancora disponibili quando viene usato Wicket (ad esempio il pageflow). Si potrà notare inoltre che la documentazione è molto centrata su JSF e necessita di una riorganizzazione per riflettere il supporto di prima categoria per Wicket.

12.1. Aggiungere Seam ad un'applicazione Wicket

Le caratteristiche aggiunte ad un'applicazione Wicket possono essere divise in due categorie: la bijection e l'orchestrazione. Esse vengono discusse in dettaglio di seguito.

E' comune un uso intensivo di classi interne (inner class) quando si costruisce un'applicazione Wicket, in cui l'albero dei componenti viene generato nel costruttore. Seam gestisce pienamente l'uso di annotazioni di controllo nelle inner class e nei costruttori (a differenza che nei componenti Seam normali).

Le annotazioni sono elaborate *dopo* ogni chiamata alla superclasse. Ciò significa che qualsiasi attributo iniettato non può essere passato come argomento in una chiamata a `this()` o `super()`.

Nota

Stiamo lavorando per migliorare questo aspetto.

Quando un metodo è chiamato in una inner class, la bijection avviene per ogni classe che la contiene. Ciò consente di posizionare le variabili bi-iniettabili nella classe esterna e riferirsi ad esse in qualsiasi inner class.

12.1.1. Bijection

Un'applicazione Wicket abilitata per Seam ha accesso completo a tutti i contesti Seam standard (`EVENT`, `CONVERSATION`, `SESSION`, `APPLICATION` e `BUSINESS_PROCESS`).

Per accedere ai componenti Seam da Wicket basta iniettarli usando `@In`:

```
@In(create=true)
private HotelBooking hotelBooking;
```

Suggerimento

Poiché la classe Wicket non è un componente Seam completo, non c'è bisogno di annotarla con `@Name`.

E' anche possibile fare l'outject di un oggetto da un componente Wicket nei contesti Seam:

```
@Out(scope=ScopeType.EVENT, required=false)
private String verify;
```

TODO Rendere questa parte più orientata allo use case

12.1.2. Orchestrazione

E' possibile abilitare la sicurezza di un componente Wicket usando l'annotazione `@Restrict`. Questa può essere messa nel componente più esterno o in qualsiasi componente interno. Se viene indicato `@Restrict`, l'accesso al componente verrà automaticamente limitato agli utenti registrati. Facoltativamente è possibile usare un'espressione EL nell'attributo `value` per specificare la restrizione da applicare. Per maggiori dettagli vedi *Capitolo 15, Sicurezza*.

Ad esempio:

```
@Restrict
public class Main extends WebPage {

  ...
```

Suggerimento

Seam applicherà automaticamente la restrizione ad ogni classe interna.

E' possibile demarcare le conversazioni da un componente Wicket attraverso l'uso di `@Begin` e `@End`. La semantica di queste annotazioni è la stessa di quando sono usate nei componenti Seam. E' possibile mettere `@Begin` e `@End` in qualsiasi metodo.

Nota

L'attributo deprecato `ifOutcome` non è gestito.

Ad esempio:

```
item.add(new Link("viewHotel") {

  @Override
  @Begin
  public void onClick() {
    hotelBooking.selectHotel(hotel);
    setResponsePage(org.jboss.seam.example.wicket.Hotel.class);
  }
};
```

E' possibile che l'applicazione abbia delle pagine che possono essere visitate solo quando l'utente ha una conversazione lunga (long-running conversation) attiva. Per imporre questo criterio è possibile usare l'annotazione `@NoConversationPage`:

```
@Restrict
@NoConversationPage(Main.class)
public class Hotel extends WebPage {
```

Se si vogliono ulteriormente disaccoppiare le classi dell'applicazione è possibile usare gli eventi Seam. Naturalmente è possibile lanciare un evento usando `Events.instance().raiseEvent("pippo")`. In alternativa è possibile annotare un metodo con `@RaiseEvent("pippo")`; se il metodo restituisce un valore non nullo senza eccezioni, l'evento verrà lanciato.

E' anche possibile controllare task e processi nelle classi Wicket attraverso l'uso di `@CreateProcess`, `@ResumeTask`, `@BeginTask`, `@EndTask`, `@StartTask` e `@Transition`.

TODO - Realizzare il controllo BPM - JBSEAM-3194

12.2. Impostare il progetto

Seam ha bisogno di intervenire sul bytecode delle classi Wicket in modo da poter intercettare le annotazioni usate. Seam fornisce due modi per fare questo. Il primo è di mettere le classi in `WEB-INF/wicket`. Seam cercherà le classi messe in questa cartella durante l'avvio e interverrà sul loro bytecode. Un approccio alternativo, che può essere utilizzato insieme al primo, è di usare

un task ant per modificare il bytecode. Seam fornisce questo task: è contenuto in `jboss-seam-wicket-ant.jar` e può essere usato nel seguento modo:

```
<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="${build.instrumented}">
  <classpath refid="build.classpath"/>
  <fileset dir="${build.classes}" includes="**/*.class"/>
</instrumentWicket
>
```

Dopo bisogna fare in modo che Ant copi le classi alterate da `${build.instrumented}` a `WEB-INF/classes`. Se si vuole attivare l'esecuzione a caldo dei componenti Wicket è possibile copiare le classi alterate in `WEB-INF/hot`. Se si usa l'esecuzione a caldo, accertarsi che anche la classe `WicketApplication` sia eseguita nello stesso modo. Dopo che le classi eseguite a caldo vengono ricaricate, l'intera istanza di WicketApplication deve essere reinizializzata allo scopo di recuperare tutti i nuovi riferimenti alle classi delle pagine montate.

12.2.1. Definire l'applicazione

Un'applicazione web Wicket che usa Seam deve usare `SeamWebApplication` come classe base. Questa crea gli agganci nel ciclo Wicket che consentono a Seam di propagare auto-magicamente la conversazione quando necessario. Aggiunge pure i messaggi di stato alla pagina.

Ad esempio:

La `SeamAuthorizationStrategy` delega le autorizzazioni a Seam Security, consentendo l'uso di `@Restrict` nei componenti Wicket. `SeamWebApplication` provvede ad installare la strategia di autorizzazioni. E' possibile specificare una pagina di login implementando il metodo `getLoginPage()`.

C'è poi bisogno di impostare la home page dell'applicazione implementando il metodo `getHomePage()`.

```
public class WicketBookingApplication extends SeamWebApplication {
```

```
  @Override
  public Class getHomePage() {
    return Home.class;
  }

  @Override
  protected Class getLoginPage() {
    return Home.class;
  }

}
```

Seam installa automaticamente il filtro Wicket (assicurando che sia inserito nella posizione corretta), ma è ancora necessario indicare a Wicket quale classe `WebApplication` usare:

```
<components xmlns="http://jboss.com/products/seam/components"
 xmlns:wicket="http://jboss.com/products/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.com/products/seam/wicket
   http://jboss.com/products/seam/wicket-2.1.xsd">

  <wicket:web-application
    application-class="org.jboss.seam.example.wicket.WicketBookingApplication" />
</components
```

In aggiunta se si pensa di usare le pagine basate su JSF in un'appliczione con pagine wicket, bisogna assicurarsi che il filtro delle eccezioni jsf sia abilitato per gli url jsf:

```
<components xmlns="http://jboss.com/products/seam/components"
 xmlns:web="http://jboss.com/products/seam/web"
 xmlns:wicket="http://jboss.com/products/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.com/products/seam/web
   http://jboss.com/products/seam/web-2.1.xsd">

    <!-- Si mappa solo il filtro per le eccezioni jsf di seam nel path jsf, che viene identificato con
 il path *.seam -->
       <web:exception-filter url-pattern="*.seam"/>
</components
```

Suggerimento

Per maggiori informazioni sulle strategie di autorizzazione e gli altri metodi che possono essere implementati sulla classe `Application` fare riferimento alla documentazione Wicket.

Seam Application Framework

Seam semplifica la creazione di applicazioni tramite la scrittura di classi Java semplici con annotazioni, che non hanno bisogno di estendere speciali interfacce o superclassi. Ma è possibile semplificare ulteriormente alcuni comuni compiti di programmazione, fornendo un set di componenti predefiniti che possono essere riutilizzati o tramite configurazione in `components.xml` (per casi molto semplici) o tramite estensione.

Seam Application Framework può ridurre la quantità di codice da scrivere nel fornire l'accesso ai database nelle applicazioni web, usando Hibernate o JPA.

Sottolineiamo che il framework è estremamente semplice, solamente una manciata di classi molto semplici, facili da capire e da estendere. La "magia" è in Seam stesso — la stessa magia che si usa nel creare un'applicazione Seam anche senza usare questo framework.

13.1. Introduzione

I componenti forniti dal framework Seam possono essere usati secondo due differenti approcci. Il primo modo è installare e configurare un'istanza del componente in `components.xml`, come si è fatto con altri tipi di componenti Seam predefiniti. Per esempio, il seguente frammento da `components.xml` installa un componente che esegue semplici operazioni CRUD per un'entità `Person`:

```
<framework:entity-home name="personHome"
                       entity-class="eg.Person"
                       entity-manager="#{personDatabase}">
   <framework:id
>#{param.personId}</framework:id>
</framework:entity-home
>
```

Se per i propri gusti tutto questo sembra troppo "programmare in XML", è possibile altrimenti usare l'estensione:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {

   @In EntityManager personDatabase;

   public EntityManager getEntityManager() {
      return personDatabase;
```

```
  }

}
```

Il secondo approccio ha un vantaggio enorme: si possono facilmente aggiungere funzionalità extra ed eseguire l'override di funzionalità predefinite (le classi del framework sono state attentamente progettate per l'estensione e la personalizzazione).

Un secondo vantaggio è che le classi possono essere bean di sessione stateful EJB, se si vuole. (Non devono per forza esserlo, se si vuole possono essere componenti JavaBean semplici.) Se si sta usando JBoss AS, serve la versione 4.2.2.GA o successive:

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person
> implements LocalPersonHome {

}
```

Si possono rendere le proprie classi bean di sessione stateless. In questo caso *occorre* usare l'injection per fornire il contesto di persistenza, anche se viene chiamato l'`entityManager`:

```
@Stateless
@Name("personHome")
public class PersonHome extends EntityHome<Person
> implements LocalPersonHome {

  @In EntityManager entityManager;

  public EntityManager getPersistenceContext() {
    entityManager;
  }

}
```

Attualmente Seam fornisce quattro componenti predefiniti: `EntityHome` e `HibernateEntityHome` per le operazioni CRUD, assieme a `EntityQuery` e `HibernateEntityQuery` le query.

I componenti Home e Query sono scritti per funzionare con scope di sessione, evento o conversazione. Quale scope usare dipende dal modello di stato che si desidera usare nella propria applicazione.

Seam Application Framework funziona solo con contesti di persistenza gestiti da Seam. Di default i componenti cercano un contesto di persistenza chiamato `entityManager`.

13.2. Oggetti Home

Un oggetto Home fornisce operazioni per la persistenza per una particolare classe entity. Si supponga di avere una classe `Person`:

```
@Entity
public class Person {
   @Id private Long id;
   private String firstName;
   private String lastName;
   private Country nationality;

   //getters and setters...
}
```

E' possibile definire un componente `personHome` o via configurazione:

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

O tramite estensione:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {}
```

Un oggetto Home fornisce le seguenti operazioni: `persist()`, `remove()`, `update()` e `getInstance()`. Prima di chiamare le operazioni `remove()`, o `update()`, occorre prima impostare l'identificatore dell'oggetto interessato, usando il metodo `setId()`.

Si può usare un Home direttamente da una pagina JSF, per esempio:

```
<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
>
```

Di solito è più comodo poter fare riferimento a `Person` semplicemente come `person`, e quindi si aggiunga una linea a `components.xml`:

```
<factory name="person"
        value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                    entity-class="eg.Person" />
```

(Se si usa la configurazione.) O si aggiunga un metodo `@Factory` a `PersonHome`:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {

   @Factory("person")
   public Person initPerson() { return getInstance(); }

}
```

(Se si usa l'estensione.) Questo cambiamento semplifica le pagine JSF come segue:

```
<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
```

```
>
```

Bene, queste crea le nuove entry per `Person`. Esatto, è tutto quanto il codice che serve! Ora se si vuole mostrare, aggiornare e cancellare entry di `Person` già esistenti nel database, occorre passare l'identificatore delle entry a `PersonHome`. I parametri di pagina sono un eccezionale modo per farlo:

```
<pages>
   <page view-id="/editPerson.jsp">
      <param name="personId" value="#{personHome.id}"/>
   </page>
</pages
>
```

Ora possiamo aggiungere operazioni extra alle pagine JSF:

```
<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
>
```

Quando ci si collega alla pagina senza parametri di richiesta, la pagina verrà mostrata come una pagina "Create Person". Quando si fornisce un valore per il parametro di richiesta `personId`, sarà una pagina "Edit Person".

Si supponga di dover creare entry di `Person` con la nazionalità inizializzata. E' possibile farlo semplicemente via configurazione:

```
<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
>
```

O tramite estensione:

```
@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);
   }

}
```

Certamente `Country` può essere un oggetto gestito da un altro oggetto Home, per esempio, `CountryHome`.

Per aggiungere altre operazioni sofisticate (gestione dell'associazione, ecc.) si possono aggiungere dei metodi a `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();
   }

}
```

L'oggetto Home solleva un'evento `org.jboss.seam.afterTransactionSuccess` quando una transazione ha successo (una chiamata a `persist()`, `update()` o `remove()` ha successo). Osservando questo evento si può fare il refresh delle query quando le entità sottostanti cambiano. Se si vuole solo eseguire il refresh quando una particolare entità viene persistita, aggiornata o rimossa, si può osservare l'evento `org.jboss.seam.afterTransactionSuccess.<name>` (dove `<name>` è il nome dell'entity).

L'oggetto Home mostra automaticamente i messaggi faces quando un'operazione ha successo. Per personalizzare questi messaggi si può ancora usare la configurazione:

```
<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
>
```

O estensione:

```
@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"); }

}
```

Ma il modo migliore per specificare i messaggi è metterli in un resource bundle noto a Seam (di default, il nome del bundle è `messages`).

```
Person_created=New person #{person.firstName} #{person.lastName} created
Person_deleted=Person #{person.firstName} #{person.lastName} deleted
Person_updated=Person #{person.firstName} #{person.lastName} updated
```

Questo abilita l'internazionalizzazione e mantiene il codice e la configurazione puliti dagli elementi di presentazione.

Il passo finale è aggiungere alla pagina la funzionalità di validazione, usando `<s:validateAll>` e `<s:decorate>`, ma verrà lasciato al lettore come esercizio.

13.3. Oggetti Query

Se occorre una lista di tutte le istanze `Person` nel database, si può usare un oggetto Query. Per esempio:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"/>
```

E' possibile usarlo da una pagina JSF:

```
<h1
>List of people</h1>
<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable
>
```

Probabilmente occorre un supporto per la paginazione:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"
                        order="lastName"
                        max-results="20"/>
```

Si userà un parametro di pagina per determinare la pagina da mostrare:

```
<pages>
   <page view-id="/searchPerson.jsp">
      <param name="firstResult" value="#{people.firstResult}"/>
   </page>
</pages
>
```

Il codice JSF per il controllo della paginazione è un pò verboso, ma gestibile:

```
<h1
>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable>

<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="First Page">
   <f:param name="firstResult" value="0"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="Previous Page">
   <f:param name="firstResult" value="#{people.previousFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Next Page">
   <f:param name="firstResult" value="#{people.nextFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Last Page">
   <f:param name="firstResult" value="#{people.lastFirstResult}"/>
</s:link
>
```

Le schermate di ricerca consentono all'utente di inserire una serie di criteri di ricerca per restringere la lista dei risultati restituiti. L'oggetto Query consente di specificare delle "restrizioni" opzionali per supportare quest'importante caso d'uso:

```
<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
>
```

Si noti l'uso di un oggetto "esempio".

```
<h1
>Search for people</h1>
<h:form>
   <div
>First name: <h:inputText value="#{examplePerson.firstName}"/></div>
   <div
>Last name: <h:inputText value="#{examplePerson.lastName}"/></div>
   <div
><h:commandButton value="Search" action="/search.jsp"/></div>
</h:form>

<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable
>
```

Per fare il refresh della query qualora cambino le entità sottostanti, si può osservare l'evento `org.jboss.seam.afterTransactionSuccess`:

```
<event type="org.jboss.seam.afterTransactionSuccess">
   <action execute="#{people.refresh}" />
</event
>
```

O semplicemente per fare il refresh della query quando l'entity person viene persistita, aggiornata o rimossa attraverso `PersonHome`:

```
<event type="org.jboss.seam.afterTransactionSuccess.Person">
```

```
  <action execute="#{people.refresh}" />
  </event
>
```

Sfortunatamente gli oggetti Query non funzionano bene con query *join fetch* - non è consigliato l'uso della paginazione con queste query, ed occorrerà implementare un proprio metodo di calcolo del numero totale di risultati (con l'override di `getCountEjbql()`).

Gli esempi in questa sezione hanno mostrato tutti il riuso tramite configurazione. Comunque il riuso tramite estensione è ugualmente possibile per gli oggetti Query.

13.4. Oggetti controllori

Una parte totalmente opzionale del framework Seam è la classe `Controller` e le sue sottoclassi `EntityController`, `HibernateEntityController` e `BusinessProcessController`. Queste classi forniscono nient'altro che alcuni metodi di convenienza per l'accesso a componenti predefiniti comunemente usati e a metodi di componenti predefiniti. Essi aiutano a risparmiare alcuni colpi di tastiera ed a fornire un trampolino di lancio ai nuovi utenti per esplorare le ricche funzionalità definite in Seam.

Per esempio, qua è come appare `RegisterAction` dell'esempio Registrazione:

```
@Stateless
@Name("register")
public class RegisterAction extends EntityController implements Register
{

  @In private User user;

  public String register()
  {
    List existing = createQuery("select u.username from User u where u.username=:username")
      .setParameter("username", user.getUsername())
      .getResultList();

    if ( existing.size()==0 )
    {
      persist(user);
      info("Registered new user #{user.username}");
      return "/registered.jspx";
    }
    else
    {
      addFacesMessage("User #{user.username} already exists");
```

```
        return null;
      }
    }

}
```

Come si può vedere, non è un miglioramento sconvolgente...

Seam e JBoss Rules

Seam facilita le chiamate alle regole di JBoss Rules (Drools) dai componenti Seam o dalle definizioni di processo jBPM.

14.1. Installazione delle regole

Il primo passo è creare un'istanza di `org.drools.RuleBase` disponibile in una variabile del contesto di Seam. Per i test Seam fornisce un componente interno che compila un set statico di regole dal classpath. Si può installare questo componente tramite `components.xml`:

```
<drools:rule-base name="policyPricingRules">
   <drools:rule-files>
      <value
>policyPricingRules.drl</value>
   </drools:rule-files>
</drools:rule-base
>
```

Questo componente compila le regole da un set di file `.drl` e mette in cache un'istanza di `org.drools.RuleBase` nel contesto `APPLICATION` di Seam. Notare che è abbastanza probabile che in un'applicazione guidata dalle regole occorra installare altre basi di regole.

Se si vuole utilizzare una Drool DSL, devi specificare la definizione DSL:

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
   <drools:rule-files>
      <value
>policyPricingRules.drl</value>
   </drools:rule-files>
</drools:rule-base
>
```

Nella maggior parte delle applicazioni guidate dalle regole, le regole devono essere dinamicamente deployabili, e quindi un'applicazione in produzione dovrà usare un Drools RuleAgent per gestire la RuleBase. Il RuleAgent può connettersi al server di regole Drool (BRMS) od eseguire l'hot deploy dei pacchetti di regole dal repository locale. La RuleBase gestita dal RulesAgen è configurabile in `components.xml`:

```
<drools:rule-agent name="insuranceRules"
```

```
                configurationFile="/WEB-INF/deployedrules.properties" />
```

Il file delle proprietà contiene proprietà specifiche per RulesAgent. Ecco un file di configurazione d'esempio proveniente dalla distribuzione Drools.

```
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
```

E' anche possibile configurare le opzioni derettamente sul componente, bypassando il file di configurazione.

```
<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" />
```

Successivamente occorre rendere disponibile ad ogni conversazione un'istanza di `org.drools.WorkingMemory`. (Ogni `WorkingMemory` accumula fatti relativi alla conversazione corrente.)

```
<drools:managed-working-memory name="policyPricingWorkingMemory" auto-create="true"
 rule-base="#{policyPricingRules}"/>
```

Notare che è stato dato a `policyPricingWorkingMemory` un riferimento alla base di regole tramite la proprietà di configurazione `ruleBase`.

14.2. Utilizzo delle regole da un componente SEAM

Ora è possibile iniettare la `WorkingMemory` in un qualsiasi componente di Seam, asserire i fatti e lanciare le regole:

```
@In WorkingMemory policyPricingWorkingMemory;

@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException
{
   policyPricingWorkingMemory.assertObject(policy);
   policyPricingWorkingMemory.assertObject(customer);
   policyPricingWorkingMemory.fireAllRules();
}
```

14.3. Utilizzo delle regole da una definizione di processo jBPM

Si può anche consentire alla base di regole di agire come action handler di jBPM, decision handler, o assignment handler — sia in una definizione di pageflow sia in un processo di business.

```
<decision name="approval">

   <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
      <workingMemoryName
>orderApprovalRulesWorkingMemory</workingMemoryName>
      <assertObjects>
         <element
>#{customer}</element>
         <element
>#{order}</element>
         <element
>#{order.lineItems}</element>
      </assertObjects>
   </handler>

   <transition name="approved" to="ship">
      <action class="org.jboss.seam.drools.DroolsActionHandler">
         <workingMemoryName
>shippingRulesWorkingMemory</workingMemoryName>
         <assertObjects>
            <element
>#{customer}</element>
            <element
```

```
>#{order}</element>
            <element
>#{order.lineItems}</element>
         </assertObjects>
      </action>
   </transition>

   <transition name="rejected" to="cancelled"/>

</decision
>
```

L'elemento `<assertObjects>` specifica le espressioni EL che restituiscono un oggetto od una collezione di oggetti da asserire come fatti nella `WorkingMemory`.

Esiste anche il supporto per l'uso di Drools per le assegnazioni dei task in jBPM:

```
<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
>
```

Alcuni oggetti sono consultabili dalle regole come Drools globals, chiamate `Assignable` in jBPM, come `assignable` ed oggetto `Decision` in Seam, come `decision`. Le regole che gestiscono le decisioni dovrebbero chiamare `decision.setOutcome("result")` per determinare il risultato

della decisione. Le regole che eseguono assegnazioni dovrebbero impostare l'actor id usando `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
```

Nota

Si possono trovare altre informazioni su Drools all'indirizzo *http://www.drools.org*

Attenzione

Seam viene fornito con dipendenze Drools sufficienti per implementare alcune regole semplici. Per aggiungere ulteriori funzionalità a Drools occorre scaricare la distribuzione completa ed aggiungere le dipendenze necessarie.

Suggerimento

Drools viene rilasciato con MVEL compilato per Java 1.4, che è compatibile con Java 1.4, Java 5 e Java 6. E' possibile cambiare il jar MVEL con quello compilato per la propria versione di Java.

Sicurezza

15.1. Panoramica

Le API della sicurezza di Seam forniscono una serie di caratteristiche relative alla sicurezza di un'applicazione basata su Seam, coprendo le seguenti aree:

- Autenticazione - uno strato estensibile, basato su JAAS che consente all'utente di autenticarsi con qualsiasi fornitore di servizi di sicurezza.
- Gestione delle identità - una API per gestire a run time gli utenti e i ruoli di una applicazione Seam.
- Autorizzazione - un framework di autorizzazione estremamente comprensibile, che gestisce i ruoli degli utenti, i permessi persistenti oppure basati sulle regole e un risolutore di permessi modulare che consente di implementare facilmente una logica personalizzata per la gestione della sicurezza.
- Gestione dei permessi - un insieme di componenti Seam predefiniti che consente una gestione facile delle politiche di sicurezza dell'applicazione.
- Gestione dei CAPTCHA - per assistere nella prevenzione dagli attacchi automatici tramite software o script verso un sito basato su Seam.
- E molto altro

Queste capitolo si occuperà in dettaglio di ciascuna di queste caratteristiche.

15.2. Disabilitare la sicurezza

In determinate situazioni può essere necessario disabilitare la gestione della sicurezza in Seam, ad esempio durante i test oppure perché si sta usando un diverso approccio alla sicurezza, come l'uso diretto di JAAS. Per disabilitare l'infrastruttura della sicurezza chiamare semplicemente il metodo statico `Identity.setSecurityEnabled(false)`. Ovviamente non è molto pratico dover chiamare un metodo statico quando si vuole configurare un'applicazione, perciò in alternativa è possibile controllare questa impostazione in components.xml:

- Sicurezza delle entità
- Intercettore della sicurezza in Hibernate
- Intercettore della sicurezza in Seam
- Restrizioni sulle pagine
- Integrazione con la sicurezza delle API Servlet

Assumendo che si stia pianificando di sfruttare i vantaggi che la sicurezza Seam ha da offrire, il resto di questo capitolo documenta l'insieme delle opzioni disponibili per dare agli utenti un'identità

dal punto di vista del modello di sicurezza (autenticazione) e un accesso limitato all'applicazione secondo dei vincoli stabiliti (autorizzazione). Iniziamo con la questione dell'autenticazione poiché è il fondamento di ogni modello di sicurezza.

15.3. Autenticazione

Le caratteristiche relative all'autenticazione nella gestione della sicurezza di Seam sono costruite su JAAS (Java Authentication and Authorization Service, servizio di autenticazione e autorizzazione Java) e, come tali, forniscono una API robusta e altamente configurabile per gestire l'autentifica degli utenti. Comunque, per requisiti di autentifica meno complessi, Seam offre un metodo di autentifica molto semplificato che nasconde la complessità di JAAS.

15.3.1. Configurare un componente Authenticator

Nota

Nel caso si utilizzino le funzioni di gestione delle identità di Seam (discusse più avanti in questo capitolo) non è necessario creare un componente Authenticator (e si può saltare questo paragrafo).

Il metodo di autenticazione semplificato fornito da Seam usa un modulo di login JAAS già fatto, `SeamLoginModule`, il quale delega l'autentifica ad uno dei componenti dell'applicazione. Questo modulo di login è già configurato all'interno di Seam come parte dei criteri di gestione di default e in quanto tale non richiede alcun file di configurazione aggiuntivo. Esso consente di scrivere un metodo di autentifica usando le classi entità che sono fornite dall'applicazione o, in alternativa, di esegure l'autentifica con qualche altro fornitore di terze parti. Per configurare questa forma semplificata di autentifica è richiesto di di configurare il componente `Identity` in `components.xml`:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:core="http://jboss.com/products/seam/core"
        xmlns:security="http://jboss.com/products/seam/security"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
                "http://jboss.com/products/seam/components  http://jboss.com/products/seam/
components-2.1.xsd
                http://jboss.com/products/seam/security http://jboss.com/products/seam/security-
2.1.xsd">

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

</components
>
```

L'espressione EL `#{authenticator.authenticate}` è la definizione di un metodo tramite la quale si indica che il metodo `authenticate` del componente `authenticator` verrà usato per autenticare l'utente.

15.3.2. Scrivere un metodo di autenticazione

La proprietà `authenticate-method` specificata per `identity` in `components.xml` specifica quale metodo sarà usato dal `SeamLoginModule` per autenticare l'utente. Questo metodo non ha parametri ed è previsto che restituisca un boolean, il quale indica se l'autenticazione ha avuto successo o no. Il nome utente e la password possono essere ottenuti da `Credentials.getUsername()` e `Credentials.getPassword()` rispettivamente (è possibile avere un riferimento al componente `credentials` tramite `Identiy.instance().getCredentials()`). Tutti i ruoli di cui l'utente è membro devono essere assegnati usando `Identity.addRole()`. Ecco un esempio completo di un metodo di autentifica all'interno di un componente POJO:

```
@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;
    }

  }
```

```
}
```

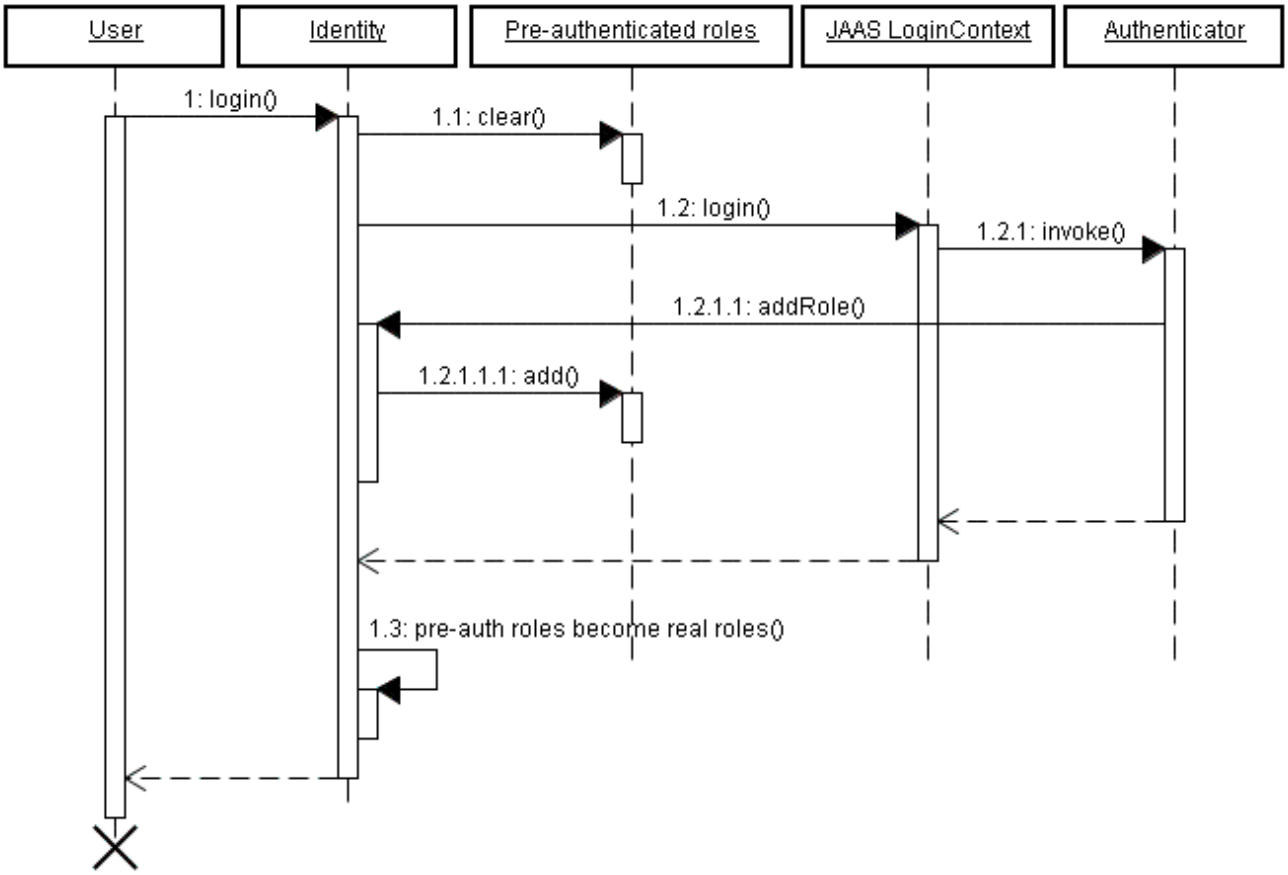
Nell'esempio precedente sia `User` che `UserRole` sono entity bean specifici dell'applicazione. Il parametro `roles` è popolato con i ruoli di cui l'utente è membro, che devono essere aggiunti alla `Set` come valori stringa, ad esempio "amministratore", "utente". In questo caso, se il record dell'utente non viene trovato e una `NoResultException` viene lanciata, il metodo di autenticazione restituisce `false` per indicare che l'autentifica è fallita.

Suggerimento

Nella scrittura di metodo di autenticazione è importante ridurlo al minimo e libero da ogni effetto collaterale. Il motivo è che non c'è garanzia sul numero di volte che il metodo di autentifica può essere chiamato dalle API della sicurezza, di conseguenza esso potrebbe essere invocato più volte durante una singola richiesta. Perciò qualsiasi codice che si vuole eseguire in seguito ad una autentifica fallita o completata con successo dovrebbe essere scritto implementando un observer. Vedi il paragrafo sugli Eventi di Sicurezza più avanti in questo capitolo per maggiori informazioni su quali eventi sono emessi dalla gestione della sicurezza Seam.

15.3.2.1. Identity.addRole()

Il metodo `Identity.addRole()` si comporta in modo diverso a seconda che la sessione corrente sia autenticata o meno. Se la sessione non è autenticata, allora `addRole()` dovrebbe essere chiamato *solo* durante il processo di autenticazione. Quando viene chiamato in questo contesto, il nome del ruolo è messo in una lista temporanea di ruoli pre autenticati. Una volta che l'autentifica è completata i ruoli pre autenticati diventano ruoli "reali" e chiamando `Identity.hasRole()` per questi ruoli si otterrà `true`. Il seguente diagramma di sequenza rappresenta la lista dei ruoli pre autenticati come oggetto in primo piano per mostrare più chiaramente come si inserisce nel processo di autentifica.



Se la sessione corrente è già autenticata, allora la chiamata `Identity.addRole()` avrà l'effetto atteso di concedere immediatamente il ruolo specificato all'utente corrente.

15.3.2.2. Scrivere un observer per gli eventi relativi alla sicurezza

Supponiamo, ad esempio, che in seguito ad un accesso concluso con successo debbano essere aggiornate certe statistiche relative all'utente. Questo può essere fatto scrivendo un observer per l'evento `org.jboss.seam.security.loginSuccessful`, come questo:

```
@In UserStats userStats;

@Observer("org.jboss.seam.security.loginSuccessful")
public void updateUserStats()
{
  userStats.setLastLoginDate(new Date());
  userStats.incrementLoginCount();
}
```

Questo metodo observer può essere messo ovunque, anche nello stesso componente Authenticator. E' possibile trovare maggiori informazioni sugli eventi relativi alla sicurezza più avanti in questo capitolo.

15.3.3. Scrivere una form di accesso

Il componente `credentials` fornisce sia la proprietà `username` che la `password`, soddisfacendo lo scenario di autenticazione più comune. Queste proprietà possono essere collegate direttamente ai campi username e password di una form di accesso. Una volta che queste proprietà sono impostate, chiamando `identity.login()` si otterrà l'autentifica dell'utente usando le credenziali fornite. Ecco un esempio di una semplice form di accesso:

```
<div>
   <h:outputLabel for="name" value="Nome utente"/>
   <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="Accedi" action="#{identity.login}"/>
</div
>
```

Allo stesso modo, l'uscita dell'utente viene fatta chiamando `#{identity.logout}`. La chiamata di questa azione cancellerà lo stato della sicurezza dell'utente correntemente autenticato e invaliderà la sessione dell'utente.

15.3.4. Riepilogo della configurazione

Riepilogando, ci sono tre semplici passi per configurare l'autenticazione:

- Configurare un metodo di autenticazione in `components.xml`.
- Scrivere un metodo di autenticazione.
- Scrivere una form di accesso così che l'utente possa autenticarsi.

15.3.5. Ricordami su questo computer

La sicurezza di Seam gestisce lo stesso tipo di funzionalità "Ricordami su questo computer" che si incontra comunemente in molte applicazioni basate sull'interfaccia web. In effetti essa è gestita in due diverse "varietà" o modalità. La prima modalità consente al nome utente di essere

memorizzato nel browser dell'utente come un cookie e lascia che sia il browser ad inserire la password (molti browser moderni sono in grado di ricordare le password).

La seconda modalità gestisce la memorizzazione di un identificativo unico in un cookie e consente all'utente di autenticarsi automaticamente non appena ritorna sul sito, senza dover fornire una password.

Avvertimento

L'autenticazione automatica tramite un cookie persistente memorizzato sulla macchina client è pericolosa. Benché sia conveniente per gli utenti, qualsiasi debolezza nella sicurezza che consenta un cross-site scripting nel sito avrebbe effetti drammaticamente più gravi del solito. Senza il cookie di autentifica, il solo cookie che un malintenzionato può prelevare tramite un attacco XSS è il cookie della sessione corrente dell'utente. Ciò significa che l'attacco funziona solo quando l'utente ha una sessione aperta, ovvero per un intervallo di tempo limitato. Al contrario è molto più allettante e pericoloso se un malintenzionato ha la possibilità di prelevare il cookie relativo alla funzione "Ricordami su questo computer", il quale gli consentirebbe di accedere senza autentifica ogni volta che vuole. Notare che questo dipende anche da quanto è efficace la protezione del sito dagli attacchi XSS. Sta a chi scrive l'applicazione fare in modo che il sito sia sicuro al 100% dagli attacchi XSS, un obiettivo non banale per qualsiasi sito che consente di rappresentare sulle pagine un contenuto scritto dagli utenti.

I produttori di browser hanno riconosciuto questo problema e hanno introdotto la funzione "Ricorda la password", oggi disponibile su quasi tutti i browser. In questo caso il browser ricorda il nome utente e la password per un certo sito e dominio, e riempie la form di accesso automaticamente quando non è attiva una sessione con il sito. Se poi il progettista del sito offre una scorciatoia da tastiera conveniente, questo approccio è quasi altrettanto immediato come il cookie "Ricordami su questo computer", ma molto più sicuro. Alcuni browser (ad esempio Safari su OS X) memorizzano addirittura i dati delle form di accesso nel portachiavi cifrato di sistema. Oppure, in un ambiente di rete, il portachiavi può essere trasportato dall'utente (tra il portatile e il desktop, ad esempio), mentre i cookie del browser di solito non sono sincronizzati.

In definitiva: benché tutti lo stiano facendo, il cookie "Ricordami su questo computer" con l'autenticazione automatica è un cattiva pratica e non dovrebbe essere usata. I cookie che "ricordano" solo il nome dell'utente e riempiono la form di accesso con quel nome utente per praticità, non comportano rischi.

Per abilitare la funzione "Ricordami su questo computer" nella modalità di default (quella sicura, con il solo nome utente) non è richiesta alcuna speciale configurazione. Basta collegare un checkbox "Ricordami su questo computer" a `rememberMe.enabled` nella form di accesso, come nel seguente esempio:

```
 <div>
   <h:outputLabel for="name" value="Nome utente"/>
   <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="Ricordami su questo computer"/>
   <h:selectBooleanCheckbox id="rememberMe" value="#{rememberMe.enabled}"/>
 </div
>
```

15.3.5.1. La modalità di autenticazione "Ricordami su questo computer" basata sul token

Per usare la modalità automatica, attraverso il token, della funzione "Ricordami su questo computer", occorre prima configurare la memorizzazione del token. Nello scenario più comune (gestito da Seam) questi token di autenticazione vengono memorizzati nel database, comunque è possibile implementare la propria memorizzazione dei token implementando l'interfaccia `org.jboss.seam.security.TokenStore`. In questo paragrafo si suppone che per la memorizzazione dei token in una tabella del database si stia usando l'implementazione fornita con Seam `JpaTokenStore`.

Il primo passo consiste nel creare una nuova entità che conterrà i token. Il seguente esempio mostra una possibile struttura che può essere usata:

```
@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;
  }
}
```

Come si può vedere dal listato, vengono usate un paio di annotazioni speciali, `@TokenUsername` e `@TokenValue`, per configurare le proprietà token e nome utente dell'entità. Queste annotazioni sono richieste per l'entità che conterrà i token di autenticazione.

Il passo successivo consiste nel configurare il `JpaTokenStore` per usare questo entity bean per memorizzare e recuperare i token di autenticazione. Ciò viene fatto in `components.xml` specificando l'attributo `token-class`.

```
<security:jpa-token-store token-class="org.jboss.seam.example.seamspace.AuthenticationToken"/>
```

Una volta fatto questo, l'ultima cosa da fare è configurare anche il componente `RememberMe` in `components.xml`. La sua proprietà `mode` dovrà essere impostata a `autoLogin`:

```
<security:remember-me mode="autoLogin"/>
```

Questo è tutto ciò che è necessario. L'autenticazione automatica degli utenti avverrà quando torneranno a visitare il sito (purché abbiano impostato il checkbox "Ricordami su questo computer").

Per essere sicuri che gli utenti siano autenticati automaticamente quando tornano sul sito, il seguente codice deve essere posizionato in `components.xml`:

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

15.3.6. Gestire le eccezioni della sicurezza

Per prevenire il fatto che gli utenti ricevano la pagina di errore di default in risposta ad un errore di sicurezza, si raccomanda che in `pages.xml` sia configurata una redirezione degli errori di sicurezza ad una pagina più "carina". I due principali tipi di eccezione lanciati dalle API della sicurezza sono:

- `NotLoggedInException` - Questa eccezione viene lanciata se l'utente tenta di accedere ad un'azione o ad una pagina protetta quando non ha fatto l'accesso.
- `AuthorizationException` - Questa eccezione viene lanciata solo se l'utente ha già fatto l'accesso e ha tentato di accedere ad un'azione o ad una pagina per la quale non ha i privilegi necessari.

Nel caso della `NotLoggedInException`, si raccomanda che l'utente venga rediretto o sulla pagina di accesso o su quella di registrazione, così che possa accedere. Per una `AuthorizationException`, può essere utile redirigere l'utente su una pagina di errore. Ecco un esempio di un `pages.xml` che redirige entrambe queste eccezioni:

```
<pages>

   ...

   <exception class="org.jboss.seam.security.NotLoggedInException">
      <redirect view-id="/login.xhtml">
         <message
>Per eseguire questa operazione devi prima eseguire l'accesso</message>
      </redirect>
```

```
   </exception>

   <exception class="org.jboss.seam.security.AuthorizationException">
      <end-conversation/>
      <redirect view-id="/security_error.xhtml">
         <message
>Non disponi dei privilegi di sicurezza necessari per eseguire questa operazione.</message>
      </redirect>
   </exception>

</pages
>
```

La maggior parte delle applicazioni web richiede una gestione più sofisticata della redirezione sulla pagina di accesso, perciò Seam include alcune funzionalità speciali per gestire questo problema:

15.3.7. Redirezione alla pagina di accesso

E' possibile chiedere a Seam di redirigere l'utente su una pagina di accesso quando un utente non autenticato tenta di accedere ad una particolare view (o ad una view il cui id corrisponda ad una wildcard), nel modo seguente:

```
<pages login-view-id="/login.xhtml">

   <page view-id="/members/*" login-required="true"/>

   ...

</pages
>
```

Suggerimento

Non è che una banale semplificazione rispetto alla gestione dell'eccezione illustrata prima, ma probabilmente dovrà essere usata insieme ad essa.

Dopo che l'utente ha eseguito l'accesso, lo si vorrà rimandare automaticamente indietro da dove è venuto, così che potrà riprovare ad eseguire l'azione che richiedeva l'accesso. Se si aggiungono i seguenti listener in `components.xml`, i tentativi di accesso ad una view protetta eseguiti quando non si è fatto l'accesso verranno ricordati così, dopo che l'utente ha eseguito l'accesso, può essere rediretto alla view che aveva originariamente richiesto, compresi tutti i parametri di pagina che esistevano nella richiesta originale.

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

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

Notare che la redirezione dopo l'accesso è implementata con un meccanismo con visibilità sulla conversazione, perciò occorre evitare di terminare la conversazione nel metodo `authenticate()`.

15.3.8. Autenticazione HTTP

Benché l'uso non sia raccomandato a meno che non sia assolutamente necessario, Seam fornisce gli strumenti per l'autenticazione in HTTP sia con metodo Basic che Digest (RFC 2617). Per usare entrambe le forme di autentifica, occorre abilitare il componente `authentication-filter` in `components.xml`:

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

Per abilitare il filtro per l'autenticazione Basic impostare `auth-type` a `basic`, oppure per l'autentifica Digest, impostarlo a `digest`. Se si usa l'autentifica Digest, occorre impostare anche un valore per `key` e `realm`:

```
<web:authentication-filter url-pattern="*.seam" auth-type="digest" key="AA3JK34aSDlkj"
realm="La mia Applicazione"/>
```

`key` può essere un qualunque valore stringa. `realm` è il nome del dominio di autenticazione che viene presentato all'utente quando si autentica.

15.3.8.1. Scrivere un autenticatore Digest

Se si usa l'autenticazione Digest, la classe authenticator deve estendere la classe astratta `org.jboss.seam.security.digest.DigestAuthenticator` e usare il metodo `validatePassword()` per validare la password in chiaro dell'utente con la richiesta Digest. Ecco un esempio:

```
public boolean authenticate()
{
  try
  {
    User user = (User) entityManager.createQuery(
      "from User where username = :username")
      .setParameter("username", identity.getUsername())
      .getSingleResult();

    return validatePassword(user.getPassword());
  }
  catch (NoResultException ex)
  {
    return false;
  }
}
```

15.3.9. Caratteristiche di autenticazione avanzate

Questo paragrafo esplora alcune delle caratteristiche avanzate fornite dalle API di sicurezza per affrontare requisiti di sicurezza più complessi.

15.3.9.1. Utilizzare la configurazione JAAS del container

Se non si vuole usare la configurazione JAAS semplificata fornita dalle API di sicurezza di Seam, è possibile delegare alla configurazione JAAS di default del sistema fornendo una proprietà `jaas-config-name` in `components.xml`. Ad esempio, se si sta usando JBoss AS e si vuole usare la politica `other` (la quale usa il modulo di login `UsersRolesLoginModule` fornito da JBoss AS), allora la voce da mettere in `components.xml` sarà simile a questa:

```
<security:identity jaas-config-name="other"/>
```

E' il caso di tenere ben presente che facendo in questo modo non significa che l'utente verrà autenticato in qualsiasi container in cui venga eseguita l'applicazione Seam. Questa configurazione istruisce semplicemente la sicurezza di Seam ad autenticarsi usando le politiche di sicurezza JAAS configurate.

15.4. Gestione delle identità

La gestione delle identità fornisce un'API standard per la gestione degli utenti e dei ruoli di una applicazione Seam, a prescindere da quale dispositivo di memorizzazione delle identità è usato internamente (database, LDAP, ecc). Al centro delle API per la gestione delle identità c'è il componente `identityManager`, il quale fornisce tutti i metodi per creare, modificare e cancellare utenti, concedere e revocare ruoli, cambiare le password, abilitare e disabilitare gli utenti, autenticare gli utenti ed elencare utenti e ruoli.

Prima di essere usato, `identityManager` deve essere configurato con uno o più `IdentityStore`. Questi componenti fanno il vero lavoro di interagire con il fornitore di sicurezza sottostante, sia che si tratti di un database, di un server LDAP o di qualcos'altro.

15.4.1. Configurare l'IdentityManager

Il componente `identityManager` consente di separare i dispositivi di memorizzazione configurati per le operazioni di autenticazione e di autorizzazione. Ciò significa che è possibile autenticare gli utenti tramite un dispositivo di memorizzazione, ad esempio una directory LDAP, e poi avere i loro ruoli caricati da un altro dispositivo di memorizzazione, come un database relazionale.

Seam fornisce due implementazioni `IdentityStore` già pronte. `JpaIdentityStore` usa un database relazionale per memorizzare le informazioni su utenti e ruoli ed è il dispositivo di memorizzazione di identità di default che viene usato se non viene configurato niente in modo esplicito nel componente `identityManager`. L'altra implementazione fornita è `LdapIdentityStore`, che usa una directory LDAP per memorizzare utenti e ruoli.

Ci sono due proprietà configurabili per il componente `identityManager`, `identityStore` e `roleIdentityStore`. Il valore di queste proprietà deve essere un'espressione EL che fa riferimento ad un componente Seam che implementa l'interfaccia `IdentityStore`. Come già detto, se viene lasciato non configurato allora `JpaIdentityStore` viene assunto come default. Se è configurata solamente la proprietà `identityStore` allora lo stesso valore verrà usato anche per `roleIdentityStore`. Ad esempio la seguente voce in `components.xml` configura `identityManager` per usare un `LdapIdentityStore` sia per le operazioni relative agli utenti che per quelle relative ai ruoli:

```
<security:identity-manager identity-store="#{ldapIdentityStore}"/>
```

Il seguente esempio configura `identityManager` per usare un `LdapIdentityStore` per le operazioni relative agli utenti e un `JpaIdentityStore` per le operazioni relative ai ruoli.

```
<security:identity-manager
  identity-store="#{ldapIdentityStore}"
  role-identity-store="#{jpaIdentityStore}"/>
```

Il paragrafo seguente spiega con maggiore dettaglio entrambe queste implementazioni di `IdentityStore`.

15.4.2. JpaIdentityStore

Questa memorizzazione delle identità consente agli utenti e ai ruoli di essere memorizzati in un database relazionale. E' progettato per essere il meno restrittivo possibile riguardo allo schema del database, consentendo una grande flessibilità per la struttura delle tabelle sottostanti. Questo si ottiene tramite l'uso di uno speciale insieme di annotazioni, consentendo agli entity bean di essere configurati per memorizzare utenti e ruoli.

15.4.2.1. Configurare JpaIdentityStore

`JpaIdentityStore` richiede che siano configurate sia la proprietà `user-class` che `role-class`. Queste proprietà devono riferirsi a classi entità che servono per memorizzare i record relativi agli utente e ai ruoli, rispettivamente. Il seguente esempio illustra la configurazione di `components.xml` nell'applicazione di esempio SeamSpace:

```
<security:jpa-identity-store
  user-class="org.jboss.seam.example.seamspace.MemberAccount"
  role-class="org.jboss.seam.example.seamspace.MemberRole"/>
```

15.4.2.2. Configurare le entità

Come già menzionato, un apposito insieme di annotazioni viene usato per configurare gli entity bean per la memorizzazione di utenti e ruoli. La seguente tabella elenca ciascuna di queste annotazioni e la relativa descrizione.

Tabella 15.1. Annotazioni per l'entità utente

Annotazione	Stato	Descrizione
`@UserPrincipal`	Richiesta	Questa annotazione contrassegna il campo o il metodo che contiene lo username dell'utente.

`@UserPassword`	Richiesta	Questa annotazione contrassegna il campo o il metodo che contiene la password dell'utente. Consente di specificare un algoritmo di `hash` per nascondere la password. I possibili valori per `hash` sono `md5`, `sha` e `none`. Ad esempio:
		`@UserPassword(hash = "md5")`
		`public String getPasswordHash() {`
		`return passwordHash;`
		`}`
		Se un'applicazione richiede un algoritmo di hash che non è supportato direttamente da Seam, è possibile estendere il componente `PasswordHash` per implementare un altro algoritmo.

`@UserFirstName`	Opzionale	Questa annotazione contrassegna il campo

Annotazione	Stato	Descrizione
		o il metodo contenente il nome dell'utente.
`@UserLastName`	Opzionale	Questa annotazione contrassegna il campo o il metodo contenente il cognome dell'utente.
`@UserEnabled`	Opzionale	Questa annotazione contrassegna il campo o il metodo contenente lo stato di abilitazione dell'utente. Questo deve essere una proprietà boolean e, se non presente, tutti gli utenti saranno considerati abilitati.
`@UserRoles`	Richiesta	Questa annotazione contrassegna il campo o il metodo contenente i ruoli dell'utente. Questa proprietà verrà descritta in maggiore dettaglio successivamente.

Tabella 15.2. Annotazioni per l'entità ruolo

Annotazione	Stato	Descrizione
`@RoleName`	Richiesta	Questa annotazione contrassegna il campo o il metodo contenente il nome del ruolo.
`@RoleGroups`	Opzionale	Questa annotazione contrassegna il campo o il metodo contenente i gruppi

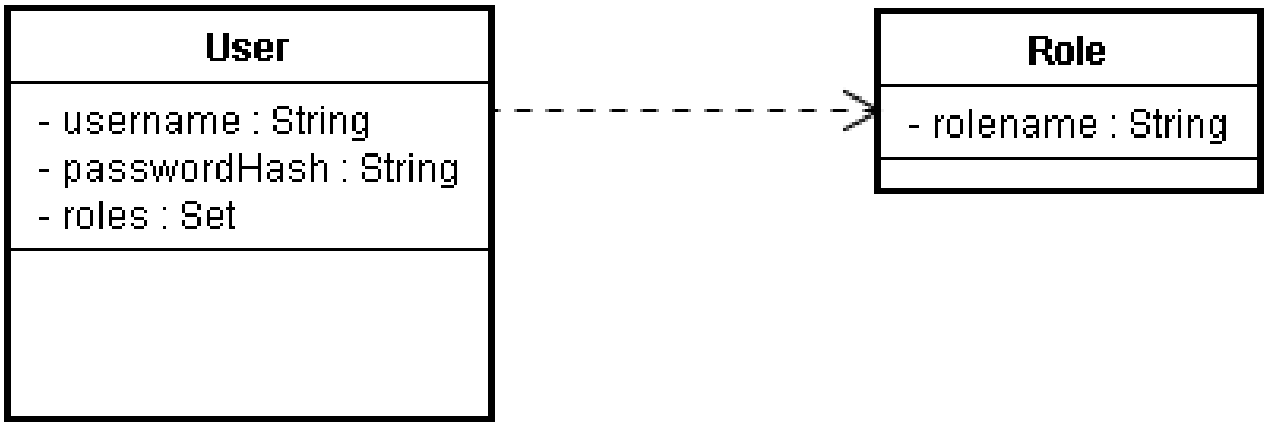
Annotazione	Stato	Descrizione
		di appartenenza del ruolo.
`@RoleConditional`	Opzionale	Questa annotazione contrassegna il campo o il metodo che indica se il ruolo è condizionale o no. I ruoli condizionali verranno spiegati più avanti in questo capitolo.

15.4.2.3. Esempi di entity bean

Come detto precedentemente, `JpaIdentityStore` è progettato per essere il più possibile flessibile per ciò che riguarda lo schema del database delle tabelle degli utenti e dei ruoli. Questo paragrafo esamina una serie di possibili schemi di database che possono essere usati per memorizzare i record degli utenti e dei ruoli.

15.4.2.3.1. Esempio di uno schema minimo

In questo esempio minimale una tabella di utenti e una di ruoli sono legate tramite una relazione molti-a-molti che utilizza una tabella di collegamento chiamata `UserRoles`.



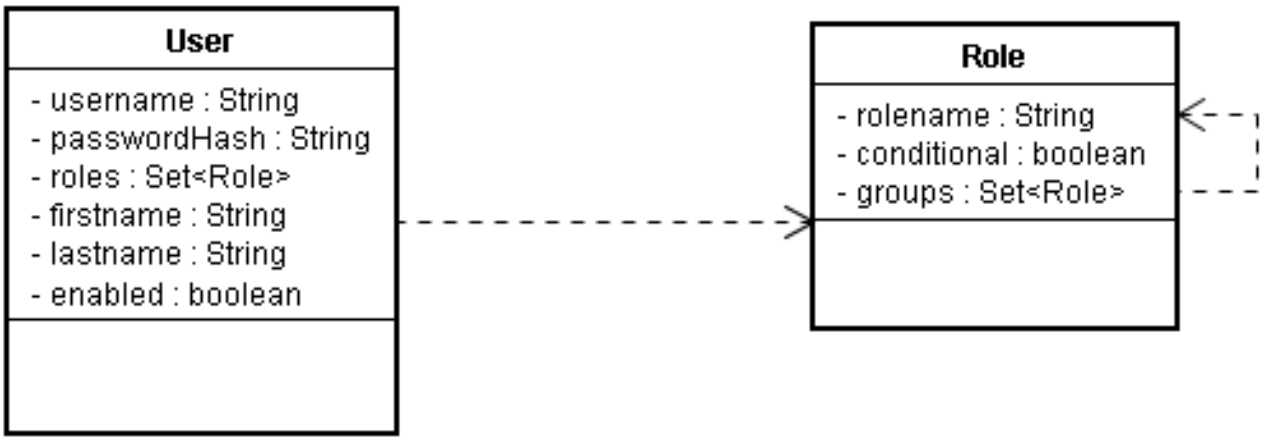

```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role
> roles;
```

```
  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) { this.passwordHash = passwordHash; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role
> getRoles() { return roles; }
  public void setRoles(Set<Role
> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }
}
```

15.4.2.3.2. Esempio di uno schema complesso

Questo esempio è costruito a partire dall'esempio minimo includendo tutti i campi opzionali e consentendo ai ruoli di appartenere ai gruppi.




```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role
> roles;
  private String firstname;
  private String lastname;
  private boolean enabled;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) { this.passwordHash = passwordHash; }

  @UserFirstName
  public String getFirstname() { return firstname; }
  public void setFirstname(String firstname) { this.firstname = firstname; }

  @UserLastName
  public String getLastname() { return lastname; }
  public void setLastname(String lastname) { this.lastname = lastname; }
```

```
  @UserEnabled
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) { this.enabled = enabled; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role
> getRoles() { return roles; }
  public void setRoles(Set<Role
> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;
  private boolean conditional;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }

  @RoleConditional
  public boolean isConditional() { return conditional; }
  public void setConditional(boolean conditional) { this.conditional = conditional; }

  @RoleGroups
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "RoleGroups",
    joinColumns = @JoinColumn(name = "RoleId"),
    inverseJoinColumns = @JoinColumn(name = "GroupId"))
  public Set<Role
> getGroups() { return groups; }
  public void setGroups(Set<Role
> groups) { this.groups = groups; }
```

```
}
```

15.4.2.4. Eventi del JpaIdentityStore

Quando si usa `JpaIdentityStore` come implementazione della memorizzazione delle identità con `IdentityManager`, alcuni eventi vengono lanciati in corrispondenza dell'invocazione di certi metodi di `IdentityManager`.

15.4.2.4.1. JpaIdentityStore.EVENT_PRE_PERSIST_USER

Questo evento viene lanciato in corrispondenza della chiamata `IdentityManager.createUser()`. Subito prima che l'entità utente sia resa persistente sul database questo evento viene lanciato passando l'istanza dell'entità come parametro dell'evento. L'entità sarà un istanza di `user-class` configurata per `JpaIdentityStore`.

Scrivere un metodo che osserva questo evento può essere utile per impostare valori addizionali sui campi dell'entità che non vengono impostati nell'ambito delle funzionalità standard di `createUser()`.

15.4.2.4.2. JpaIdentityStore.EVENT_USER_CREATED

Anche questo evento viene lanciato in corrispondenza di `IdentityMananger.createUser()`. Però viene lanciato dopo che l'entità utente è già stata resa persistente sul database. Come per l'evento `EVENT_PRE_PERSIST_USER`, anche questo passa l'istanza dell'entità come un parametro dell'evento. Può essere utile osservare questo evento se c'è bisogno di rendere persistenti altre entità che fanno riferimento all'entità utente, ad esempio informazioni di dettaglio del contatto o altri dati specifici dell'utente.

15.4.2.4.3. JpaIdentityStore.EVENT_USER_AUTHENTICATED

Questo evento viene lanciato quando viene chiamata `IdentityManager.authenticate()`. Passa l'istanza dell'entità utente come parametro dell'evento e risulta utile per leggere proprietà aggiuntive dall'entità dell'utente che è stato autenticato.

15.4.3. LdapIdentityStore

Questa implementazione della memorizzazione delle identità è progettata per funzionare quando le informazioni sugli utenti sono memorizzate in una directory LDAP. E' molto configurabile consentendo una grande flessibilità sul modo in cui utenti e ruoli sono memorizzati nella directory. Il seguente paragrafo descrive le opzioni di configurazione per questa implementazione e fornisce alcuni esempi di configurazione.

15.4.3.1. Configurare LdapIdentityStore

La seguente tabella descrive le proprietà disponibili che possono essere configurate in `components.xml` per `LdapIdentityStore`.

Tabella 15.3. Proprietà di configurazione di LdapIdentityStore

Proprietà	Valore di default	Descrizione
`server-address`	`localhost`	L'indirizzo del server LDAP
`server-port`	`389`	Il numero di porta su cui il server LDAP è in ascolto.
`user-context-DN`	`ou=Person,dc=acme,dc=com`	Il Distinguished Name (DN) del contesto contenente le informazioni sugli utenti.
`user-DN-prefix`	`uid=`	Questo valore è usato come prefisso anteponendolo al nome utente durante la ricerca delle informazioni sull'utente.
`user-DN-suffix`	`,ou=Person,dc=acme,dc=com`	Questo valore è aggiunto alla fine del nome utente per ricercare le informazioni sull'utente.
`role-context-DN`	`ou=Role,dc=acme,dc=com`	Il DN del contesto contenente le informazioni sui ruoli.
`role-DN-prefix`	`cn=`	Questo valore è usato come prefisso anteponendolo al nome del ruolo per formare il DN nella ricerca delle informazioni sul ruolo.
`role-DN-suffix`	`,ou=Roles,dc=acme,dc=com`	Questo valore è aggiunto al nome del ruolo per formare il DN nella ricerca delle informazioni sul ruolo.
`bind-DN`	`cn=Manager,dc=acme,dc=com`	

Proprietà	Valore di default	Descrizione
		Questo è il contesto usato per collegare il server LDAP.
`bind-credentials`	`secret`	Queste sono le credenziali (la password) usate per collegare il server LDAP.
`user-role-attribute`	`roles`	Questo è il nome dell'attributo sulle informazioni dell'utente che contiene la lista dei ruoli di cui l'utente è membro.
`role-attribute-is-DN`	`true`	Questa proprietà boolean indica se l'attributo del ruolo nelle informazioni dell'utente è esso stesso un Distinguished Name.
`user-name-attribute`	`uid`	Indica quale attributo delle informazioni sull'utente contiene il nome utente.
`user-password-attribute`	`userPassword`	Indica quale attributo nelle informazioni sull'utente contiene la password dell'utente.
`first-name-attribute`	`null`	Indica quale attributo nelle informazioni sull'utente contiene il nome proprio dell'utente.
`last-name-attribute`	`sn`	Indica quale attributo nelle informazioni sull'utente contiene il cognome dell'utente.

Proprietà	Valore di default	Descrizione
`full-name-attribute`	`cn`	Indica quale attributo nelle informazioni sull'utente contiene il nome per esteso dell'utente.
`enabled-attribute`	`null`	Indica quale attributo nelle informazioni sull'utente determina se l'utente è abilitato.
`role-name-attribute`	`cn`	Indica quale attributo nell'informazioni sul ruolo contiene il nome del ruolo.
`object-class-attribute`	`objectClass`	Indica quale attributo determina la classe di un oggetto nella directory.
`role-object-classes`	`organizationalRole`	Un elenco di classi di oggetto con cui devono essere create le informazioni su un nuovo ruolo.
`user-object-classes`	`person.uidObject`	Un elenco di classi di oggetto con cui devono essere create le informazioni su un nuovo utente.

15.4.3.2. Esempio di configurazione di LdapIdentityStore

La seguente configurazione di esempio mostra come `LdapIdentityStore` può essere configurato per una directory LDAP sul sistema immaginario `directory.mycompany.com`. Gli utenti sono memorizzati all'interno di questa directory sotto il contesto `ou=Person,dc=mycompany,dc=com` e sono identificati usando l'attributo `uid` (che corrisponde al loro nome utente). I ruoli sono memorizzati nel loro contesto, `ou=Roles,dc=mycompany,dc=com` e referenziati dalla voce dell'utente tramite l'attributo `roles`. Le voci dei ruoli sono identificate tramite il loro common name (l'attributo `cn`), che corrisponde al nome del ruolo. In questo esempio gli utenti possono essere disabilitati impostando il valore del loro attributo `enabled` a false.

```
<security:ldap-identity-store
```

```
server-address="directory.mycompany.com"
bind-DN="cn=Manager,dc=mycompany,dc=com"
bind-credentials="secret"
user-DN-prefix="uid="
user-DN-suffix=",ou=Person,dc=mycompany,dc=com"
role-DN-prefix="cn="
role-DN-suffix=",ou=Roles,dc=mycompany,dc=com"
user-context-DN="ou=Person,dc=mycompany,dc=com"
role-context-DN="ou=Roles,dc=mycompany,dc=com"
user-role-attribute="roles"
role-name-attribute="cn"
user-object-classes="person,uidObject"
enabled-attribute="enabled"
/>
```

15.4.4. Scrivere il proprio IdentityStore

Scrivere la propria implementazione della memorizzazione delle identità consente di autenticare ed eseguire le operazioni di gestione delle identità su fornitori di sicurezza che non sono gestiti da Seam così com'è. Per ottenere ciò è richiesta una sola classe ed essa deve implementare l'interfaccia `org.jboss.seam.security.management.IdentityStore`.

Fare riferimento al JavaDoc di `IdentityStore` per una descrizione dei metodi che devono essere implementati.

15.4.5. L'autenticazione con la gestione delle identità

Se in un'applicazione Seam si stanno usando le funzioni di gestione delle identità, allora non è richiesto di fornire un componente `authenticator` (vedi il precedente paragrafo Autenticazione) per abilitare l'autentifica. Basta omettere `authenticator-method` dalla configurazione di `identity` in `components.xml` e il `SeamLoginModule` userà per default `IdentityManger` per autenticare gli utenti dell'applicazione, senza nessuna configurazione speciale.

15.4.6. Usare IdentityManager

`IdentityManager` può essere utilizzato sia iniettandolo in un componente Seam come di seguito:

```
@In IdentityManager identityManager;
```

sia accedendo ad esso tramite il suo metodo statico `instance()`:

```
IdentityManager identityManager = IdentityManager.instance();
```

La seguente tabella descrive i metodi di API per `IdentityManager`:

Tabella 15.4. API per la gestione delle identità

Metodo	Valore restituito	Descrizione
`createUser(String name, String password)`	`boolean`	Crea un nuovo utente con il nome e la password specificate. Restituisce `true` se l'operazione si è conclusa con successo, altrimenti `false`.
`deleteUser(String name)`	`boolean`	Elimina le informazioni dell'utente con il nome specificato. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`createRole(String role)`	`boolean`	Crea un nuovo ruolo con il nome specificato. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`deleteRole(String name)`	`boolean`	Elimina il ruolo con il nome specificato. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`enableUser(String name)`	`boolean`	Abilita l'utente con il nome specificato. Gli

Metodo	Valore restituito	Descrizione
		utenti che non sono abilitati non sono in grado di autenticarsi. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`disableUser(String name)`	`boolean`	Disabilita l'utente con il nome specificato. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`changePassword(String name, String password)`	`boolean`	Modifica la password dell'utente con il nome specificato. Restituisce `true` se l'operazione si è conclusa con successo, oppure `false`.
`isUserEnabled(String name)`	`boolean`	Restituisce `true` se l'utente specificato è abilitato, oppure `false` se non lo è.
`grantRole(String name, String role)`	`boolean`	Concede il ruolo specificato all'utente o al ruolo. Il ruolo deve già esistere per essere concesso. Restituisce `true` se il ruolo è stato concesso,

Metodo	Valore restituito	Descrizione
		oppure `false` se era già stato concesso all'utente.
`revokeRole(String name, String role)`	`boolean`	Revoca il ruolo specificato all'utente o al ruolo. Restituisce `true` se l'utente specificato era membro del ruolo e questo è stato revocato con successo, oppure `false` se l'utente non è un membro del ruolo.
`userExists(String name)`	`boolean`	Restituisce `true` se l'utente specificato esiste, oppure `false` se non esiste.
`listUsers()`	`listUsers(String filter)`	Restituisce una lista di tutti i nomi utente in ordine alfanumerico.
`listUsers(String filter)`	`listUsers(String filter)`	Restituisce una lista di tutti i nomi utente filtrata secondo il parametro di filtro specificato e in ordine alfanumerico.
`listRoles()`	`listUsers(String filter)`	Restituisce una lista di tutti i nomi dei ruoli.
`getGrantedRoles(String name)`	`listUsers(String filter)`	Restituisce una lista dei nomi

Metodo	Valore restituito	Descrizione
		di tutti i ruoli esplicitamente concessi all'utente con il nome specificato.
`getImpliedRoles(String name)`	`List<String>`	Restituisce la lista dei nomi di tutti i ruoli implicitamente concessi all'utente specificato. I ruoli implicitamente concessi includono quelli che non sono concessi direttamente all'utente, ma sono concessi ai ruoli di cui l'utente è membro. Ad esempio, se il ruolo `admin` è un membro del ruolo `user` e un utente è membro del ruolo `admin`, allora i ruoli impliciti per l'utente sono sia `admin` che `user`.
`authenticate(String name, String password)`	`boolean`	Autenticazione il nome utente e la password specificati usando l'Identity Store configurato. Restituisce `true` se conclude con successo,

Metodo	Valore restituito	Descrizione
		oppure `false` se l'autentifica fallisce. Il successo dell'autenticazione non implica niente oltre al valore restituito dal metodo. Non cambia lo stato del componente `Identity`. Per eseguire un vero e proprio login deve essere invece usato il metodo `Identity.login()`.
`addRoleToGroup(String role, String group)`	`boolean`	Aggiunge il ruolo specificato come membro del gruppo specificato. Restituisce `true` se l'operazione va a buon fine.
`removeRoleFromGroup(String role, String group)`	`boolean`	Rimuove il ruolo specificato dal gruppo specificato. Restituisce `true` se l'operazione va a buon fine.
`listRoles()`	`listUsers(String filter)`	Elenca i nomi di tutti i ruoli.

L'uso delle API per la gestione delle identità richiede che l'utente chiamante abbia le autorizzazioni appropriate per invocare i suoi metodi. La seguente tabella descrive i permessi richiesti per ciascuno dei metodi in `IdentityManager`. Gli oggetti dei permessi elencati qui sotto sono valori stringa.

Tabella 15.5. Permessi di sicurezza nella gestione delle identità

Metodo	Oggetto del permesso	Azione del permesso
`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`
`revokeRole()`	`seam.user`	`update`
`userExists()`	`seam.user`	`read`
`listUsers()`	`seam.user`	`read`
`listRoles()`	`seam.role`	`read`
`addRoleToGroup()`	`seam.role`	`update`
`removeRoleFromGroup()`	`seam.role`	`update`

Il seguente listato fornisce un esempio con un insieme di regole di sicurezza che concedono al ruolo `admin` l'accesso a tutti i metodi relativi alla gestione delle identità:

```
rule ManageUsers
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.user", granted == false)
  Role(name == "admin")
then
  check.grant();
end

rule ManageRoles
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.role", granted == false)
  Role(name == "admin")
then
```

```
  check.grant();
end
```

15.5. Messaggi di errore

Le API di sicurezza producono una serie di messaggi di default per i diversi eventi relaivi alla sicurezza. La seguente tabella elenca le chiavi dei messaggi che possono essere usate per sovrascrivere questi messaggi specificandoli in un file `message.properties`. Per sopprimere un messaggio basta mettere nel file la chiave con un valore vuoto.

Tabella 15.6. Chiavi dei messaggi di sicurezza

Chiave del messaggio	Descrizione
`org.jboss.seam.loginSuccessful`	Questo messaggio viene prodotto quando un utente porta a buon fine un login tramite le API di sicurezza.
`org.jboss.seam.loginFailed`	Questo messaggio viene prodotto quando il processo di login fallisce, perché il nome utente e la password forniti dall'utente non sono corretti, oppure perché l'autenticazione è fallita per qualche altro motivo.
`org.jboss.seam.NotLoggedIn`	Questo messaggio viene prodotto quando un utente tenta di eseguire un'azione o di accedere ad una pagina che richiede un controllo di sicurezza e l'utente non è al momento autenticato.
`org.jboss.seam.AlreadyLoggedIn`	Questo messaggio viene prodotto quando un utente che è già autenticato tenta di eseguire di nuovo il login.

15.6. Autorizzazione

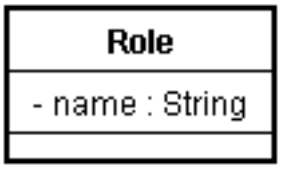
Ci sono diversi meccanismi di autorizzazione forniti dalle API di sicurezza di Seam per rendere sicuro l'accesso ai componenti, ai metodi dei componenti e alle pagine. Questo paragrafo descrive ognuno di essi. Un aspetto importante da notare è che qualora si voglia utilizzare una delle caratteristiche avanzate (come i permessi basati sulle regole) il `components.xml` potrebbe dover essere configurato per gestirle. Vedi il paragrafo Configurazione più sopra.

15.6.1. Concetti principali

La sicurezza di Seam è costruita intorno alla premessa per cui agli utenti vengono concessi ruoli e/o permessi, consentendo loro di eseguire operazioni che non sarebbero altrimenti permesse agli utenti senza i necessari privilegi di sicurezza. Ognuno dei meccanismi di autorizzazione forniti dalle API di sicurezza di Seam è costruito intorno a questo concetto principale di ruoli e permessi, con un framework espandibile che fornisce più modi per rendere sicure le risorse di un'applicazione.

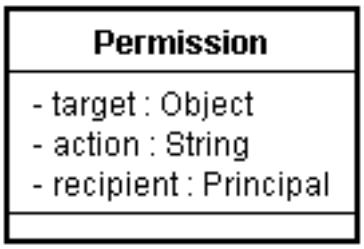
15.6.1.1. Cosa è un ruolo?

Un ruolo è un *gruppo*, o un *tipo*, di utente al quale possono essere concessi certi privilegi per eseguire una o più azioni specifiche nell'ambito dell'applicazione. Essi sono dei semplici costrutti consistenti solo di un nome che "amministratore", "utente", "cliente", ecc. Possono sia essere concessi ad un utente (o in alcuni casi ad altri ruoli) che essere usati per creare gruppi logici di utenti per facilitare l'assegnazione di determinati privilegi dell'applicazione.



15.6.1.2. Cosa è un permesso?

Un permesso è un privilegio (a volte una-tantum) per eseguire una singola, specifica azione. E' del tutto possibile costruire un'applicazione usando nient'altro che i privilegi, comunque i ruoli offrono un livello di facilitazione più alto quando si tratta di concedere dei privilegi a gruppi di utenti. Essi sono leggermente più complessi nella struttura rispetto ai ruoli ed essenzialmente consistono di tre "aspetti": un obiettivo , un'azione e un destinatario. L'obiettivo di un permesso è l'oggetto (o un nome arbitrario o una classe) per il quale è consentito di eseguire una determinata azione da parte di uno specifico destinatario (o utente). Ad esempio, l'utente "Roberto" può avere il permesso di cancellare gli oggetti cliente. In questo caso l'obiettivo del permesso può essere "clienti", l'azione del permesso sarà "cancella" e il recipiente sarà "Roberto".



Nell'ambito di questa documentazione i permessi sono generalmente rappresentati nella forma `obiettivo:azione` (omettendo il destinatario, benché nella realtà sarà sempre richiesto).

15.6.2. Rendere sicuri i componenti

Iniziamo ad esaminare la forma più semplice di autorizzazione, la sicurezza dei componenti, inziando con l'annotazione `@Restrict`.

@Restrict e le annotazioni di sicurezza tipizzate

Benché l'uso dell'annotazione `@Restrict` fornisca un metodo flessibile e potente per rendere sicuri i componenti grazie alla sua possibilità di gestire le espressione

EL, è consigliabile usare l'equivalente tipizzato (descritto più avanti), se non altro per la sicurezza a livello di compilazione che fornisce.

15.6.2.1. L'annotazione @Restrict

I componenti Seam possono essere resi sicuri sia a livello di metodo che a livello di classe usando l'annotazione `@Restrict`. Se sia un metodo che la classe in cui è dichiarato sono annotati con `@Restrict`, la restrizione sul metodo ha la precedenza (e la restrizione sulla classe non si applica). Se nell'invocazione di un metodo fallisce il controllo di sicurezza, viene lanciata un'eccezione come definito nel contratto di `Identity.checkRestriction()` (vedi Restrizioni in linea). Una `@Restrict` solo sulla classe del componente stesso è equivalente ad aggiungere `@Restrict` a ciascuno dei suoi metodi.

Una `@Restrict` vuota implica un controllo di permesso per `nomeComponente:nomeMetodo`. Prendiamo ad esempio il seguente metodo di un componente:

```
@Name("account")
public class AccountAction {
   @Restrict public void delete() {
     ...
   }
}
```

In questo esempio il permesso richiesto per chiamare il metodo `delete()` è `account:delete`. L'equivalente di ciò sarebbe stato scrivere `@Restrict("#{s:hasPermission('account','delete')}")`. Ora vediamo un altro esempio:

```
@Restrict @Name("account")
public class AccountAction {
   public void insert() {
     ...
   }
   @Restrict("#{s:hasRole('admin')}")
   public void delete() {
     ...
   }
}
```

Questa volta la classe stessa del componente è annotata con `@Restrict`. Ciò significa che tutti i metodi senza una annotazione `@Restrict` a sovrascrivere, richiedono un controllo implicito di permesso. Nel caso di questo esempio il metodo `insert()` richiede un permesso per `account:insert`, mentre il metodo `delete()` richiede che l'utente sia membro del ruolo `admin`.

Prima di andare avanti, esaminiamo l'espressione `#{s:hasRole()}` vista nell'esempio precedente. Sia `s:hasRole()` che `s:hasPermission` sono funzioni EL, le quali delegano ai metodi con i nomi corrispondenti nella classe `Identity`. Queste funzioni possono essere usate all'interno di una espressione EL in tutte le API di sicurezza.

Essendo un'espressione EL, il valore dell'annotazione `@Restrict` può fare riferimento a qualunque oggetto che sia presente in un contesto Seam. Ciò è estremamente utile quando si eseguono i controlli sui permessi per una specifica istanza di un oggetto. Ad esempio:

```
@Name("account")
public class AccountAction {
   @In Account selectedAccount;
   @Restrict("#{s:hasPermission(selectedAccount,'modifica')}")
   public void modify() {
      selectedAccount.modify();
   }
}
```

La cosa interessante da notare in questo esempio è il riferimento a `selectedAccount` che si vede all'interno della chiamata alla funzione `hasPermission`. Il valore di questa variabile verrà ricercato all'interno del contesto Seam e passato al metodo `hasPermission()` di `Identity`, il quale in questo caso può determinare se l'utente ha il permesso richiesto per modificare l'oggetto `Account` specificato.

15.6.2.2. Restrizioni in linea

A volte può risultare desiderabile eseguire un controllo di sicurezza nel codice, senza usare l'annotazione `@Restrict`. In questa situazione basta usare semplicemente `Identity.checkRestriction()` per risolvere l'espressione di sicurezza, così:

```
public void deleteCustomer() {
   Identity.instance().checkRestriction("#{s:hasPermission(selectedCustomer,'delete')}");
}
```

Se l'espressione specificata non risolve a `true`, allora

- se l'utente non ha eseguito l'accesso, l'eccezione `NotLoggedInException` viene lanciata, oppure
- se l'utente ha eseguito l'accesso, viene lanciata un'eccezione `AuthorizationException`.

E' anche possibile chiamare i metodi `hasRole()` e `hasPermission()` direttamente dal codice Java:

```
if (!Identity.instance().hasRole("amministratore"))
      throw new AuthorizationException("Devi essere un amministratore per eseguire questa
 azione");

if (!Identity.instance().hasPermission("cliente", "crea"))
    throw new AuthorizationException("Non puoi creare nuovi clienti");
```

15.6.3. La sicurezza nell'interfaccia utente

Una degli indici di un'interfaccia utente ben progettata è quando agli utenti non vengono presentate opzioni per le quali non hanno i permessi necessari per usarle. La sicurezza di Seam consente la visualizzazione condizionale sia di sezioni di una pagina che di singoli controlli, basata sui privilegi dell'utente, usando esattamente le stesse espressioni EL che sono usate nella sicurezza dei componenti.

Diamo un'occhiata ad alcuni esempi della sicurezza nell'interfaccia. Prima di tutto prentendiamo di avere una form di accesso che debba essere visualizzata solo se l'utente non ha già fatto l'accesso. Usando la proprietà `identity.isLoggedIn()` possiamo scrivere questo:

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}"
>
```

Se l'utente non ha eseguito l'accesso, allora la form di accesso verrà visualizzata. Fin qui tutto bene. Ora vogliamo che ci sia un menu sulla pagina che contenga alcune azioni speciali che devono essere accessibili solo agli utenti del ruolo `dirigente`. Ecco un modo in cui ciò potrebbe essere scritto:

```
<h:outputLink action="#{reports.listManagerReports}" rendered="#{s:hasRole('dirigente')}">
   Rapporti per i dirigenti
</h:outputLink
>
```

Anche fin qui tutto bene. Se l'utente non è un membro del ruolo `dirigente`, allora `outputLink` non verrà visualizzato. L'attributo `rendered` in generale può essere usato per il controllo stesso oppure in un controllo `<s:div>` o `<s:span>` che ne comprende altri.

Ora andiamo su qualcosa di più complesso. Supponiamo di avere in una pagina un controllo `h:dataTable` che elenca delle righe per le quali si può volere visualizzare o meno i link alle azioni in funzione dei permessi dell'utente. La funzione EL `s:hasPermission` ci consente di passare un parametro oggetto che può essere usato per determinare se l'utente ha o meno il permesso

richiesto per quell'oggetto. Ecco come può apparire una `dataTable` con dei link controllati dalla sicurezza:

```
<h:dataTable value="#{clients}" var="cl">
   <h:column>
      <f:facet name="header"
>Nome</f:facet>
      #{cl.name}
   </h:column>
   <h:column>
      <f:facet name="header"
>Città</f:facet>
      #{cl.city}
   </h:column>
   <h:column>
      <f:facet name="header"
>Azione</f:facet>
      <s:link value="Modifica cliente" action="#{clientAction.modify}"
            rendered="#{s:hasPermission(cl,'modifica')}"/>
      <s:link value="Cancella cliente" action="#{clientAction.delete}"
            rendered="#{s:hasPermission(cl,'cancella')}"/>
   </h:column>
</h:dataTable
>
```

15.6.4. Rendere sicure le pagine

La sicurezza delle pagine richiede che l'applicazione usi un file `pages.xml`. Comunque è molto semplice da configurare. Basta includere un elemento `>restrict<` all'interno degli elementi `page` che si vogliono rendere sicuri. Se tramite l'elemento `restrict` non viene indicata esplicitamente una restrizione, verrà controllato implicitamente il permesso `/viewId.xhtml:render` quando la richiesta della pagina avviene in modo non-faces (GET), e il permesso`/viewId.xhtml:restore` quando un JSF postback (il submit della form) viene originato dalla pagina. Altrimenti viene la restrizione specificata verrà valutata come una normale espressione di sicurezza. Ecco un paio di esempi:

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

Questa pagina richiede implicitamente un permesso `/settings.xhtml:render` per le richieste non-faces e un permesso `/settings.xhtml:restore` per le richieste faces.

```
<page view-id="/reports.xhtml">
   <restrict
>#{s:hasRole('amministratore')}</restrict>
</page
>
```

Sia le richieste faces che quelle non-faces a questa pagina richiedono che l'utente sia membro del ruolo `amministratore`.

15.6.5. Rendere sicure le entità

La sicurezza di Seam consente anche di applicare le restrizioni di sicurezza alle azioni per leggere, inserire, aggiornare e cancellare le entità.

Per rendere sicure tutte le azioni per una classe entità, aggiungere un'annotazione `@Restrict` alla classe stessa:

```
@Entity
@Name("customer")
@Restrict
public class Customer {
  ...
}
```

Se nell'annotazione `@Restrict` non è indicata alcuna espressione, il controllo di sicurezza di default che viene eseguito è una verifica del permesso `entità:azione`, dove l'obiettivo del permesso è l'istanza dell'entità e `azione` è `read`, `insert`, `update` o `delete`.

E' anche possibile applicare una restrizione solo a determinate azioni, posizionando l'annotazione `@Restrict` nel corrispondente metodo relativo al ciclo di vita dell'entità (annotato come segue):

- `@PostLoad` - Chiamato dopo che l'istana di una entità viene caricata dal database. Usare questo metodo per configurare un permesso `read`.
- `@PrePersist` - Chiamato prima che una nuova istanza dell'entità sia inserita. Usare questo metodo per configurare un permesso `insert`.
- `@PreUpdate` - Chiamato prima che un'entità sia aggiornata. Usare questo metodo per configurare un permesso `update`.

- `@PreRemove` - Chiamato prima che un'entità venga cancellata. Usare questo metodo per configuare un permesso `delete`.

Ecco un esempio di come un'entità potrebbe essere configurata per eseguire un controllo di sicurezza per tutte le operazioni `insert`. Notare che non è richiesto che il metodo faccia qualcosa, la sola cosa importante per quanto riguarda la sicurezza è come è annotato:

```
@PrePersist @Restrict
public void prePersist() {}
```

Usare `/META-INF/orm.xml`

E' anche possibile specificare i metodi callback 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
>
```

Ovviamente c'è sempre bisogno di annotare il metodo `prePersist()` in `Customer` con `@Restrict`.

Ed ecco un esempio di una regola sui permessi di entità che controlla se all'utente autenticato è consentito di inserire un record `MemberBlog` (dall'applicazione di esempio seamspace). L'entità per la quale viene fatto il controllo di sicurezza è inserita automaticamente nella working memory (in questo caso `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;
```

Questa regola concederà il permesso `memberBlog:insert` se l'utente attualmente autenticato (indicato dal fatto `Principal`) ha lo stesso nome del membro per il quale è stata creata la voce del blog. La riga "`principal:  Principal()`" può essere vista nel codice di esempio come un collegamento con una variabile. Essa collega l'istanza dell'oggetto `Principal` nella working memory (posizionato durante l'autenticazione) e lo assegna ad una variabile chiamata `principal`. I collegamenti con le variabili consentono di fare riferimento al valore in altri posti, come nella riga successiva che confronta il nome dell'utente con il nome del `Principal`. Per maggiori dettagli fare riferimento alla documentazione di JBoss Rules.

Infine abbiamo bisogno di installare una classe listener che integra la sicurezza Seam con la libreria JPA.

15.6.5.1. La sicurezza delle entità con JPA

I controlli di sicurezza sugli entity bean EJB3 sono eseguiti con un `EntityListener`. E' possibile installare questo listener usando il seguente file `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">

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener class="org.jboss.seam.security.EntitySecurityListener"/>
         </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>

</entity-mappings
```

```
>
```

15.6.5.2. Sicurezza delle entità con una sessione Hibernate gestita

Se si sta usando un `SessionFactory` di Hibernate configurato tramite Seam e si stanno usando le annotazioni oppure `orm.xml`, allora non c'è bisogno di fare niente di particolare per usare la sicurezza sulle entità.

15.6.6. Annotazioni tipizzate per i permessi

Seam fornisce una serie di annotazioni che possono essere usate come un'alternativa a `@Restrict` e che hanno l'ulteriore vantaggio di essere verificabili durante la compilazione, dato che non gestiscono espressioni EL arbitrarie nel modo in cui succede per la `@Restrict`.

Così com'è, Seam contiene delle annotazioni per i permessi standard per le operazioni CRUD, comunque è solo questione di aggiungerne altre. Le seguenti annotazioni sono fornire nel package `org.jboss.seam.annotations.security`:

- @Insert
- @Read
- @Update
- @Delete

Per usare queste annotazioni basta metterle sul metodo o sul parametro per il quale si vuole eseguire il controllo di sicurezza. Se messe su un metodo, allora dovranno specificare la classe obiettivo per la quale il permesso deve essere controllato. Si prenda il seguente esempio:

```
@Insert(Customer.class)
public void createCustomer() {
  ...
}
```

In questo esempio un controllo di permessi viene fatto sull'utente per assicurarsi che abbia i diritti per creare un nuovo oggetto `Customer`. L'obiettivo del controllo di permessi sarà `Customer.class` (l'effettiva istanza di `java.lang.Class`) e l'azione è la rappresentazione a lettere minuscole del nome dell'annotazione, che in questo esempio è `insert`.

E' anche possibile annotare i parametri di un metodo di un componente allo stesso modo. Se viene fatto in questo modo non è richiesto di specificare l'obiettivo del permesso (dato che il valore stesso del parametro sarà l'obiettivo del controllo di permessi):

```
public void updateCustomer(@Update Customer customer) {
```

```
  ...
 }
```

Per creare una propria annotazione di sicurezza basta annotarla con `@PermissionCheck`, ad esempio:

```
@Target({METHOD, PARAMETER})
@Documented
@Retention(RUNTIME)
@Inherited
@PermissionCheck
public @interface Promote {
  Class value() default void.class;
}
```

Se si vuole modificare il nome dell'azione di default del permesso (che è la versione a lettere minuscole del nome dell'annotazione) con un altro valore, è possibile specificarlo all'interno dell'annotazione `@PermissionCheck`:

```
@PermissionCheck("upgrade")
```

15.6.7. Annotazioni tipizzate per i ruoli

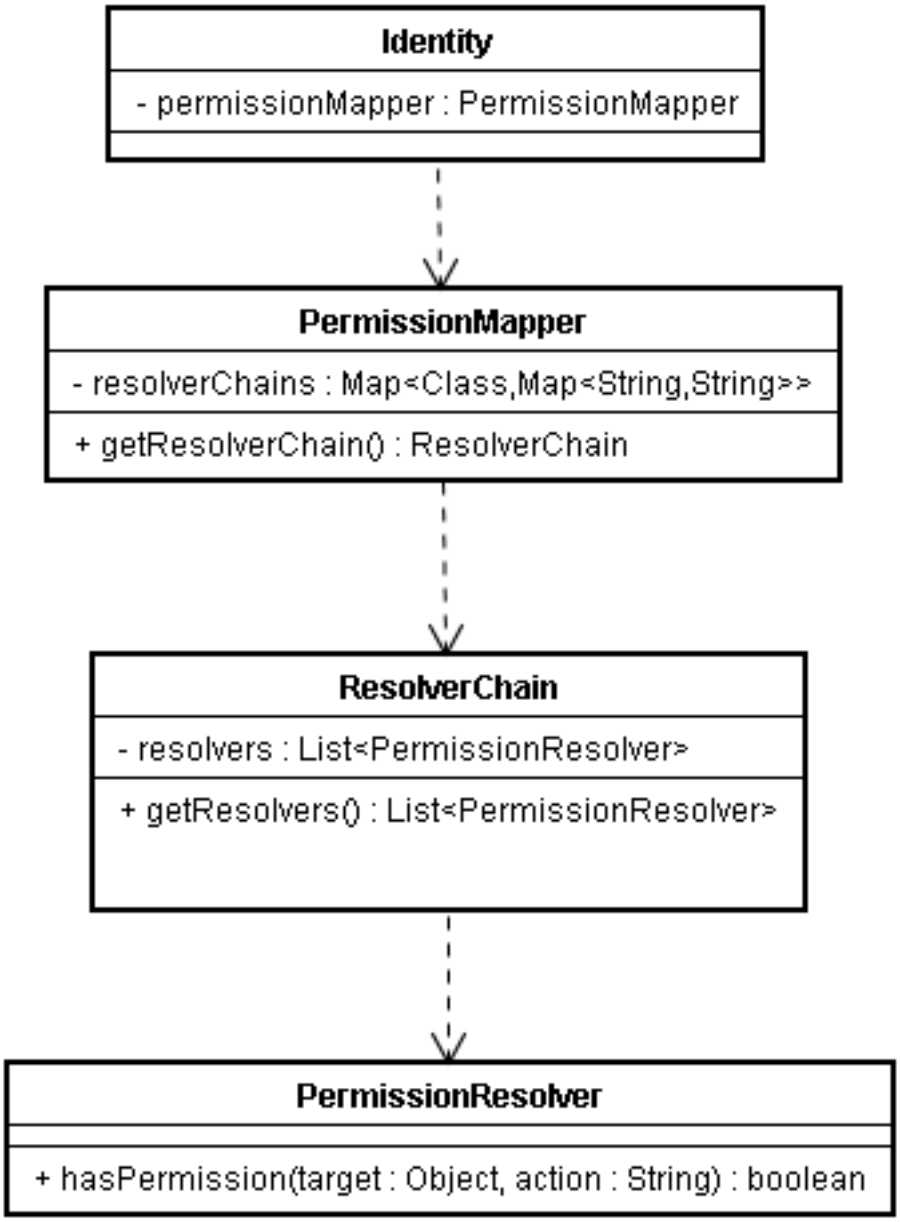
In aggiunta alla gestione tipizzata delle annotazioni sui permessi, la sicurezza di Seam fornisce anche le annotazioni tipizzate per i ruoli che consentono di limitare l'accesso ai metodi dei componenti in base all'appartenenza ad un ruolo dell'utente attualmente autenticato. Seam fornisce una di queste annotazioni già fatta, `org.jboss.seam.annotations.security.Admin`, usata per limitare l'accesso ad un metodo agli utenti che sono membri del ruolo `admin` (purché l'applicazione gestisca un tale ruolo). Per creare le proprie annotazioni per i ruoli basta meta-annotarle con `org.jboss.seam.annotations.security.RoleCheck`, come nel seguente esempio:

```
@Target({METHOD})
@Documented
@Retention(RUNTIME)
@Inherited
@RoleCheck
public @interface User {
}
```

Qualsiasi metodo successivamente annotato con l'annotazione `@User` come mostrata nell'esempio precedente, sarà automaticamente intercettato e sarà verificata l'appartenenza dell'utente al ruolo con il nome corrispondente (che è la versione a lettere minuscole del nome dell'annotazione, in questo caso `user`).

15.6.8. Il modello di autorizzazione dei permessi

La sicurezza di Seam fornisce un framework espandibile per risolvere i permessi dell'applicazione. Il seguente diagramma di classi mostra una panoramica dei componenti principali del framework dei permessi:



Le classi rilevanti sono spiegate in maggiore dettaglio nel seguente paragrafo.

15.6.8.1. PermissionResolver

Questa è in realtà un'interfaccia che fornisce i metodi per risolvere i singoli permessi sugli oggetti. Seam fornisce le seguenti implementazioni già fatte di `PermissionResolver`, che sono descritte in maggiore dettaglio più avanti in questo capitolo:

- `RuleBasedPermissionResolver` - Questo risolutore di permessi usa Drools per risolvere i controlli di permesso basati sulle regole.
- `PersistentPermissionResolver` - Questo risolutore di permessi memorizza gli oggetti permesso in un dispositivo persistente, come un database relazionale.

15.6.8.1.1. Scrivere il proprio PermissionResolver

E' molto semplice implementare il proprio risolutore di permessi. L'interfaccia `PermissionResolver` definisce solo due metodi che devono essere implementati, come mostra la seguente tabella. Includendo la propria implementazione di `PermissionResolver` nel proprio progetto Seam, essa sarà automaticamente rilevata durante l'esecuzione e registrata nel `ResolverChain` predefinito.

Tabella 15.7. L'interfaccia PermissionResolver

Tipo restituito	Metodo	Descrizione
`boolean`	`hasPermission(Object target, String action)`	Questo metodo deve stabilire se l'utente attualmente autenticato (ottenuto tramite una chiamata a `Identity.getPrincipal()`) ha il permesso specificato dai parametri `target` e `action`. Deve restituire `true` se l'utente ha il permesso, oppure `false` se non ce l'ha.
`void`		

Tipo restituito	Metodo	Descrizione
	`filterSetByAction(Set<Object> targets, String action)`	Questo metodo deve rimuovere dall'insieme specificato tutti gli oggetti per i quali si otterrebbe `true` se venissero passati al metodo `hasPermission()` con lo stesso valore del parametro `action`.

Nota

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.

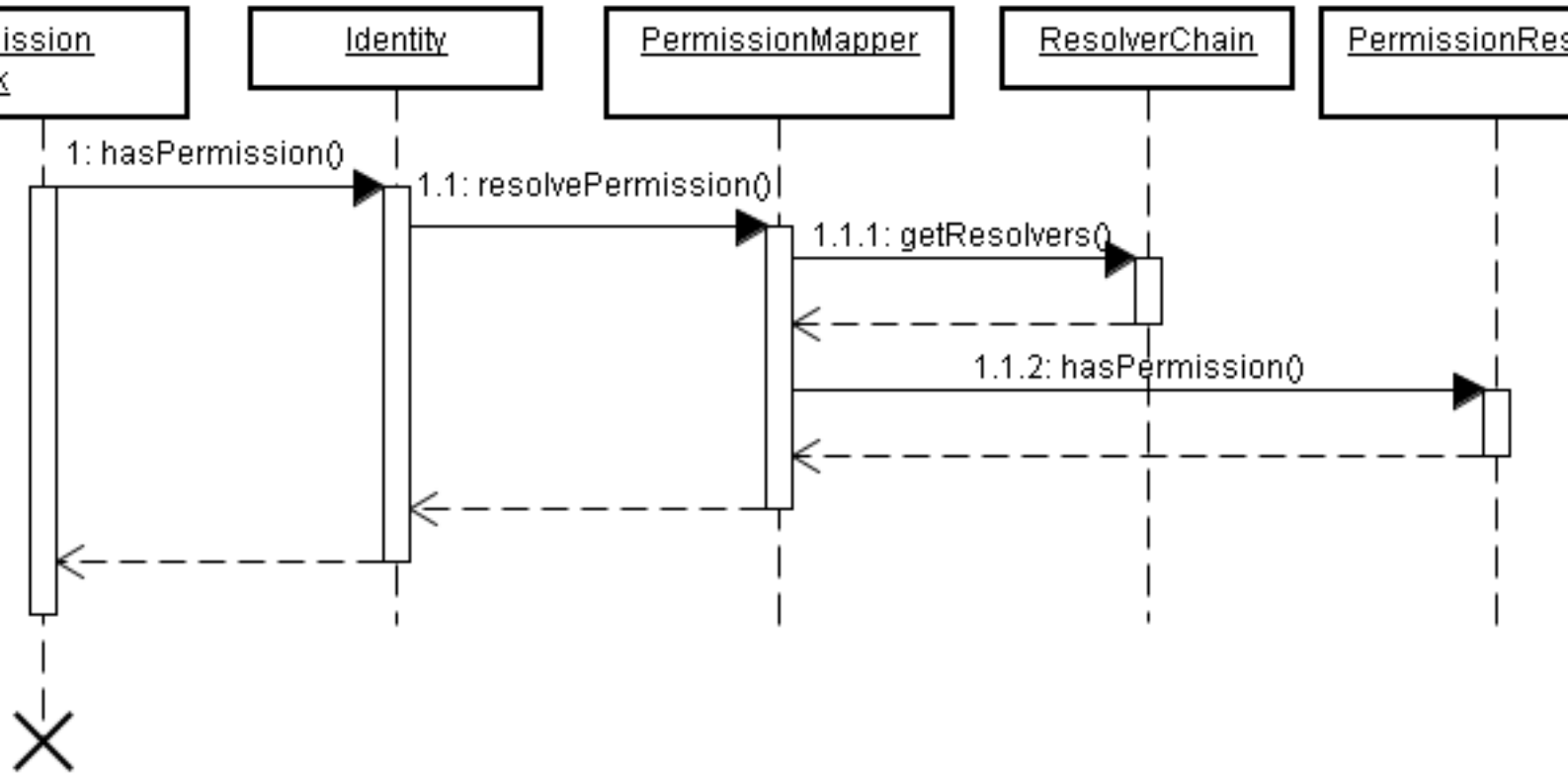
15.6.8.2. ResolverChain

Un `ResolverChain` contiene un elenco ordinato di `PermissionResolver`, con lo scopo di risolvere i permessi sugli oggetti di una determinata classe oppure i permessi obiettivo.

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.

Il seguente diagramma di sequenza mostra l'interazione tra i componenti del framework dei permessi durante la verifica di un permesso (segue la spiegazione). Una verifica di permesso può

essere originata da una serie di possibili fonti, ad esempio gli intercettori di sicurezza, la funzione EL `s:hasPermission`, oppure tramite una chiamata alla API `Identity.checkPermission`:



- 1. Una verifica di permesso viene iniziata da qualche parte (dal codice o tramite un'espressione EL) provocando una chiamata a `Identity.hasPermission()`.
- 1.1. `Identity` chiama `PermissionMapper.resolvePermission()`, passando il permesso che deve essere risolto.
- 1.1.1. `PermissionMapper` conserva una `Map` di istanze di `ResolverChain`, indicizzate per classe. Usa questa mappa per identificare la giusta `ResolverChain` per l'oggetto obiettivo del permesso. Una volta che ha la giusta `ResolverChain`, recupera l'elenco dei `PermissionResolver` che contiene tramite una chiamata a `ResolverChain.getResolvers()`.
- 1.1.2. Per ciascun `PermissionResolver` nel `ResolverChain`, il `PermissionMapper` chiama il suo metodo `hasPermission()`, passando l'istanza del permesso da verificare. Se qualcuno dei `PermissionResolver` restituisce `true`, allora la verifica del permesso ha avuto successo e il `PermissionMapper` restituisce anch'esso `true` a `Identity`. Se nessuno dei `PermissionResolver` restituisce `true`, allora la verifica del permesso è fallita.

15.6.9. RuleBasedPermissionResolver

Uno dei risolutori di permesso già fatti forniti da Seam, `RuleBasedPermissionResolver`, consente di valutare i permessi in base ad un insieme di regole di sicurezza Drools (JBoss Rules). Un paio di vantaggi nell'uso di un motore di regole sono: 1) una posizione centralizzata della logica di

gestione che è usata per valutare i permessi degli utenti; 2) la velocità, Drools usa algoritmi molto efficienti per valutare grandi quantità di regole complesse comprendenti condizioni multiple.

15.6.9.1. Requisiti

Se si usa la funzione dei permessi basati sulle regole fornita dalla sicurezza di Seam, Drools richiede che i seguenti file jar siano distribuiti insieme al progetto:

- drools-compiler.jar
- drools-core.jar
- janino.jar
- antlr-runtime.jar
- mvel14.jar

15.6.9.2. Configurazione

La configurazione per `RuleBasedPermissionResolver` richiede che una base di regole venga prima configurata in `components.xml`. Per difetto si aspetta che questa base di regole sia chiamata `securityRules`, come nel seguente esempio:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:drools="http://jboss.com/products/seam/drools"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.1.xsd
                    http://jboss.com/products/seam/components  http://jboss.com/products/seam/
components-2.1.xsd
               http://jboss.com/products/seam/drools http://jboss.com/products/seam/drools-2.1.xsd"
                     http://jboss.com/products/seam/security  http://jboss.com/products/seam/
security-2.1.xsd">

    <drools:rule-base name="securityRules">
       <drools:rule-files>
          <value
>/META-INF/security.drl</value>
       </drools:rule-files>
    </drools:rule-base>

 </components
>
```

Il nome predefinito della base di regole può essere modificato specificando la proprietà `security-rules` per `RuleBasedPermissionResolver`:

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

Una volta che il componente `RuleBase` è configurato, è il momento di scrivere le regole di sicurezza.

15.6.9.3. Scrivere le regole di sicurezza

Il primo passo per scrivere delle regole di sicurezza è di creare un nuovo file di regole nella cartella `/META-INF` del file jar dell'applicazione. Di solito questo file dovrebbe essere chiamato qualcosa come `security.drl`, comunque lo si può chiamare nel modo che si preferisce purché sia configurato in maniera corrispondente in `components.xml`.

Dunque, che cosa deve contenere il file delle regole di sicurezza? A questo punto potrebbe essere una buona idea almeno sbirciare nella documentazione Drools, comunque per partire ecco un esempio estremamente semplice:

```
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
```

Dividiamolo passo per passo. La prima cosa che vediamo è la dichiarazione del package. Un package in Drools è essenzialmente una collezione di regole. Il nome del package può essere qualsiasi, non è in relazione con niente che sia al di fuori della visibilità della base di regole.

La cosa successiva che possiano notare è un paio di dichiarazioni `import` per le classi `PermissionCheck` e `Role`. Questi `import` informano il motore di regole che all'interno delle nostre regole faremo riferimento a queste classi.

Infine abbiamo il codice della regola. Ogni regola all'interno di un package deve avere un nome univoco (di solito descrive lo scopo della regola). In questo caso la nostra regola si chiama

`CanUserDeleteCustomers` e verrà usata per verificare se ad un utente è consentito di cancellare un record relativo ad un cliente.

Guardando il corpo della definizione della regola si possono notare due distinte sezioni. Le regole hanno quello che è noto come lato sinistro (LHS, left hand side) e un lato destro (RHS, right hand side). Il lato sinistro consiste nella parte condizionale della regola, cioè l'elenco delle condizioni che devono essere soddisfatte per la regola si applichi. Il lato sinistro è rappresentato dalla sezione `when`. Il lato destro è la conseguenza, o la parte di azione della regola che si applica solo se tutte le condizioni del lato sinistro sono verificate. Il lato destro è rappresentato dalla sezione `then`. La fine della regola è stabilita dalla linea `end`.

Se guardiamo la parte sinistra della regola vediamo che ci sono due condizioni. Esaminiamo la prima condizione:

```
c: PermissionCheck(target == "customer", action == "delete")
```

Letta in inglese questa condizione dice che all'interno della working memory deve esistere un oggetto `PermissionCheck` con una proprietà `target` uguale a "customer" e una proprietà `action` uguale a "delete".

Dunque cos'è la working memory? Nota anche come "stateful session" nella terminologia Drools, la working memory è un oggetto collegato alla sessione che contiene le informazioni contestuali che sono richieste dal motore di regole per prendere una decisione sul controllo di permesso. Ogni volta che il metodo `hasPermission()` viene chiamato, viene creato un oggetto, o *Fatto*, temporaneo `PermissionCheck`, e viene inserito nella working memory. Questo `PermissionCheck` corrisponde esattamente al permesso che si sta controllando, così, ad esempio, se viene chiamato `hasPermission("account", "create")` allora verrà inserito nella working memory un oggetto `PermissionCheck` con `target` uguale a "account" e `action` uguale a "create", per la durata del controllo di permesso.

Accanto al fatto `PermissionCheck` c'è anche un fatto `org.jboss.seam.security.Role` per ogni ruolo di cui l'utente autenticato è membro. Questi fatti `Role` sono sincronizzati con i ruoli dell'utente autenticato all'inizio di ogni controllo di permesso. Di conseguenza qualsiasi oggetto `Role` che venisse inserito nella working memory nel corso del controllo di permesso sarebbe rimosso prima che il controllo di permesso successivo avvenga, a meno che l'utente autenticato non sia effettivamente membro di quel ruolo. Insieme ai fatti `PermissionCheck` e `Role` la working memory contiene anche l'oggetto `java.security.Principal` che era stato creato come risultato del processo di autentifica.

E' anche possibile inserire ulteriori fatti nella working memory chiamando `RuleBasedPermissionResolver.instance().getSecurityContext().insert()`, passando l'oggetto come parametro. Fanno eccezione a questo gli oggetti `Role` che, come già detto, sono sincronizzati all'inizio di ciascun controllo di permesso.

Tornando al nostro esempio, possiamo anche notare che la prima linea della nostra parte sinistra ha il prefisso `c:`. Questa è una dichiarazione di variabile ed è usata per fare riferimento all'oggetto

rilevato dalla condizione (in questo caso il `PermissionCheck`). Passando alla seconda linea della nostra parte sinistra vediamo questo:

```
Role(name == "admin")
```

Questa condizione dichiara semplicemente che ci deve essere un oggetto `Role` con un `name` uguale ad "admin" nella working memory. Come già menzionato, i ruoli dell'utente sono inseriti nella working memory all'inizio di ogni controllo di permesso. Così, mettendo insieme entrambe le condizioni, questa regola in pratica dice "mi attiverò quando ci sarà un controllo per il permesso `customer:delete` e l'utente è un membro del ruolo `admin`".

Quindi qual è la conseguenza dell'attivazione della regola? Diamo un'occhiata alla parte destra della regola:

```
c.grant()
```

La parte destra è costituita da codice Java e, in questo caso, esso invoca il metodo `grant()` dell'oggetto `c` il quale, come già detto, è una variabile che rappresenta l'oggetto `PermissionCheck`. Insieme alle proprietà `name` e `action`, nell'oggetto `PermissionCheck` c'è anche una proprietà `granted` che inizialmente è impostata a `false`. Chiamando `grant()` su un `PermissionCheck` la proprietà `granted` viene impostata a `true`, il che significa che il controllo di permesso è andato a buon fine, consentendo all'utente di portare avanti qualsiasi azione per cui il controlo di permesso era stato inteso.

15.6.9.4. Permessi con obiettivi non stringa

Finora abbiamo visto solo controlli di permesso per obiettivi di tipo stringa. E' naturalmente possibile scrivere regole di sicurezza anche per obiettivi del permesso di tipo più complesso. Ad esempio, supponiamo che si voglia scrivere una regola di sicurezza che consenta agli utenti di creare un commento in un blog. La seguente regola mostra come questo possa essere espresso, richiedendo che l'obiettivo del controllo di permesso sia un'istanza di `MemberBlog` e anche che l'utente correntemente autenticato sia un membro del ruolo `user`:

```
rule CanCreateBlogComment
  no-loop
  activation-group "permissions"
when
  blog: MemberBlog()
  check: PermissionCheck(target == blog, action == "create", granted == false)
  Role(name == "user")
then
  check.grant();
```

```
end
```

15.6.9.5. Controlli di permesso

E' possibile realizzare dei controlli di permesso (che consentono l'accesso a tutte le funzioni per un determinato obiettivo) basati su wildcard omettendo il vincolo `action` per il `PermissionCheck` nella regola, in questo modo:

```
rule CanDoAnythingToCustomersIfYouAreAnAdmin
when
  c: PermissionCheck(target == "customer")
  Role(name == "admin")
then
  c.grant();
end;
```

Questa regola consente agli utenti con il ruolo `admin` di eseguire *qualsiasi* azione per qualsiasi controllo di permesso su `customer`.

15.6.10. PersistentPermissionResolver

Un altro risolutore di permessi incluso in Seam, il `PersistentPermissionResolver` consente di caricare i permessi da un dispositivo di memorizzazione persistente, come una database relazionale. Questo risolutore di permessi fornisce una sicurezza orientata alle istanze in stile ACL (Access Control List), permettendo di assegnare specifici permessi sull'oggetto a utenti e ruoli. Allo stesso modo permette inoltre di assegnare in modo persistente permessi con un nome arbitrario (non necessariamente basato sull'oggetto o la classe).

15.6.10.1. Configurazione

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}"/>
```

15.6.10.2. 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:

Tabella 15.8. PermissionStore interface

Tipo restituito	Metodo	Descrizione
`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

Tipo restituito	Metodo	Descrizione
		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.
`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

Tipo restituito	Metodo	Descrizione
		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).

15.6.10.3. 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"/>
```

15.6.10.3.1. 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:

Tabella 15.9. Entity Permission annotations

Annotazione	Target	Descrizione
`@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

Annotazione	Target	Descrizione
		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

Annotazione	Target	Descrizione
		`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: `= "u", roleValue = "r")`

15.6.10.3.2. 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.

15.6.10.3.3. 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:

Tabella 15.10. Class Permission Annotations

Annotazione	Target	Descrizione
`@Permissions`	`TYPE`	A container annotation, this annotation may contain an array of `@Permission` annotations.
`@Permission`	`TYPE`	This annotation defines a single allowable permission

Annotazione	Target	Descrizione
		action for the target class. Its `action` property must be specified, and an optional `mask` property may also be specified if permission actions are to be persisted as bitmasked values (see 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`.

15.6.10.3.4. Permission masks

By default, multiple permissions for the same target object and recipient will be persisted as a single database record, with the `action` property/column containing a 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.

15.6.10.3.5. 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.

15.6.10.3.6. 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 {
```

15.6.10.3.7. 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`".

15.7. 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.

15.7.1. 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`:

Tabella 15.11. PermissionManager API methods

Tipo restituito	Metodo	Descrizione
`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 permission store. Returns true if the operation was successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	Persists (grants) the specified list

Tipo restituito	Metodo	Descrizione
		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

Tipo restituito	Metodo	Descrizione
		dependent on the `@Permission` annotations configured on the target object's class.

15.7.2. 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.

Tabella 15.12. Permission Management Security Permissions

Metodo	Oggetto del permesso	Azione del permesso
`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`
`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`

15.8. 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.

15.8.1. 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.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-2.1.xsd"
```

```
    no-conversation-view-id="/home.xhtml"
    login-view-id="/login.xhtml"
    http-port="8080"
    https-port="8443"
>
```

15.9. 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.

15.9.1. 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
>
```

15.9.2. 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.

15.9.3. 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;
  }
}
```

15.10. Security Events

The following table describes a number of events (see *Capitolo 6, Eventi, interceptor e gestione delle eccezioni*) raised by Seam Security in response to certain security-related events.

Tabella 15.13. Security Events

Event Key	Descrizione
`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`	

Event Key	Descrizione
	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.

15.11. 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`.

15.12. 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 precendence 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();
  }
}
```

Avvertimento

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.

15.13. 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 chosing. 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.

15.13.1. 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.

15.13.2. 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.

15.13.3. 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>
```

Thie `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.

15.13.4. 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.

15.13.5. 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.

Internazionalizzazione, localizzazione e temi

Seam rende facile la costruzione di applicazioni internazionali. Prima di tutto verranno percorse le varie fasi necessarie per rendere internazionale e tradotta un'applicazione. In seguito si darà un'occhiata al modo in cui Seam gestisce i gruppi di stringhe associate ai componenti (resource bundle).

16.1. Internazionalizzare un'applicazione

Un'applicazione JEE consiste di molti componenti ed ognuno di essi deve essere configurato opportunamente affinché l'applicazione venga tradotta.

Partendo dalla base, il primo passo è assicurarsi che il server e il client del database utilizzino la codifica di caratteri corretta per la traduzione. Di solito si vorrà utilizzare UTF-8. Come fare questo non è oggetto di questa guida.

16.1.1. Configurazione dell'application server

Per essere sicuri che l'application server riceva i parametri delle richieste dai client nella codifica corretta occorre configurare il connettore Tomcat. Se si usa Tomcat o JBoss AS, aggiungere l'attributo `URIEncoding="UTF-8"` alla configurazione del connettore. Per JBoss AS 4.2 modificare `${JBOSS_HOME}/server/(default)/deploy/jboss-web.deployer/server.xml`:

```
<Connector port="8080" URIEncoding="UTF-8"/>
```

C'è un'alternativa che è probabilmente migliore. E' possibile dire a JBoss AS che la codifica dei parametri della richiesta deve essere ricavata dalla richiesta:

```
<Connector port="8080" useBodyEncodingForURI="true"/>
```

16.1.2. Traduzione delle stringhe dell'applicazione

Ci sarà bisogno di tradurre le stringhe per tutti i *messaggi* dell'applicazione (per esempio le etichette dei campi nelle pagine). In primo luogo occorre assicurarsi che il resource bundle sia codificato utilizzando la giusta codifica di carattere. Per default viene usato ASCII. Benché la codifica ASCII sia sufficiente per molte lingue, essa non fornisce i caratteri per tutte le lingue.

I resource bundles devono essere creati in ASCII, oppure devono utilizzare una notazione Unicode per rappresentare i caratteri Unicode. Poiché un file `.properties` non viene compilato in byte-

code, non c'è modo di dire alla JVM quale codifica caratteri utilizzare. Perciò occorre usare caratteri ASCII oppure usare la notazione Unicode per i caratteri che non fanno parte dell'insieme ASCII. E' possibile rappresentare un carattere Unicode in un file Java usando la notazione \uXXXX, dove XXXX è la rappresentazione esadecimale del carattere.

E' possibile scrivere la traduzione delle etichette (
<xlink>Etichette</xlink>
) nei resource bundles con la codifica del proprio sistema e poi convertire il contenuto del file nel formato con le notazioni Unicode attraverso lo strumento `native2ascii` fornito con JDK. Questo strumento converte un file scritto nella codifica originale in uno dove i caratteri non-ASCII sono rappresentati come sequenze di notazioni Unicode.

L'uso di questo strumento è descritto *qui per Java 5* [http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#intl] oppure *qui per Java 6* [http://java.sun.com/javase/6/docs/technotes/tools/#intl]. Ad esempio, per convertire un file da UTF-8:

```
$ native2ascii -encoding UTF-8 messages_cs.properties >
 messages_cs_escaped.properties
```

16.1.3. Altre impostazioni per la codifica

Occorre essere sicuri che le pagine mostrino i dati tradotti e i messaggi utilizzando il corretto insieme di caratteri e che anche i dati inviati usino usino la codifica corretta.

Per impostare la codifica dei caratteri per le pagine occorre utilizzare la tag `<f:view locale="cs_CZ"/>` (in questo modo diciamo a JSF di usare il locale Ceco). Si può voler modificare la codifica del documento XML stesso se si vuole includere stringhe tradotte all'interno dell'XML. Per fare questo occorre modificare l'attributo encoding nella dichiarazione XML `<xml version="1.0" encoding="UTF-8">` con il valore desiderato.

Anche JSF/Facelets dovrebbe inviare tutte le richieste utilizzando la codifica caratteri specificata, ma per essere sicuri che tutte le richieste che non specificano un valore di codifica abbiamo il valore corretto è possibile forzare la codifica delle richieste utilizzando un filtro Servlet. Questo si configura in `components.xml`:

```
<web:character-encoding-filter encoding="UTF-8"
  override-client="true"
  url-pattern="*.seam" />
```

16.2. Traduzioni

Ogni sessione utente registrata ha associata un'istanza di `java.util.Locale` (disponibile nell'applicazione come un componente chiamato `locale`). In condizioni normali non sarà

necessario fare alcuna configurazione particolare per impostare la lingua. Seam delega a JSF il compito di determinare la lingua attiva:

- Se c'è un linguaggio associato con la richiesta HTTP (il linguaggio del browser), e questo linguaggio è presente nella lista delle lingue gestite in `faces-config.xml`, allora questa lingua verrà usata per il resto della sessione.
- Altrimenti, se in `faces-config.xml` è specificata una lingua di default, questa lingua verrà usata per il resto della sessione.
- Altrimenti viene usata la lingua di default del server.

E' *possibile* impostare la lingua manualmente tramite le proprietà di configurazione di Seam `org.jboss.seam.international.localeSelector.language`, `org.jboss.seam.international.localeSelector.country` e `org.jboss.seam.internationale.localeSelector.variant`, ma non c'è una vera buona ragione per farlo.

E' comunque utile consentire all'utente di impostare la lingua manualmente tramite l'interfaccia utente. Seam fornisce una funzionalità per sovrascrivere il linguaggio determinato dall'algoritmo descritto sopra. Tutto ciò che è necessario fare è aggiungere il seguente brano ad una form in una pagina JSP o Facelets:

```
<h:selectOneMenu value="#{localeSelector.language}">
   <f:selectItem itemLabel="English" itemValue="en"/>
   <f:selectItem itemLabel="Deutsch" itemValue="de"/>
   <f:selectItem itemLabel="Francais" itemValue="fr"/>
   <f:selectItem itemLabel="Italiano" itemValue="it"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
   value="#{messages['ChangeLanguage']}"/>
```

Oppure, se si vuole mostrare una lista delle lingue gestite da `faces-config.xml`, si può usare:

```
<h:selectOneMenu value="#{localeSelector.localeString}">
   <f:selectItems value="#{localeSelector.supportedLocales}"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
   value="#{messages['ChangeLanguage']}"/>
```

Quando l'utente seleziona una voce dal menu a discesa e poi fa click sul bottone di comando, la lingua di Seam e di JSF viene sovrascritta per il resto della sessione.

Tutto ciò porta a domandarsi dove siano definite le lingue gestite. Tipicamente nell'elemento `<locale-config>` del file di configurazione JSF (/META-INF/faces-config.xml) si indica una lista di lingue per le quali si dispone dei corrispondenti resource bundle. Ad ogni modo si è imparato ad apprezzare che il meccanismo di configurazione dei componenti Seam è più completo di quello fornito in Java EE. Per questa ragione è possibile configurare le lingue gestite e la lingua di default del server usando il componente `org.jboss.seam.international.localeConfig`. Per usarlo occorre prima dichiarare il namespace XML per il pacchetto international di Seam nel descrittore dei componenti Seam, quindi definire la lingua di default e le lingue gestite come segue:

```
<international:locale-config default-locale="fr_CA" supported-locales="en fr_CA fr_FR it_IT"/>
```

Ovviamente se c'è la dichiarazione che una certa lingua è gestita, sarà meglio fornire il resource bundle corrispondente! Nel prossimo capitolo si imparerà come si definiscono le etichette per una lingua specifica.

16.3. Etichette

JSF gestisce l'internazionalizzazione delle etichette e del testo descrittivo nell'interfaccia utente tramite l'uso di `f:loadBundle>`. Questo approccio è possibile nelle applicazioni Seam. In alternativa è possibile sfruttare i vantaggi offerti dal componente Seam `messages` per mostrare label costruite tramite modelli con espressioni EL.

16.3.1. Definire le etichette

Seam fornisce un `java.util.ResourceBundle` (disponibile all'applicazione come un `org.jboss.seam.core.resourceBundle`). Occorre rendere disponibili le nostre etichette tradotte tramite questo speciale resource bundle. Per default il resource bundle usato da Seam si chiama `messages` così che occorre definire le etichette in file chiamati `messages.properties`, `messages_en.properties`, `messages_en_AU.properties`, ecc. Questi file di solito risiedono nella cartella `WEB-INF/classes`.

Quindi, in `messages_en.properties`:

```
Hello=Hello
```

E in `messages_en_AU.properties`:

```
Hello=G'day
```

E' possibile indicare un nome diverso per il resource bundle impostando la proprietà di configurazione Seam `org.jboss.seam.core.resourceLoader.bundleNames`. E' possibile

persino specificare un elenco di nomi di resource bundle sui quali devono essere ricercati i messaggi (a partire dall'ultimo).

```
<core:resource-loader>
   <core:bundle-names>
      <value>mycompany_messages</value>
      <value>standard_messages</value>
   </core:bundle-names>
</core:resource-loader>
```

Se si vuole definire un messaggio solo per una particolare pagina, è possibile specificarlo in un resource bundle con lo stesso nome dell'identificativo della view JSF, omettendo il `/` iniziale e l'estensione del file finale. Così è possibile mettere il nostro messaggio in `welcome/hello_en.properties` se si desidera mostrare il messaggio solo in `/welcome/hello.jsp`.

E' anche possibile specificare esplicitamente un nome di resource bundle in `pages.xml`:

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

Quindi possiamo usare i messaggi definiti in `HelloMessages.properties` in `/welcome/hello.jsp`.

16.3.2. Mostrare le etichette

Se si definiscono le etichette utilizzando il resource bundle di Seam è possibile usarle senza dover scrivere `<f:loadBundle... />` in ogni pagina. E' possibile invece scrivere semplicemente:

```
<h:outputText value="#{messages['Hello']}"/>
```

oppure:

```
<h:outputText value="#{messages.Hello}"/>
```

Ancora meglio, i messaggi stessi possono contenere espressioni EL:

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G'day, #{user.firstName}
```

E' possibile anche usare i messaggi nel codice:

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

16.3.3. Messaggi Faces

Il componente `facesMessages` è un modo super-conveniente per mostare messaggi di conferma o di errore all'utente. La funzionalità che è stata appena descritta funziona anche per i messaggi faces:

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
   @In FacesMessages facesMessages;

   public String sayIt() {
      facesMessages.addFromResourceBundle("Hello");
   }
}
```

Questo mostrerà `Hello, Gavin King` oppure `G'day, Gavin`, a seconda della lingua dell'utente.

16.4. Fusi orari

C'è anche un'istanza a livello sessione di `java.util.Timezone`, chiamata `org.jboss.seam.international.timezone`, e un componente Seam per cambiare il fuso orario chiamato `org.jboss.seam.interanational.timezoneSelector`. Per default il fuso orario è il fuso orario di default del server. Purtroppo le specifiche JSF dicono che tutte le date e orari devono essere considerati come UTC e mostrati come UTC a meno che un fuso orario non sia esplicitamente specificato usando `<f:convertDateTime>`. Questo è un comportamento di default estremamente sconveniente.

Seam modifica questo comportamento e imposta il fuso orario di tutte le date e orari al fuso orario di Seam. In più, Seam fornisce la tag `<s:convertDateTime>` che esegue sempre questa conversione nel fuso orario di Seam.

Seam fornisce anche un converter di date di default per convertire una valore stringa in una data. Questo risparmia di dover specificare un converter sui campi d'input che sono semplicemente catturati come data. Il pattern viene selezionato in accordo con il locale dell'utente e la timezone viene selezionata come descritto sopra.

16.5. Temi

Le applicazioni Seam sono anche molto facilmente personalizzabili nell'aspetto. Le API per i temi sono molto simili alle API per la traduzione, ma ovviamente questi due concetti sono ortogonali e alcune applicazione gestiscono sia le traduzioni che i temi.

Prima di tutto occorre configurare l'insieme dei temi gestiti:

```
<theme:theme-selector cookie-enabled="true">
   <theme:available-themes>
      <value>default</value>
      <value>accessible</value>
      <value>printable</value>
   </theme:available-themes>
</theme:theme-selector>
```

Notare che il primo tema elencato è il tema di default.

I temi sono definiti in file di proprietà con lo stesso nome del tema. Ad esempio, il tema `default` è definito come un insieme di voci in `default.properties`. Ad esempio `default.properties` potrebbe definire:

```
css ../screen.css
template /template.xhtml
```

Di solito le voci nel resource bundle di un tema saranno percorsi a fogli di stile CSS o immagini e nomi di modelli facelets (a differenza dei resource bundle per le traduzioni che normalmente contengono testo).

Ora è possibile usare queste voci nella pagine JSP o facelets. Ad esempio, per gestire con un tema il foglio di stile di una pagina facelets:

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

Oppure, quando la definizione della pagina risiede in una sottocartella:

```
<link href="#{facesContext.externalContext.requestContextPath}#{theme.css}"
   rel="stylesheet" type="text/css" />
```

In modo più flessibile, facelets consente di gestire con i temi il modello usato da un `<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}">
```

Così come per selezionare la lingua, c'è un componente che consente all'utente di cambiare liberamente il tema:

```
<h:selectOneMenu value="#{themeSelector.theme}">
   <f:selectItems value="#{themeSelector.themes}"/>
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme"/>
```

16.6. Registrare la scelta della lingua e del tema tramite cookies

Le selezioni della lingua, del tema e del fuso orario gestiscono tutte la registrazione della scelta in un cookie. Basta impostare la proprietà `cookie-enabled` 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

I siti web orientati alla collaborazione tra utenti richiedono un linguaggio per marcare in modo comprensibile il testo formattato da inserire nei post di un forum, nelle pagine wiki, nei commenti, ecc. Seam fornisce il controllo `<s:formattedText/>` per mostrare il testo formattato in modo conforme con il linguaggio *Seam Text*. Seam Text è realizzato utilizzando un interprete basato su ANTLR. Comunque non c'è bisogno di sapere niente di ANTLR per utilizzarlo.

17.1. Formattazione di base

Ecco un semplice esempio:

```
E' semplice rendere il testo *evidenziato*, |a spaziatura fissa|,
~cancellato~, sovra^scritto^ o _sottolineato_.
```

Se mostriamo questo testo usando `<s:formattedText/>`, otteniamo il seguente codice HTML:

```
E' semplice rendere il testo *evidenziato*, |a spaziatura fissa|,
~cancellato~, sovra^scritto^ o _sottolineato_.
```

E' possibile usare una riga vuota per indicare un nuovo paragrafo e un + per indicare un titolo:

```
+Questo è un grande titolo
/Dovrai/ avere del testo dopo il titolo!

++Questo è un titolo più piccolo
Questo è il primo paragrafo. Lo possiamo dividere in più
righe, ma per terminarlo serve una riga vuota.

Questo è il secondo paragrafo.
```

(Notare che un semplice a-capo viene ignorato, è necessaria una riga vuota per avere il testo in un nuovo paragrafo). Questo è il codice HTML risultante:

```
<h1
>Questo è un grande titolo</h1>
<p>
<i
>Dovrai</i
```

```
> avere del testo dopo il titolo!
</p>

<h2
>Questo è un titolo più piccolo</h2>
<p>
Questo è il primo paragrafo. Lo possiamo dividere in più
righe, ma per terminarlo serve una riga vuota.
</p>

<p>
Questo è il secondo paragrafo.
</p
>
```

Le liste ordinate sono generate dal carattere #. Le liste non ordinate dal carattere =:

```
Una lista ordinata:

#prima voce
#seconda voce
#e anche la /terza/ voce

Una lista non ordinata:

=una voce
=un'altra voce
```

```
<p>
Una lista ordinata:
</p>

<ol
>
<li
>prima voce</li>
<li
>seconda voce</li>
<li
>e anche una <i
>terza</i
```

```
> voce</li>
</ol>

<p>
Una lista non ordinata:
</p>

<ul>
<li
>una voce</li>
<li
>un'altra voce</li>
</ul
>
```

I brani con citazioni devono essere racchiusi tra virgolette:

```
L'altro ragazzo disse:

"Nyeah nyeah-nee
/nyeah/ nyeah!"

Ma cosa pensi abbia voluto dire con "nyeah-nee"?
```

```
<p>
L'altro ragazzo disse:
</p>

<q
>Nyeah nyeah-nee
<i
>nyeah</i
> nyeah!</q>

<p>
Ma cosa pensi abbia voluto dire con <q
>nyeah-nee</q
>?
</p
>
```

17.2. Inserire codice e testo con caratteri speciali

Caratteri speciali come `*`, `|` e `#`, e anche i caratteri HTML come `<`, `>` e `&` possono essere inseriti usando il carattere di escape `\`:

```
E' possibile scrivere equazioni come 2\*3\=6 e tag HTML
come \<body\
> usando il carattere di escape: \\.
```

```
<p>
E' possibile scrivere equazioni come 2*3=6 e tag HTML
come &lt;body&gt; usando il carattere di escape: \.
</p
>
```

Ed è possibile citare blocchi di codice usando l'apice inverso (purtroppo l'apice inverso non c'è nella tastiera italiana, ndt):

```
Il mio codice non funziona:

`for (int i=0; i<100; i--)
{
   doSomething();
}`

Qualche idea?
```

```
<p>
Il mio codice non funziona:
</p>

<pre
>for (int i=0; i&lt;100; i--)
{
   doSomething();
}</pre>

<p>
Qualche idea?
```

```
</p
>
```

Notare che la formattazione in linea a spaziatura fissa considera sempre i caratteri speciali (la maggior parte del testo formattato a spaziatura fissa in effetti è codice o tag con molti caratteri speciali). Così, ad esempio, è possibile scrivere:

```
Questo è un |<tag attribute="value"/>| esempio.
```

senza bisogno di usare il carattere di escape per i caratteri all'interno del brano formattato a spaziatura fissa. Lo svantaggio è che non è possibile formattare il testo in linea a spaziatura fissa in altri modi (corsivo, sottolineato, e così via).

17.3. Link

Un link può essere creato utilizzando la seguente sintassi:

```
Vai al sito web di Seam [=
>http://jboss.com/products/seam].
```

Oppure, se si vuole specificare il testo del link:

```
Vai al [sito web di Seam=
>http://jboss.com/products/seam].
```

Per gli utenti esperti è possibile anche personalizzare l'interprete Seam Text in modo da comprendere i link in formato wiki scritti usando questa sintassi.

17.4. Inserire codice HTML

Il testo può anche includere un certo sottoinsieme limitato di HTML (non c'è da preoccuparsi, il sottoinsieme è stato scelto in modo da essere sicuro rispetto alla possibilità di attacchi di tipo cross-site scripting). Questo è utile per creare dei link:

```
Potresti voler fare un link a <a href="http://jboss.com/products/seam"
>qualcosa di
forte</a
>, oppure includere un'immagine: <img src="/logo.jpg"/>
```

E per creare delle tabelle:

```
<table>
   <tr
><td
>Nome:</td
><td
>Gavin</td
></tr>
   <tr
><td
>Cognome:</td
><td
>King</td
></tr>
</table
>
```

Ma è possibile fare molto di più, volendo!

Generazione di PDF con iText

Seam adesso include un componente per la generazione di documenti usando iText. Il primo focus del supporto di Seam ai documenti iText è per la generazione dei documenti PDF, ma Seam offre anche un supporto base per la generazione di documenti RTF.

18.1. Utilizzo del supporto PDF

Il supporto a iText è fornito da `jboss-seam-pdf.jar`. Questo JAR contiene i controlli JSF di iText, che sono usati per costruire le viste che possono generare il PDF, e il componente DocumentStore, che serve i documenti renderizzati per l'utente. Per includere il supporto PDF nell'applicazione, si metta `jboss-seam-pdf.jar` nella directory `WEB-INF/lib` assieme al file JAR di iText. Non serve alcuna ulteriore configurazione per usare il supporto a iText di Seam.

Il modulo iText di Seam richiede l'uso dei Facelets come tecnologia per la vista.Versioni future della libreria potrebbero supportare anche l'uso di JSP. In aggiunta, si richiede l'uso del pacchetto seam-ui.

Il progetto `examples/itext` contiene un esempio di supporto PDF. Viene mostrato il corretto impacchettamento per il deploy e l'esempio contiene un gran numero di funzionalità per la generazione PDF attualmente supportate.

18.1.1. Creazione di un documento

`<p:document>`	*Descrizione* I documenti vengono generati dai file XHTML facelet usando dei tag nel namespace `http://jboss.com/products/seam/pdf`. I documenti dovrebbero sempre avere il tag `document` alla radice del documento. Il tag `document` prepara Seam a generare un documento nel DocumentStore e compie un redirect HTML a quel contenuto memorizzato. *Attributi* • `type` — Il tipo di documento da produrre. Valori validi sono `PDF`, `RTF` e `HTML`. Seam imposta come default la generazione PDF, e molte caratteristiche funzionano correttamente solo quando si generano documenti PDF. • `pageSize` — La dimensionedella pagina da generare. I valori maggiormente usati dovrebbero essere `LETTER` e `A4`. Una lista completa delle dimensioni di pagina supportate si trova nella classe `com.lowagie.text.PageSize`. In alternativa, pageSize può direttamente fornire la larghezza e l'altezza della pagina. Per

esempio, il valore "612 792" è equivalente alla dimensione della pagina LETTER.

- `orientation` — L'orientamento della pagina. Valori validi sono `portrait` e `landscape`. Nella modalità landscape, i valori dell'altezza e della larghezza della pagina sono invertiti.
- `margins` — Valori di margine left, right, top e bottom.
- `marginMirroring` — Indica che le impostazioni dei margini dovrebbero essere invertite in pagine alternate.
- `disposition` — Quando si generano PDF in un browser, questo determina la `Content-Disposition` HTTP del documento. Valori validi sono `inline`, che indica che il documento deve essere mostrato in una finestra del browser se possibile, e `attachment`, che indica che il documento deve essere trattato come download. Il valore di default è `inline`.
- `fileName` — Per gli allegati questo valore sovrascrive il nome del file scaricato.

Attributi dei metadati

- `title`
- `subject`
- `keywords`
- `author`
- `creator`

Utilizzo

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
>
  The document goes here.

</p:document
>
```

18.1.2. Elementi base per il testo

I documenti utili dovranno contenere più che il solo testo; comunque i componenti standard UI sono idonei per la generazione HTML e non sono utili per generare contenuto in PDF.

Invece Seam fornisce dei componenti UI speciali per generare contenuto in PDF idoneo. Tag quali `<p:image>` e `<p:paragraph>` sono la base per i semplici documenti. Tag come `<p:font>` forniscono informazioni di stile a tutto il contenuto che sta intorno.

`<p:paragraph>`	*Descrizione*
	
	La maggior parte dell'uso del testo dovrebbe essere sezionato in paragrafi, affinché i frammenti del testo possano scorrere, formattati ed con uno stile in gruppi logici.
	
	Attributi
	
	• `firstLineIndent`
	• `extraParagraphSpace`
	• `leading`
	• `multipliedLeading`
	• `spacingBefore` — Lo spazio bianco da inserire prima dell'elemento.
	• `spacingAfter` — Lo spazio bianco da inserire dopo l'elemento.
	• `indentationLeft`
	• `indentationRight`
	• `keepTogether`
	
	Utilizzo
	
	`<p:paragraph alignment="justify">`
	`  This is a simple document.  It isn't very fancy.`
	`</p:paragraph`
	`>`

`<p:text>`	*Descrizione* Il tag `text` consente ai frammenti del testo di essere prodotti dai dati dell'applicazione usando normali meccanismi convertitori JSF. E' molto simile al tag `outputText` impiegato quando si generano documenti HTML. *Attributi*

	• `value` — Il valore da visualizzare. Questo sarà tipicamente un'espressione di binding. *Utilizzo* `<p:paragraph>   The item costs <p:text value="#{product.price}">   <f:convertNumber type="currency" currencySymbol="$"/>   </p:text> </p:paragraph >`
`<p:html>`	*Descrizione* Il tag `html` genera contenuto HTML in PDF. *Attributi* • `value` — Il testo da visualizzare. *Utilizzo* `<p:html value="This is HTML with <b >some markup</b >." /> <p:html>   <h1 >This is more complex HTML</h1>        <li >one     <li >two     <li >three        </p:html>  <p:html>   <s:formattedText value="*This* is |Seam Text| as HTML. It's very^cool^." /> </p:html>`

```
</p:html
>
```

`<p:font>`	*Descrizione* Il tag font definisce il font di default da usarsi per tutto il testo contenuto in esso. *Attributi* • `name` — Il nome del font, per esempio: `COURIER`, `HELVETICA`, `TIMES-ROMAN`, `SYMBOL` o `ZAPFDINGBATS`. • `size` — La dimensione del punto nel font. • `style` — Gli stili del font. Una combinazione di: `NORMAL`, `BOLD`, `ITALIC`, `OBLIQUE`, `UNDERLINE`, `LINE-THROUGH`. • `encoding` — La codifica del set di caratteri. *Utilizzo* `<p:font name="courier" style="bold" size="24"><p:paragraph>My Title</p:paragraph></p:font>`
`<p:newPage>`	*Descrizione* `p:newPage` inserisce un'interruzione di pagina. *Utilizzo* `<p:newPage />`
`<p:image>`	*Descrizione* `p:image` inserisce un'immagine in un documento. Le immagini possono essere caricate dal classpath o dal contesto dell'applicazione web usando l'attributo `value`. Le risorse possono anche essere generate dinamicamente dal codice dell'applicazione. L'attributo `imageData` può specificare un'espressione di value binding il cui valore è un oggetto `java.awt.Image`.

	Attributi
	
	• `value` — Un nome di risorsa oppure un binding di metodo ad un'immagine generata dall'applicazione.
	• `rotation` — La rotazione dell'immagine in gradi.
	• `height` — L'altezza dell'immagine.
	• `width` — La larghezza dell'immagine.
	• `alignment`— L'allineamento dell'immagine. (vedere *Sezione 18.1.7.2, «Valori per l'allineamento»* per i possibili valori)
	• `alt` — Testo alternativo per la rappresentazione dell'immagine.
	• `indentationLeft`
	• `indentationRight`
	• `spacingBefore` — Lo spazio bianco da inserire prima dell'elemento.
	• `spacingAfter` — Lo spazio bianco da inserire dopo l'elemento.
	• `widthPercentage`
	• `initialRotation`
	• `dpi`
	• `scalePercent` — Il fattore di scala (come percentuale) da usare per l'immagine. Questo può esprimersi come valore di percentuale singola oppure come valori di due percentuali che rappresentano percentuali separate per la scala lungo x o lungo y.
	• `wrap`
	• `underlying`
	
	Utilizzo
	
	`<p:image value="/jboss.jpg" />`
	
	`<p:image value="#{images.chart}" />`

`<p:anchor>`	*Descrizione*

`p:anchor` definisce i link cliccabili da un documento. Supporta i seguenti attributi:

Attributi

- `name` — Il nome di una destinazione d'ancora dentro il documento.
- `reference` — La destinazione a cui il link di riferisce. I link ad altri punti del documento dovrebbero iniziare con un "#". Per esempio, "#link1" si riferisce ad una posizione dell'ancora con il `name` impostato a `link1`. I link possono anche essere URL completi ad un punto della risorsa fuori dal documento.

Utilizzo

```
<p:listItem
><p:anchor reference="#reason1"
>Reason 1</p:anchor
></p:listItem
>
...
<p:paragraph>
   <p:anchor name="reason1"
>It's the quickest way to get "rich"</p:anchor
>
   ...
</p:paragraph
>
```

18.1.3. Intestazioni e pié di pagina

`<p:header>`
 	
`<p:footer>`
	Descrizione
	
	I componenti `p:header` e `p:footer` forniscono la possibilità di collocare il testo per l'intestazione ed il pié di pagina in ogni pagina del documento generato. Le dichiarazioni di header e footer dovrebbero apparire all'inizio del documento.
	
	Attributi
	
	• `alignment` — L'allineamento della sezione header/footer. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i valori dell'allineamento)

- `backgroundColor` — Il colore del background di header/footer. (Si veda *Sezione 18.1.7.1, «Valori dei colori»* per i valori dei colori)
- `borderColor` — Il colore del bordo di header/footer. I singoli lati dei bordi possono essere impostati usando `borderColorLeft`, `borderColorRight`, `borderColorTop` e `borderColorBottom`.(Si veda *Sezione 18.1.7.1, «Valori dei colori»*per i valori dei colori)
- `borderWidth` — La larghezza del bordo. I singoli lati dei bordi possono essere specificati usando `borderWidthLeft`, `borderWidthRight`, `borderWidthTop` e `borderWidthBottom`.

Utilizzo

```
<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>`	*Descrizione*
	
	Il numero della pagina corrente può essere collocato dentro un header o un footer usando il tag `p:pageNumber`. Il tag del numero della pagina può essereusato solamente nel contesto dell'header o footer e può essere usato solo una volta.
	
	Utilizzo
	
	`<p:footer borderWidthTop="1" borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`</p:footer>`

18.1.4. Capitoli e Sezioni

`<p:chapter>`
 	
`<p:section>`
	Descrizione
	
	Se il documento generato segue una struttura libro/articolo, i tag `p:chapter` e `p:section` possono essere usati per fornire la struttura

necessaria. Le sezioni possono essere usate soltanto dentro i capitoli, ma possono essere innestate con profondità arbitraria. La maggior parte dei visualizzatori PDF forniscono una facile navigazione tra i capitoli e le sezioni di un documento.

Attributi

- `alignment` — L'allineamento della sezione header/footer. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i valori dell'allineamento)
- `number` — Il numero del capitolo. Ogni capitolo dovrebbe avere un numero assegnato.
- `numberDepth` — La profondità della numerazione per le sezioni. Tutte le sezioni sono numerate relativamente ai loro capitoli/sezioni circostanti. La quarta sezione della prima sezione del capitolo terzo avrebbe sezione 3.1.4, se visualizzata con una profondità di default di tre. Per omettere il numero del capitolo, occorre usare una profondità di numerazione pari a 2. In questo caso, il numero della sezione visualizzato sarebbe 1.4.

Utilizzo

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
        title="Hello">

  <p:chapter number="1">
    <p:title
><p:paragraph
>Hello</p:paragraph
></p:title>
    <p:paragraph
>Hello #{user.name}!</p:paragraph>
  </p:chapter>

  <p:chapter number="2">
    <p:title
><p:paragraph
>Goodbye</p:paragraph
></p:title>
    <p:paragraph
>Goodbye #{user.name}.</p:paragraph>
  </p:chapter>
```

	</p:document >
`<p:header>`	*Descrizione* Ogni capitolo o sezione può contenere un `p:title`. Il titolo verrà mostrato vicino al numero del capitolo/sezione. Il corpo del titolo può contenere del semplice testo oppure un `p:paragraph`.

18.1.5. Liste

Le strutture di lista possono essere visualizzate usando i tag `p:list` e `p:listItem`. Le liste possono contenere sottoliste arbitrariamente innestate. Gli elementi di lista non possono essere usati fuori da una lista. Il seguente documento utilizza il tag `ui:repeat` per mostrare una lista di valori recuperata da un componente Seam.

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
          xmlns:ui="http://java.sun.com/jsf/facelets"
          title="Hello">
  <p:list style="numbered">
    <ui:repeat value="#{documents}" var="doc">
      <p:listItem
>#{doc.name}</p:listItem>
    </ui:repeat>
  </p:list>
</p:document
>
```

`<p:list>`	*Attributi* • `style` — Lo stile per l'ordinamento ed il contrassegno della lista. Un valore fra: `NUMBERED`, `LETTERED`, `GREEK`, `ROMAN`, `ZAPFDINGBATS`, `ZAPFDINGBATS_NUMBER`. Se non è fornito nessuno stile, gli elementi della lista sono contrassegnati. • `listSymbol` — Per le liste contrassegnate, specifica il simbolo del contrassegno. • `indent` — Il livello di indentazione della lista. • `lowerCase` — Per gli stili di lista che usano le lettere, indica se le lettere debbano essere minuscole.

	• `charNumber` — Per ZAPFDINGBATS, indica il codice del carattere per il carattere di contrassegno. • `numberType` — Per ZAPFDINGBATS_NUMBER, indica lo stile della numerazione. *Utilizzo* `<p:list style="numbered"> <ui:repeat value="#{documents}" var="doc"> <p:listItem >#{doc.name}</p:listItem> </ui:repeat> </p:list >`
`<p:listItem>`	*Descrizione* `p:listItem` supporta i seguenti attributi: *Attributi* • `alignment` — L'allineamento della sezione header/footer. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i valori dell'allineamento) • `alignment` — L'allineamento dell'elemento della lista. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i possibili valori) • `indentationLeft` — La quantità di indentazione sinistra. • `indentationRight` — La quantità di indentazione destra. • `listSymbol` — Sovrascrive il simbolo di default della lista per quest'elemento della lista. *Utilizzo* `...`

18.1.6. Tabelle

Le strutture della tabella possono essere create usando i tag `p:table` e `p:cell`. A differenza di molte strutture di tabella, non c'è alcuna dichiarazione esplicita di riga. Se una tabella ha 3 colonne,

allora le 3 celle formeranno automaticamente una riga. Le righe di header e footer possono essere dichiarate, e gli header ed i footer verranno ripetuti nel caso una struttura di tabella prosegua su più pagine.

`<p:table>`	*Descrizione* `p:table` supporta i seguenti attributi: *Attributi* • `columns` — Il numero di colonne (celle) che formano una riga della tabella. • `widths` — Le larghezze relative di ciascuna colonna. Dovrebbe esserci un valore per ciascuna colonna. Per esempio: larghezze="2 1 1" dovrebbe indicare che ci sono 3 colonne e la prima colonna è due volte la dimensione della seconda e della terza colonna. • `headerRows` — Il numero iniziale di righe che sono considerate righe dell'header e del footer e devono essere ripetute se la tabella prosegue per più pagine. • `footerRows` — Il numero iniziale di righe che sono considerate righe del footer. Questo valore è sottratto dal valore `headerRows`. Se il documento ha 2 righe che formano l'header ed una riga che forma il footer, `headerRows` deve essere impostato a 3 `footerRows` deve essere impostato a 1. • `widthPercentage` — La percentuale della larghezza della pagina che occupa la tabella. • `horizontalAlignment` — L'allineamento orizzontale della tabella. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i possibili valori) • `skipFirstHeader` • `runDirection` • `lockedWidth` • `splitRows` • `spacingBefore` — Lo spazio bianco da inserire prima dell'elemento. • `spacingAfter` — Lo spazio bianco da inserire dopo l'elemento. • `extendLastRow`

	• `headersInEvent` • `splitLate` • `keepTogether` *Utilizzo* `<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>`	*Descrizione* `p:cell` supporta i seguenti attributi. *Attributi* • `colspan` — Le celle possono proseguire per più di una colonna dichiarando un `colspan` maggiore di 1. Le tabelle non hanno la possibilità di proseguire su più righe. • `horizontalAlignment` — L'allineamento orizzontale della cella. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i possibili valori) • `verticalAlignment` — L'allineamento verticale della cella. (Si veda *Sezione 18.1.7.2, «Valori per l'allineamento»* per i possibili valori)

- `padding` — Il padding su un particolare lato può essere specificato usando `paddingLeft`, `paddingRight`, `paddingTop` e `paddingBottom`.
- `useBorderPadding`
- `leading`
- `multipliedLeading`
- `indent`
- `verticalAlignment`
- `extraParagraphSpace`
- `fixedHeight`
- `noWrap`
- `minimumHeight`
- `followingIndent`
- `rightIndent`
- `spaceCharRatio`
- `runDirection`
- `arabicOptions`
- `useAscender`
- `grayFill`
- `rotation`

Utilizzo

```
<p:cell
>...</p:cell
>
```

18.1.7. Costanti nei documenti

Questa sezione documenta alcune costanti condivise dagli attributi nei tag multipli.

18.1.7.1. Valori dei colori

I documenti Seam non supportano ancora un specifica per tutti i colori. Attualmente solo i colori con nome sono supportati. Questi sono: `white`, `gray`, `lightgray`, `darkgray`, `black`, `red`, `pink`, `yellow`, `green`, `magenta`, `cyan` e `blue`.

18.1.7.2. Valori per l'allineamento

Quando vengono usati i valori per l'allineamento, Seam PDF supporta i seguenti valori di allineamento orizzontali: `left`, `right`, `center`, `justify` e `justifyall`. I valori di allineamento verticali sono: `top`, `middle`, `bottom`, e `baseline`.

18.2. Grafici

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>`	*Descrizione* Displays a chart created in Java by a Seam component. *Attributi* • `chart` — The chart object to display. • `height` — L'altezza del grafico. • `width` — La larghezza del grafico. *Utilizzo* `<p:chart chart="#{mycomponent.chart}" width="500" height="500" />`
`<p:barchart>`	*Descrizione* Visualizza un grafico a barre. *Attributi* • `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` — Controlla se mostrare oppure no un bordo attorno a tutto il grafico.
- `borderPaint` — Il colore del bordo, se visibile;
- `borderBackgroundPaint` — Il colore di default per lo sfondo del grafico.
- `borderStroke` —
- `domainAxisLabel` — L'etichetta del testo per gli assi del dominio.
- `domainLabelPosition` — L'angolo delle etichette di categoria degli assi. I valori validi sono `STANDARD`, `UP_45`, `UP_90`, `DOWN_45` e `DOWN_90`. In alternativa, il valore può essere un valore positivo o negativo in radianti.
- `domainAxisPaint` — Il colore dell'etichetta degli assi di dominio.
- `domainGridlinesVisible`— Controlla se mostrare oppure no all'interno del grafico le griglie per gli assi del dominio.
- `domainGridlinePaint`— Il colore delle griglie di dominio, se visibili.
- `domainGridlineStroke` — Lo stile del tratteggio delle griglie di dominio, se visibili.
- `height` — L'altezza del grafico.
- `width` — La larghezza del grafico.
- `is3D` — Un valore booleano che indica di visualizzare il grafico in 3D invece che in 2D.
- `legend` — Un valore booleano che indica se oppure no includere la legenda nel grafico.
- `legendItemPaint`— Il colore di default delle etichette di testo nella legenda.
- `legendItemBackgoundPaint`— Il colore dello sfondo della legenda, se diverso dal colore di sfondo del grafico.
- `legendOutlinePaint`— Il colore del bordo attorno alla legenda.
- `orientation` — L'orientamento del disegno, o `vertical` (di default) o `horizontal`.

- `plotBackgroundPaint`— Il colore dello sfondo del disegno.
- `plotBackgroundAlpha`— Il livello alfa (trasparenza) dello sfondo del disegno. Dovrebbe essere un numero tra 0 (completamente trasparente) e 1 (completamente opaco).
- `plotForegroundAlpha`— Il livello alfa (trasparenza) del disegno. Dovrebbe essere un numero tra 0 (completamente trasparente) e 1 (completamente opaco).
- `plotOutlinePaint`— Il colore delle griglie di range, se visibili.
- `plotOutlineStroke` — Lo stile del tratteggio delle griglie di range, se visibili.
- `rangeAxisLabel` — L'etichetta di testo per l'asse di range.
- `rangeAxisPaint` — Il colore per l'etichetta dell'asse di range.
- `rangeGridlinesVisible`— Controlla se visualizzare nel grafico oppure no le griglie per l'asse di range.
- `rangeGridlinePaint`— Il colore delle griglie di range, se visibili.
- `rangeGridlineStroke` — Lo stile del tratteggio delle griglie di range, se visibili.
- `title` — Il titolo del grafico.
- `titlePaint`— Il colore del titolo del grafico.
- `titleBackgroundPaint`— Il colore di sfondo attorno al titolo del grafico.
- `width` — La larghezza del grafico.

Utilizzo

```
<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>`	*Descrizione* Mostra un grafico a linea. *Attributi* • `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` — Controlla se mostrare oppure no un bordo attorno a tutto il grafico. • `borderPaint` — Il colore del bordo, se visibile; • `borderBackgroundPaint` — Il colore di default per lo sfondo del grafico. • `borderStroke` — • `domainAxisLabel` — L'etichetta del testo per gli assi del dominio. • `domainLabelPosition` — L'angolo delle etichette di categoria degli assi. I valori validi sono `STANDARD`, `UP_45`, `UP_90`, `DOWN_45` e `DOWN_90`. In alternativa, il valore può essere un valore positivo o negativo in radianti. • `domainAxisPaint` — Il colore dell'etichetta degli assi di dominio. • `domainGridlinesVisible`— Controlla se mostrare oppure no all'interno del grafico le griglie per gli assi del dominio. • `domainGridlinePaint`— Il colore delle griglie di dominio, se visibili. • `domainGridlineStroke` — Lo stile del tratteggio delle griglie di dominio, se visibili. • `height` — L'altezza del grafico. • `width` — La larghezza del grafico. • `is3D` — Un valore booleano che indica di visualizzare il grafico in 3D invece che in 2D.

- `legend` — Un valore booleano che indica se oppure no includere la legenda nel grafico.
- `code` — Il valore da codificare nel barcode.
- `legendItemBackgoundPaint` — Il colore di sfondo della legenda, se diverso dal colore di sfondo del grafico.
- `code` — Il valore da codificare nel barcode.
- `orientation` — L'orientamento del disegno, o `vertical` (di default) o `horizontal`.
- `code` — Il valore da codificare nel barcode.
- `plotBackgroundAlpha`— Il livello alfa (trasparenza) dello sfondo del disegno. Dovrebbe essere un numero tra 0 (completamente trasparente) e 1 (completamente opaco).
- `plotForegroundAlpha`— Il livello alfa (trasparenza) del disegno. Dovrebbe essere un numero tra 0 (completamente trasparente) e 1 (completamente opaco).
- `plotOutlinePaint` — Il colore delle griglie di range, se visibili.
- `plotOutlineStroke` — Lo stile del tratteggio delle griglie di range, se visibili.
- `rangeAxisLabel` — L'etichetta di testo per l'asse di range.
- `rangeAxisPaint` — Il colore per l'etichetta dell'asse di range.
- `rangeGridlinesVisible` — Controlla se visualizzare nel grafico oppure no le griglie dell'asse di range.
- `rangeGridlinePaint` — Il colore delle griglie di range, se visibili.
- `rangeGridlineStroke` — Lo stile del tratteggio delle griglie di range, se visibili.
- `title` — Il titolo del grafico.
- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.
- `width` — La larghezza del grafico.

Utilizzo

```
<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>`	*Descrizione* Mostra un grafico. *Attributi* • `title` — Il titolo del grafico. • `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. • `code` — Il valore da codificare nel barcode. • `legend` — Un valore booleano che indica se oppure no includere la legenda nel grafico. Di default è true. • `is3D` — Un valore booleano che indica di visualizzare il grafico in 3D invece che in 2D. • `labelLinkMargin` — Il margine di collegamento per le etichette. • `labelLinkPaint` — Il motivo usato per le linee di collegamento dell'etichetta. • `code` — Il valore da codificare nel barcode. • `labelLinksVisible` — Un flag che controlla se vengono disegnati o no i link di etichetta. • `code` — Il valore da codificare nel barcode.

- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.
- `labelGap` — Il gap tra le etichette ed il disegno come percentuale della larghezza del disegno.
- `labelBackgroundPaint` — Il colore usato per disegnare lo sfondo delle etichette di sezione. Se null, lo sfondo non viene riempito.
- `code` — Il valore da codificare nel barcode.
- `circular` — Un valore booleano che indica se il grafico deve essere disegnato come cerchio. Se false, il grafico viene disegnato come ellisse. Di default è true.
- `direction` — La direzione in cui vengono disegnate le sezioni della torta. Un valore tra: `clockwise` o `anticlockwise`. Il default è `clockwise`.
- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.
- `sectionOutlinesVisible` — Indica se viene disegnata una outline per ciascuna sezione del disegno.
- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.
- `code` — Il valore da codificare nel barcode.

Utilizzo

```
<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>`	*Descrizione* I dati di categoria posso essere spezzati in serie. I tag delle serie vengono usati per categorizzare un set di dati con serie ed applicare lo stile all'intera serie. *Attributi* • `key` — Il nome delle serie. • `seriesPaint` — Il colore di ciascun item nelle serie. • `seriesOutlinePaint` — Il colore dell'outline per ciascun elemento nella serie. • `seriesOutlineStroke` — Il tratteggio usato per disegnare ciascun elemento nella serie. • `seriesVisible` — Un valore booleano che indicare se la serie debba essere visualizzata. • `seriesVisibleInLegend` — Un valore booleano che indicare se la serie debba essere elencata nella legenda. *Utilizzo*

```
<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>`	*Descrizione* Il tag dei dati descrive ciascun punto di dati da mostrare nel grafico. *Attributi* • `key` — Il nome dell'item di dati.

- `series` — I nomi delle serie, quando non è incorporato dentro un `<p:series>`.
- `value` — Il valore numerico dei dati.
- `explodedPercent` — Per i grafici a torta, indica come viene esploso un pezzo della torta.
- `sectionOutlinePaint` — Per i grafici a barre, il colore dell'outline di sezione.
- `sectionOutlineStroke` — Per i grafici a barre, il tipo di tratteggio dell'outline di sezione.
- `sectionPaint` — Per i grafici a barre, il colore della sezione.

Utilizzo

```
<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>`

Descrizione

Il componente del colore dichiara un colore oppure un gradiente che può venire referenziato quando si disegnano forme piene.

Attributi

- `color` — Il valore del colore. Per gradienti di colore, questo è il colore iniziale. *Sezione 18.1.7.1, «Valori dei colori»*
- `color2` — Per gradienti di colore, questo è il colore che termina il gradiente.
- `point` — Le coordinate dove inizia il colore gradiente.
- `point2` — Le coordinate dove finisce il colore gradiente.

Utilizzo

```
<p:color id="foo" color="#0ff00f"/>
<p:color id="bar" color="#ff00ff" color2="#00ff00"
```

	point="50 50" point2="300 300"/>
`<p:stroke>`	*Descrizione* Descrive un tratteggio usato per disegnare le linee in un grafico. *Attributi* • `width` — La larghezza del tratteggio. • `cap` — Il tipo di line cap. Valori validi sono `butt`, `round` e `square` • `join` — Il tipo di line join. Valori validi sono `miter`, `round` e `bevel` • `miterLimit` — Per join di tipo miter, questo valore è il limite della dimensione del join. • `dash` — Il valore del dash imposta il pattern dash da usare per disegnare la linea. Gli interi separati da spazio indicano la lunghezza di ciascun segmento in modo alternato disegnato e non disegnato. • `dashPhase` — La fase di dash indica l'offset nel pattern dash che con cui la linea dovrebbe essere disegnata. *Utilizzo* <p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" />

18.3. Codici a barre

Seam può usare iText per generare barcode in un'ampia varietà di formati. Questi barcode possono essere incorporati in un documento PDF o mostrati come immagine in una pagina web. Si noti che quando sono usati in immagini HTML, i barcode non possono attualmente mostrare testo al loro interno.

`<p:barCode>`	*Descrizione* Mostra un'immagine di barcode. *Attributi* • `type` — Un tipo di barcode supportato da iText. Valori validi includono: `EAN13`, `EAN8`, `UPCA`, `UPCE`, `SUPP2`, `SUPP5`, `POSTNET`, `PLANET`, `CODE128`, `CODE128_UCC`, `CODE128_RAW` e `CODABAR`. • `code` — Il valore da codificare nel barcode.

	• `code` — Il valore da codificare nel barcode. • `code` — Il valore da codificare nel barcode. • `rotDegrees` — Per i PDF il fattore di rotazione del barcode in gradi. • `barHeight` — L'altezza delle barre nel barCode. • `minBarWidth` — La larghezza minima della barra. • `barMultiplier` — Il moltiplicatore per le barre ampie o la distanza delle barre per codici `POSTNET` e `PLANET`. • `barColor` — Il colore per disegnare le barre. • `textColor` — Il colore del testo nel barcode. • `textSize` — La dimensione testo del barcode, se esiste. • `altText` — Il testo `alt` per i link HTML delle immagini. *Utilizzo* `<p:barCode type="code128" barHeight="80" textSize="20" code="(10)45566(17)040301" codeType="code128_ucc" altText="My BarCode" />`

18.4. Form da riempire

Se si ha un PDF complesso pregenerato con campi definiti (con nome), lo si può facilmente riempire con valori dell'applicazione e presentarlo all'utente.

`<p:form>`	*Descrizione* Definisce un modello di form da popolare *Attributi* • `URL` — Un URL che punta al file PDF da usarsi come template. Se il valore non ha la partedi protocollo (://), il file viene letto localmente. • `filename` — Il nome del file da usare per il file PDF generato.

	• `exportKey` — Colloca un file PDF generato in un oggetto DocumentData sotto la chiave specificata nel contesto evento. Se impostato, non avverrà alcun redirect.

`<p:field>`	*Descrizione* Unisce un nome campo al suo valore *Attributi* • `name` — Il nome del campo. • `value` — Il valore del campo. • `readOnly` — Il campo deve essere di sola lettura? Di default è true.

```
<p:form
      xmlns:p="http://jboss.com/products/seam/pdf"
      URL="http://localhost/Concept/form.pdf">
   <p:field name="person.name" value="Me, myself and I"/>
</p:form>
```

18.5. Componenti per i rendering Swing/AWT

Seam fornisce ora il supporto sperimentale per generare componenti Swing in un'immagine di PDF. Alcuni componenti Swing look and feels, in particolare quelli che usano widget nativi, non verranno correttamente generati.

`<p:swing>`	*Descrizione* Genera un componente Swing in un documento PDF. *Attributi* • `width` — La larghezza del componente da generare. • `code` — Il valore da codificare nel barcode. • `component` — Un'espressione il cui valore è un componente Swing o AWT. *Utilizzo*

```
<p:swing width="310" height="120" component="#{aButton}" />
```

18.6. Configurazione di iText

La generazione del documento funziona senza nessuna ulteriore configurazione. Comunque ci sono alcuni punti della configurazione necessari per applicazioni più serie.

L'implementazione di default serve i documenti PDF da un URL generico, `/seam-doc.seam`. Molti browser (e utenti) preferiscono vedere URL che contengono il vero nome del PDF, come `/myDocument.pdf`. Questa capacità richiede una configurazione. Per servire file PDF, tutte le risorse `*.pdf` devono essere mappate sul 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
>
```

L'opzione `use-extensions` nel componente document store completa la funzionalità istruendo il document store su come generare l'URL con l'estensione corretta del nome per i tipi di documenti da generare.

```
<components xmlns="http://jboss.com/products/seam/components"
   xmlns:document="http://jboss.com/products/seam/document"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
     http://jboss.com/products/seam/document http://jboss.com/products/seam/document-2.1.xsd
        http://jboss.com/products/seam/components http://jboss.com/products/seam/components-
2.1.xsd">
   <document:document-store use-extensions="true"/>
</components
```

```
>
```

Document store memorizza i documenti nello scope conversazione, ed i documenti scadranno quando termina la conversazione. A quel punto i riferimenti al documento saranno invalidati. Si può specificare la vista di default da mostrare quando un documento non esiste usando la proprietà `error-page` di `documentStore`.

```
<document:document-store use-extensions="true" error-page="/documentMissing.seam" />
```

18.7. Ulteriore documentazione

Per leggere ulteriore documentazione su iText, vedere:

- *iText Home Page* [http://www.lowagie.com/iText/]
- *iText in Action* [http://www.manning.com/lowagie/]

The Microsoft® Excel® spreadsheet application

Seam supporta inoltre la generazione di fogli elettronici the Microsoft® Excel® spreadsheet application tramite l'eccellente libreria *JExcelAPI* [http://jexcelapi.sourceforge.net/]. Il documento generato è compatibile con the Microsoft® Excel® spreadsheet application versione 95, 97, 2000, XP e 2003. Attualmente viene esposto un limitato set di funzionalità di questa libreria, ma lo scopo finale è riuscire a fare tutto ciò che la libreria consente. Si prega di fare riferimento alla documentazione di JExcelAPI per ulteriori informazioni, possibilità e limitazioni.

19.1. Supporto The Microsoft® Excel® spreadsheet application

The Microsoft® Excel® spreadsheet application `jboss-seam-excel.jar`. Questo JAR contiene i controlli the Microsoft® Excel® spreadsheet application JSF, che vengono impiegati per costruire le viste che possono generare il documento, ed il componente DocumentStore, che serve il documento generato all'utente. Per includere il supporto ad the Microsoft® Excel® spreadsheet application nella propria applicazione, si includa `jboss-seam-excel.jar` nella directory `WEB-INF/lib` assieme al file `jxl.jar`. Inoltre, occorre configurare il servlet DocumentStore in web.xml.

Il modulo The Microsoft® Excel® spreadsheet application di Seam richiede l'uso di Facelets come tecnologia per la vista. Inoltre richiede l'uso del pacchetto seam-ui.

Il progetto `examples/excel` contiene un esempio del supporto the Microsoft® Excel® spreadsheet application in azione. Mostra come eseguire il correttto impacchettamento per il deploy e mostra le funzionalità esposte.

La personalizzazione del modulo per supportare altri tipi di API the Microsoft® Excel® spreadsheet application è diventata molto semplice. Si implementi l'interfaccia `ExcelWorkbook`, e si registri in components.xml.

```
<excel:excelFactory>
  <property name="implementations">
    <key
>myExcelExporter</key>
    <value
>my.excel.exporter.ExcelExport</value>
  </property>
</excel:excelFactory
>
```

e si registri il namespace excel nei tag dei componenti con

```
xmlns:excel="http://jboss.com/products/seam/excel"
```

Poi si imposti il tipo UIWorkbook per `myExcelExporter` e verrà usato il proprio esportatore. Di default è "jxl", ma è stato aggiunto anche il supporto per CSV, usando il tipo "csv".

Si veda *Sezione 18.6, «Configurazione di iText»* per informazioni su come configurare il document servlet per servire i documenti con estensione .xls.

Se si incontrano problemi nell'accedere al file generato con IE (specialmente con https), ci si assicuri di non avere messo restrizioni nel browser (si veda *http://www.nwnetworks.com/iezones.htm/*), restrizioni di sicurezza in web.xml oppure una combinazione delle due.

19.2. Creazione di un semplice workbook

L'uso base del supporto ai fogli elettronici è molto semplice; viene usato come il familiare `<h:dataTable>` e si può associare una `List`, `Set`, `Map`, `Array` o `DataModel`.

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet>
    <e:cell column="0" row="0" value="Hello world!"/>
  </e:worksheet>
</e:workbook>
```

Non è molto utile, guardiamo quindi ai casi più comuni:

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet value="#{data}" var="item">
    <e:column>
      <e:cell value="#{item.value}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

In primo luogo si ha a livello più alto un elemento workbook, che serve come container e non ha attributi. L'elemento figlio worksheet ha due attributi; value="#{data}" è il binding EL ai dati e var="item" è il nome dell'elemento corrente. Innestato dentro worksheet c'è una singola colonna e dentro essa si vede la cella, che è l'associazione finale ai dati dell'elemento iterato.

Questo è ciò che serve sapere per cominciare a mettere dati nei fogli elettronici!

19.3. Workbooks

I workbook sono i padri di livello più alto dei worksheet e dei link ai fogli di stile.

`<e:workbook>`	*Attributi* • `type` — Definisce quale module esporto usare. Il valore è una string e può essere o "jxl" o "csv". Il default è "jxl". • `description` — La descrizione del link. Il valore è una stringa. • `arrayGrowSize` — L'ammontare di memoria con cui incrementare la memoria allocata per la memorizzazione dei dati di workbook. Per processi che leggono molti workbook piccoli dentro un WAS, può essere necessario ridurre la dimensione di default. Il valore di default è 1 megabyte. Il valore è un numero (byte). • `topMargin` — Il margine superiore. Il valore è un numero (pollici). • `description` — La descrizione del link. Il valore è una stringa. • `characterSet` — Il set di caratteri. Questo viene usato solo quando viene letto il foglio elettronico, e non ha alcun effetto quando viene scritto. Il valore è una stringa (codifica del set di caratteri). • `description` — La descrizione del link. Il valore è una stringa. • `excelDisplayLanguage` — Il linguaggio in cui viene mostrato il file generato. Il valore è una stringa (codice a due caratteri ISO 3166 del paese). • `description` — La descrizione del link. Il valore è una stringa. • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). • `description` — La descrizione del link. Il valore è una stringa. • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). • `initialFileSize` — Ammontare iniziale di memoria allocata per memorizzare i dati del workbook quando viene letto un foglio elettronico. Per processi che leggono molti workbook piccoli dentro

un WAS può essere necessario ridurre la dimensione di default. Il valore di default è 5 megabytes. Il valore è un numero (byte).

- `locale` — Il locale usato da JExcelApi per generare il foglio elettronico. Impostare questo valore non ha effetti sulla lingua o la regione del file excel generato. Il valore è una stringa.
- `mergedCellCheckingDisabled` — Se il controllo della cella unita debba essere disabilitato. Il valore è un booleano.
- `description` — La descrizione del link. Il valore è una stringa.
- `propertySets` — I set di proprietà (come le macro) devono essere abilitati per essere copiati assieme al workbook? Lasciando abilitato questa caratteristica, il processo JXL userà più memoria. Il valore è un booleano.
- `rationalization` — I formati delle celle devono essere razionalizzati prima di scrivere il foglio? Il valore è un booleano. Il default è true.
- `supressWarnings` — Gli avvisi devono essere soppressi? A causa di un cambiamento nella versione 2.4 del logging, ora il comportamento degli avvisi è impostato a livello di JVM (dipende dal tipo di logger usato). Il valore è un booleano.
- `temporaryFileDuringWriteDirectory` — Usato in congiunzione con `useTemporaryFileDuringWrite` per impostare la directory target per i file temporanei. Il valore può essere NULL, nel qual caso viene usata la directory temporanea di default del sistema. Il valore è una stringa (la directory in cui scrivere i file temporanei).
- `useTemporaryFileDuringWrite` — Se un file temporaneo debba essere usato durante la generazione del workbook. Se non impostato, il workbook avverrà interamente in memoria. Impostando questo flag si valuti il trade-off tra l'uso della memoria e la performance. Il valore è un booleano.
- `bottomMargin` — Il margine inferiore. Il valore è un numero (pollici).
- `filename` — Il nome del file da usare per il download. Il valore è una stringa. Si noti che se si mappa il DocumentServlet su qualche pattern, questa estensione del file deve corrispondere.
- `exportKey` — Una chiave sotto cui memorizzare i dati di risultato in un oggetto DocumentData nello scope evento. Se usato, non c'è redirezione.

	Elementi figli • `<e:link/>` — Zero o più link a fogli di stile (vedere *Sezione 19.14.1, «Link ai fogli di stile»*). • `<e:worksheet/>` — Zero or più fogli di lavoro (see *Sezione 19.4, «Worksheets»*). *Facets* • `nessuno`

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

definisce un workbook con un foglio di lavoro ed un saluto in A1

19.4. Worksheets

I worksheet sono figli dei workbook e padri delle colonne e dei comandi worksheet. Possono anche contenere celle, formule, immagini e collegamenti esplicitamente collocati. Sono le pagine che creano il workbook.

`<e:worksheet>`	• `value` — Espressione-EL sui dati retrostanti. Il valore è una stringa. Il target di quest'espressione è esaminato per un Iterable. Si noti che se il target è una Map, l'iterazione viene fatta su Map.Entry entrySet(), e quindi occorre usare un .key o un .value per individuare i riferimenti. • `var` — Il nome della variabile della riga iteratore corrente che può successivamente essere referenziata negli attributi di valore della cella. Il valore è una stringa. • `name` — Il nome del worksheet. Il valore è una stringa. Il default è Sheet# dove # è l'indice del worksheet. Se esiste il nome del worksheet dato, tale foglio viene selezionato. Questo può essere usato per unire diversi set di dati in un singolo worksheet, solamente definendo lo stesso nome per questi (usando `startRow` e `startCol` per assicurarsi che non occupino lo stesso spazio).

- `startRow` — Definisce la riga di inizio dei dati. Il valore è un numero. Usato per collocare i dati in altri posti rispetto all'angolo in alto a sinistra (è utile in particolare se si hanno set di dati multipli per un singolo worksheet). Il default è 0.
- `startColumn` — Definisce la colonna di inizio dei dati. Il valore è un numero. Usato per collocare i dati in altri posti rispetto all'angolo in alto a sinistra (è utile in particolare se si hanno set di dati multipli per un singolo worksheet). Il default è 0.
- `automaticFormulaCalculation` — Le formule devono essere automaricamente ricalcolate? Il valore è un booleano.
- `bottomMargin` — Il margine inferiore. Il valore è un numero (pollici).
- `topMargin` — Il margine superiore. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `description` — La descrizione del link. Il valore è una stringa.
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `topMargin` — Il margine superiore. Il valore è un numero (pollici).
- `description` — La descrizione del link. Il valore è una stringa.
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `normalMagnification` — Il fattore di ingrandimento normale (non zoom o fattore di scala). Il valore è un numero (percentuale).
- `description` — La descrizione del link. Il valore è una stringa.

- `pageBreakPreviewMagnification` — Il fattore di ingrandimento dell'anteprima per l'interruzione di pagina (non fattori di zoom o di scala). Il valore è un numero (percentuale).
- `topMargin` — Il margine superiore. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `paperSize` — La dimensione dellapagina da usare quando si stampa questo foglio. Il valore è una stringa che può essere un valore tra "a4", "a3", "letter", "legal" ecc. (Si veda *jxl.format.PaperSize* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/PaperSize.html]).
- `description` — La descrizione del link. Il valore è una stringa.
- `description` — La descrizione del link. Il valore è una stringa.
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `topMargin` — Il margine superiore. Il valore è un numero (pollici).
- `bottomMargin` — Il margine inferiore. Il valore è un numero (pollici).
- `recalculateFormulasBeforeSave` — Le formule devono essere ricalcolate quando viene salvato il foglio? Il valore è un boolean. Il default è false.
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `description` — La descrizione del link. Il valore è una stringa.
- `description` — La descrizione del link. Il valore è una stringa.
- `topMargin` — Il margine superiore. Il valore è un numero (pollici).
- `leftMargin` — Il margine sinistro. Il valore è un numero (pollici).
- `rightMargin` — Il margine destro. Il valore è un numero (pollici).
- `description` — La descrizione del link. Il valore è una stringa.
- `zoomFactor` — Fattore di zoom. Da non confondere con il fattore di zoom (che è relazionato alla vista dello schermo) con il fattore di scala (che si riferisce al fattore di scala in fase di stampa). Il valore è un numero (percentuale).

Elementi figli

	• `<e:printArea/>` — Zero o più definizioni di area di stampa (Si veda *Sezione 19.11, «Stampa di aree e titoli»*). • `<e:printTitle/>` — Zero o più definizioni di titolo di stampa(Si veda *Sezione 19.11, «Stampa di aree e titoli»*). • `<e:headerFooter/>` — Zero o più definizioni di header/footer (Si veda *Sezione 19.10, «Intestazioni e pié di pagina»*). • Zero o più comandi di worksheet (si veda *Sezione 19.12, «Comandi per i fogli di lavoro (worksheet)»*). *Facets* • `header`— I contenuti che verranno messi in cima al blocco di dati, sopra le intestazioni della colonna (se presenti). • `footer`— I contenuti che verranno messi in fondo al blocco di dati, sotto i pié di pagina della colonna (se presenti).

```
<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>
```

definisce un worksheet con nome "foo", che inizia in B2.

19.5. Colonne

Le colonne sono figle dei worksheet e padri delle celle, immagini, formule e collegamenti. Sono la struttura che controlla l'iterazione dei dati del worksheet. Si veda *Sezione 19.14.5, «Impostazioni colonna»* per la formattazione.

`<e:column>`	*Attributi*

- `nessuno`

Elementi figli

- `<e:cell/>` — Zero o più celle (vedere *Sezione 19.6, «Celle»*).
- `<e:formula/>` — Zero o più formule (vedere *Sezione 19.7, «Formule»*).
- `<e:image/>` — Zero o più immagini (vedere *Sezione 19.8, «Immagini»*).
- `<e:hyperLink/>` — Zero o più hyperlink (vedere *Sezione 19.9, «Hyperlinks»*).

Facets

- `header` — Questo facet può contenere un `<e:cell>`, `<e:formula>`, `<e:image>` o `<e:hyperLink>` che verrà usato come intestazione per la colonna.
- `footer` — Questo facet può contenere un `<e:cell>`, `<e:formula>`, `<e:image>` o `<e:hyperLink>` che verrà usato come pié di pagina per la colonna.

```
<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>
```

definisce una colonna con un intestazione ed un output iterato.

19.6. Celle

Le celle sono innestate dentro le colonne (per l'iterazione) o dentro i worksheet (per il collocamento diretto usando gli attributi `column` e `row`) e sono responsabili dell'output del valore (solitamente tramite un'espressione EL che coinvolge l'attributo `var` della datatable. Vedere *???*

`<e:cell>`	*Attributi* • `column` — La colonna dove mettere la cella. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da 0. • `row` — La riga dove mettere la cella. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da 0. • `value` — Il valore da mostrare. Solitamente un'espressione-EL che fa riferimento all'attributo della variabile della datatable che lo contiene. Il valore è una stringa. • `description` — La descrizione del link. Il valore è una stringa. • `rightMargin` — Il margine destro. Il valore è un numero (pollici). • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). *Elementi figli* • Zero o più condizioni di validazione (si veda *Sezione 19.6.1, «Validazione»*). *Facets* • `nessuno`

```
        <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
```

```
>
```

definisce una colonna con un intestazione ed un output iterato.

19.6.1. Validazione

Le validazioni vengono innestate dentro celle o formule. Esse aggiungono vincoli ai dati di cella.

`<e:numericValidation>`	*Attributi*
	
	• `description` — La descrizione del link. Il valore è una stringa.
	• `description` — La descrizione del link. Il valore è una stringa.
	• `description` — La descrizione del link. Il valore è una stringa.
	  • "equal" - richiede che il valore della cella corrisponda a quello definito nell'attributo value
	  • "greater_equal" - richiede che il valore della cella sia maggiore o uguale al valore definito nell'attributo value
	  • "less_equal" - richiede che il valore della cella sia minore o uguale al valore definito nell'attributo value
	  • "less_than" - richiede che il valore della cella sia minore del valore definito nell'attributo value
	  • "not_equal" - richiede che il valore della cella non corrisponda al valore definito nell'attributo value
	  • "between" - richiede che il valore della cella sia compreso tra i valori definiti negli attributi value e value2
	  • "not_between" - richiede che il valore della cella non sia compreso tra i valori definiti negli attributi value e value2
	
	Elementi figli
	
	• `nessuno`
	
	Facets
	
	• `nessuno`

```
        <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
>
```

aggiunge la validazione numerica alla cella specificando che il valore deve essere tra 4 e 18.

`<e:rangeValidation>`	*Attributi* • `description` — La descrizione del link. Il valore è una stringa. • `description` — La descrizione del link. Il valore è una stringa. • `description` — La descrizione del link. Il valore è una stringa. • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <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
>
```

aggiunge la validazione ad una cella specificando che il valore deve essere in valori specificati nel range A1:A10.

`<e:listValidation>`	*Attributi* • `nessuno` *Elementi figli* • `Zero o più elementi di validazione lista.` *Facets* • `nessuno`

e:listValidation è solo un container per mantenere diversi tag e:listValidationItem.

`<e:listValidationItem>`	*Attributi* • `value` — Un valore da validare. *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <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
>
```

aggiunge la valodazione ad una cella specificando che il valore deve essere "manager" o "employee".

19.6.2. Maschere per il formato

Le maschere di formato sono definite nell'attributo maschera in celle o formule. Ci sono due tipi di maschere di formato, uno per numeri e uno per date.

19.6.2.1. Maschere per il numero

Incontrando una maschera di formato, in primo luogo è contrassegnata se è in una form interna, es. "format1", "accounting_float" e così via (si veda *jxl.write.NumberFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/NumberFormats.html]).

se la maschera non è nella lista, viene trattata come una maschera personalizzata (si veda *java.text.DecimalFormat* [http://java.sun.com/javase/6/docs/api/java/text/DecimalFormat.html]). Es. "0.00" e automaticamente convertita nella corrispondenza più vicina.

19.6.2.2. Maschere per la data

Incontrando una maschera di formato, in primo luogo è contrassegnata se è in una form interna, es. "format1", "format2" e così via (si veda *jxl.write.DecimalFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/DecimalFormats.html]).

se la maschera non è nella lista, viene trattata come maschera personalizzata (si veda *java.text.DateFormat* [http://java.sun.com/javase/6/docs/api/java/text/DateFormat.html]), es. "dd.MM.yyyy" viene automaticamente convertita nella corrispondenza più vicina.

19.7. Formule

Le formule sono innestate nelle colonne (per iterazione) o dentro i worksheet (per collocazione diretta usando gli attributi `column` e `row`) e aggiungono calcoli o funzioni ai range di celle. Sono essenzialmente celle, si veda *Sezione 19.6, «Celle»* per gli attributi disponibili. Si noti che possono impiegare template ed avere definizioni proprie di font, ecc. come le normali celle.

La formula della cella è collocata nell'attributo `value` come una normale annotazione the Microsoft® Excel® spreadsheet application. Si noti che quando si fanno formule con riferimenti ad altri fogli, i worksheet devo esistere prima di eseguire riferimenti ad essa. Il valore è una stringa.

```
        <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
>
```

definisce una formula in B2 che somma le celle A1 nei worksheet FooSheet e BarSheet.

19.8. Immagini

Le immagini sono innestate dentro le colonne (per iterazione) o dentro i worksheet (per collocazione diretta usando gli attributi `startColumn/startRow` e `rowSpan/columnSpan`). Gli span sono opzionali e se omessi, l'immagine verrà inserita senza ridimensionamento.

`<e:image>`	*Attributi* • `startColumn` — La colonna iniziale dell'immagine. Il default è il contatore interno. Il valore è un numero. Si noti che il valore inizia da zero. • `startRow` — La riga iniziale dell'immagine. Il default è il contatore interno. Il valore è un numero. Si noti che il valore inizia da zero. • `columnSpan` — Lo span di colonna dell'immagine. Il default è quello risultante nella larghezza di default dell'immagine. Il default è un float. • `rowSpan` — Lo span di colonna dell'immagine. Il default è quello risultante nella altezza di default dell'immagine. Il default è un float. • `description` — La descrizione del link. Il valore è una stringa.

	Elementi figli • `nessuno` *Facets* • `nessuno`

```
        <e:workbook>
          <e:worksheet>
            <e:image startRow="0" startColumn="0" rowSpan="4"
              columnSpan="4" URI="http://foo.org/logo.jpg"/>
          </e:worksheet>
        </e:workbook
>
```

definisce un'immagine in A1:A5 basato sui dati in questione

19.9. Hyperlinks

Gli hyperlink sono innestati dentro le colonne (per iterazione) o dentro i worksheet (per collocazione diretta usando gli attributi `startColumn/startRow` e `endColumn/endRow`). Aggiungono agli URI una navigazione tramite link.

`<e:hyperlink>`	*Attributi* • `startColumn` — La colonna iniziale del hyperlink. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da zero. • `startRow` — La riga iniziale del hyperlink. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da zero. • `endColumn` — La colonna finale del hyperlink. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da zero. • `endRow` — La riga finale del hyperlink. Il default è un contatore interno. Il valore è un numero. Si noti che il valore parte da zero. • `description` — La descrizione del link. Il valore è una stringa.

	• `description` — La descrizione del link. Il valore è una stringa. *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <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
>
```

definisce un hyperlink che punta a SFWK nell'area A1:E5

19.10. Intestazioni e pié di pagina

Intestazioni e pié di pagina sono figli dei fogli di lavoro e contengono facet che a loro volta contengono una stringa con i comandi da analizzare.

`<e:header>`	*Attributi* • `nessuno` *Elementi figli* • `nessuno` *Facets* • `left` — I contenuti della parte sinistra di intestazione/pié di pagina. • `center` — i contenuti della parte centrale di intestazione/pié di pagina.

	• `right` — i contenuti della parte destra di intestazione/pié di pagina.

`<e:footer>`	*Attributi* • `nessuno` *Elementi figli* • `nessuno` *Facets* • `left` — I contenuti della parte sinistra di intestazione/pié di pagina. • `center` — i contenuti della parte centrale di intestazione/pié di pagina. • `right` — i contenuti della parte destra di intestazione/pié di pagina.

Il contenuto dei facets è una stringa che può contenere vari comandi delimitati da # come segue:

<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 >	Inserisce la data corrente
#page_number#	Inserisce il numero della pagina corrente
#time#	Inserisce l'orario corrente
#total_pages#	Inserisce il conteggio totale della pagina
#worksheet_name#	Inserisce il nome del worksheet
#workbook_name#	Inserisce il nome del workbook
#bold#	Attiva il font in grassetto, usare un altro #bold# per disattivarlo
#italics#	Attiva il font in corsivo, usare un altro #italic# per disattivarlo
#underline#	Attiva la sottolineatura, usare un altro #underline# per disattivarla

#double_underline#	Attiva la doppia sottolineatura, usare un altro #double_underline# per disattivarla
#outline#	Attiva il font outline, usare un altro #outline# per disattivarlo
#shadow#	Attiva il font ombreggiato, usare un altro #shadow# per disattivarlo
#strikethrough#	Attiva il font barrato, usare un altro #strikethrough# per disattivarlo
#subscript#	Attiva il font subscripted, usare un altro #subscript# per disattivarlo
#superscript#	Attiva il font superscript, usare un altro #superscript# per disattivarlo
#font_name#	Imposta il nome dei font, si usa come #font_name=Verdana#
#font_size#	Imposta la dimensione dei font, si usa come #font_size=12#

```
        <e:workbook>
          <e:worksheet
>
            <e:header>
              <f:facet name="left">
                This document was made on #date# and has #total_pages# pages
              </f:facet>
              <f:facet name="right">
                #time#
              </f:facet>
            </e:header>
          <e:worksheet>
        </e:workbook>
```

19.11. Stampa di aree e titoli

Stampa le aree ed i titoli figli dei worksheet e dei templatee fornisce...stampa aree e titoli.

`<e:printArea>`	*Attributi* • `firstColumn` — La colonna dell'angolo in alto a sinistra dell'area. Il parametro è un numero. Si noti che il valore parte da zero. • `firstRow` — La riga dell'angolo in alto a sinistra dell'area. Il parametro è un numero. Si noti che il valore parte da zero. • `lastColumn` — La colonna dell'angolo in basso a destra dell'area. Il parametro è un numero. Si noti che il valore parte da zero.

	• `lastRow` — La riga dell'angolo in basso a destra dell'area. Il parametro è un numero. Si noti che il valore parte da zero. *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
<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>
```

definisce un titolo di stampa tra A1:A10 ed un'area di stampa tra B2:J10.

19.12. Comandi per i fogli di lavoro (worksheet)

I comandi per i fogli di lavoro sono figli dei workbook e vengono solitamente eseguiti solo una volta.

19.12.1. Raggruppamento

Fornisce un raggruppamento di colonne e righe.

`<e:groupRows>`	*Attributi* • `topMargin` — Il margine superiore. Il valore è un numero (pollici). • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). • `description` — La descrizione del link. Il valore è una stringa. *Elementi figli*

	• `nessuno` *Facets* • `nessuno`
`<e:groupColumns>`	*Attributi* • `topMargin` — Il margine superiore. Il valore è un numero (pollici). • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). • `description` — La descrizione del link. Il valore è una stringa. *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <e:workbook>
          <e:worksheet
>
            <e:groupRows startRow="4" endRow="9" collapse="true"/>
            <e:groupColumns startColumn="0" endColumn="9" collapse="false"/>
          </e:worksheet>
        </e:workbook>
```

raggruppa dalla riga 5 alla 10 e dalla colonna 5 alla 10 in modo che le righe siano inizialmente collassate (ma non le colonne).

19.12.2. Interruzioni di pagina

Fornisce interruzioni di pagina

`<e:rowPageBreak>`	*Attributi* • `topMargin` — Il margine superiore. Il valore è un numero (pollici).

	Elementi figli • `nessuno` *Facets* • `nessuno`

```
<e:workbook>
  <e:worksheet
>
    <e:rowPageBreak row="4"/>
  </e:worksheet>
</e:workbook
>
```

interrompe la pagina alla riga 5.

19.12.3. Fusione (merge)

Fornisce l'unione delle celle

`<e:mergeCells>`	*Attributi* • `topMargin` — Il margine superiore. Il valore è un numero (pollici). • `topMargin` — Il margine superiore. Il valore è un numero (pollici). • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). • `leftMargin` — Il margine sinistro. Il valore è un numero (pollici). *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <e:workbook>
          <e:worksheet>
            <e:mergeCells startRow="0" startColumn="0" endRow="9" endColumn="9"/>
          </e:worksheet>
        </e:workbook
>
```

unisce le celle nel range A1:J10

19.13. Esportatore di datatable

Se si preferisce esportare una datatable JSF esistente invece di scrivere un apposito documento XHTML, si può eseguire il componente `org.jboss.seam.excel.excelExporter.export`, passando l'id della datatable come parametro Seam EL. Si presuma di avere una datatable

```
      <h:form id="theForm">
        <h:dataTable id="theDataTable" value="#{personList.personList}"
          var="person">
          ...
        </h:dataTable>
      </h:form>
```

che si vuole visualizzare come foglio Microsoft® Excel®. Si collochi un

```
      <h:commandLink
        value="Export"
        action="#{excelExporter.export('theForm:theDataTable')}"
      />
```

nella form e si è già finito. Si può certamente eseguire l'exporter con un pulsante, s: link oppure con un altro metodo preferito. Ci sono anche piani per un tag dedicato all'export che possa essere messo nel tag della datatable per non dover più fare riferimento all'id della datatable.

Si veda *Sezione 19.14, «Font e layout»* per la formattazione.

19.14. Font e layout

Il controllo dell'aspetto dell'output è fatto con una combinazione di attributi stile-CSS e di attributi tag. I più comuni (font, bordi, background, ecc.) sono CSS ed alcune impostazioni generali sono negli attributi tag.

Gli attributi CSS vanno in cascata dal padre ai figli e da un tag si scende in cascata attraverso classi CSS referenziate negli attributi `styleClass` ed infine lungo gli attributi CSS definiti nell'attributo `style`. Si può collocarli ovunque, ma ad esempio impostare la larghezza di colonna in una cella innestata dentro tale colonna ha poco senso.

19.14.1. Link ai fogli di stile

Gli stylesheet esterni sono referenziati con il tag e:link. Vengono messi come figli del workbook.

`<e:link>`	*Attributi* • `URL` — l'URL al foglio di stile *Elementi figli* • `nessuno` *Facets* • `nessuno`

```
        <e:workbook>
          <e:link URL="/css/excel.css"/>
        </e:workbook
>
```

Fa riferimento allo stylesheet che può essere trovato in /css/excel.css

19.14.2. Font

Questo gruppo di attributi XLS-CSS definisce un font ed i suoi attributi

xls-font-family	Il nome del font. Assicurarsi che sia supportato del proprio sistema.
xls-font-size	La dimensione del font. Si usi un numero intero.
xls-font-color	Il colore dei font (si veda *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-font-bold	Il font deve essere in grassetto? I valori validi sono "true" and "false"
xls-font-italic	Il font deve essere in corsivo? I valori validi sono "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	Lo stile sottolineato del font (si veda *jxl.format.UnderlineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/UnderlineStyle.html]).
xls-font-struck-out	Il font deve essere barrato? I valori validi sono "true" e "false"
xls-font	Una notazione breve per impostare tutti i valori. Si collochi il nome del font in fondo e si usino i tick mark per i font separati da uno spazio, es. 'Times New Roman'. Si usi "italic", "bold" e "struckout". Esempio style="xls-font: red bold italic 22 Verdana"

19.14.3. Bordi

Questo gruppo di attributi XLS-CSS definisce i bordi della cella

xls-border-left-color	Il colore del bordo del lato sinistro della cella (si veda *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-left-line-style	Lo stile di linea del bordo del lato sinistro della cella (si veda *jxl.format.LineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/LineStyle.html]).
xls-border-left	Una notazione breve per impostare lo stile della linea ed il colore del lato sinistro della cella, es. style="xls-border-left: thick red"
xls-border-top-color	Il colore del bordo del lato alto della cella (si veda *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-top-line-style	Lo stile di linea del bordo del lato alto della cella (si veda *jxl.format.LineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/LineStyle.html]).

xls-border-top	Una notazione breve per impostare lo stile della linea ed il colore del lato alto della cella, es. style="xls-border-top: red thick"
xls-border-right-color	Il colore del bordo del lato destro della cella (si veda *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-right-line-style	Lo stile di linea del bordo del lato destro della cella (si veda *jxl.format.LineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/LineStyle.html]).
xls-border-right	Una notazione breve per impostare lo stile della linea ed il colore del lato destro della cella, es. style="xls-border-right: thick red"
xls-border-bottom-color	Il colore del bordo del lato basso della cella (si veda *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-bottom-line-style	Lo stile di linea del bordo del lato basso della cella (si veda *jxl.format.LineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/LineStyle.html]).
xls-border-bottom	Una notazione breve per impostare lo stile della linea ed il colore del lato basso della cella, es. style="xls-border-bottom: thick red"
xls-border	Una notazione breve per impostare lo stile della linea ed il colore per tutti i lati della cella, es. style="xls-border: thick red"

19.14.4. Background

Questo gruppo di attributi XLS-CSS definisce il background della cella

xls-background-color	Il colore del background (si veda *jxl.format.LineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/LineStyle.html]).
xls-background-pattern	Il pattern del background (si veda *jxl.format.Pattern* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Pattern.html]).
xls-background	Una notazione breve per impostare il pattern ed il colore del background. Vedere sopra per le regole.

19.14.5. Impostazioni colonna

Questo gruppo di attributi XLS-CSS definisce le larghezze di colonna, ecc.

xls-column-width	La larghezza della colonna. Si usino valori ampi (~5000) per cominciare. Usata da e:column nella modalità xhtml.
xls-column-widths	La larghezza della colonna. Si usino valori ampi (~5000) per cominciare. Usata dall'excel exporter, collocato nell'attributo style della datatable. Si usino valori numerici o * per bypassare una colonna.

	Esempio style="xls-column-widths: 5000, 5000, *, 10000"
xls-column-autosize	Deve essere fatto un tentativo per autodimensionare la colonna? I valori validi sono "true" and "false".
xls-column-hidden	La colonna deve essere nascosta? I valori validi sono "true" e "false".
xls-column-export	La colonna deve essere mostrata nell'esportazione? I valori validi sono "true" e "false". Il valore di default è "true".

19.14.6. Impostazioni cella

Questo gruppo di attributi XLS-CSS definisce le proprietà della cella

xls-alignment	L'allineamento del valore della cella (si veda *jxl.format.Alignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Alignment.html]).
xls-force-type	Il tipo forzato dei dati della cella. Il valore è una stringa che può essere una tra "general", "number", "text", "date", "formula" o "bool". Il tipo è automaticamente individuato e quindi quest'attributo si usa raramente.
xls-format-mask	La maschera di formato della cella, si veda *Sezione 19.6.2, «Maschere per il formato»*
xls-indentation	L'indentazione del valore della cella. Il valore è numerico.
xls-locked	Se la cella debba essere bloccata. Da usare con il livello workbook bloccato. I valori validi sono "true" e "false".
xls-orientation	L'orientamento del valore della cella (si veda *jxl.format.Orientation* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Orientation.html]).
xls-vertical-alignment	L'allineamento verticale del valore della cella (si veda *jxl.format.VerticalAlignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/VerticalAlignment.html]).
xls-shrink-to-fit	I valori della cella devono adattarsi alla dimensione? I valori validi sono "true" e "false".
xls-wrap	La cella deve contenere nuove linee? I valori validi sono "true" e "false".

19.14.7. L'exporter delle datatable

L'exporter delle datatable impiega gli stessi attributi xls-css come documento xhtml con l'eccezione che le larghezze di colonna vengono definite con l'attributo `xls-column-widths` nella datatable (poiché UIColumn non supporta gli attributi style o styleClass).

19.14.8. Esempi di layout

TODO

19.14.9. Limitazioni

Nell'attuale versione ci sono alcune note limitazioni riguardanti il supporto CSS.

- Usando i documenti xhtml, gli stylesheet devono fare riferimento tramite il tag `<e:link> tag`
- Usando l'exporter delle datatable, i CSS devono essere inseriti negli attributi style, gli stylesheet esterni non sono supportati

19.15. Internazionalizzazione

Ci sono solo due chiavi di resource bundle usate, entrambe per il formato di data invalido ed entrambi prendono un parametro (il valore chiave)

- `org.jboss.seam.excel.not_a_number` — Quando un valore supposto essere un numero non può essere trattato come tale
- `org.jboss.seam.excel.not_a_date` — Quando un valore supposto essere una data non può essere trattato come tale

19.16. Link ed ulteriore documentazione

Il nucleo delle funzionalità the Microsoft® Excel® spreadsheet application è basato sull'eccellente libreria JExcelAPI che può essere trovata in *http://jexcelapi.sourceforge.net/* [http://jexcelapi.sourceforge.net] e la maggior parte delle funzionalità e le possibili limitazioni sono ereditate da qua.

Se si usano il forum o la mailing list, si prega di ricordare che non questi non possono fornire informazioni su come Seam impiega la loro libreria, ogni problema dovrebbe essere riportato in JBoss Seam JIRA sotto il modulo "excel".

Supporto RSS

Con Seam è semplice gestire i feed RSS tramite la libreria *YARFRAW* [http://yarfraw.sourceforge.net/]. Il supporto RSS in questa versione è da considerarsi nello stato di anteprima.

20.1. Installazione

Per abilitare il supporto RSS includere `jboss-seam-rss.jar` nella cartella `WEB-INF/lib` dell'applicazione. La libreria RSS ha anche alcune dipendenze che devono essere posizionate nella stessa cartella. Vedi *???* per la lista delle librerie da includere.

Il supporto RSS in Seam richiede di utilizzare Facelets come tecnologia per la vista.

20.2. Generare dei feed

Il progetto `examples/rss` contiene un esempio del supporto RSS in azione. Mostra il modo appropriato per posizionare le librerie e illustra le funzionalità esposte.

Un feed è una pagina xhtml che consiste in un feed e una lista di elementi annidati.

```
<r:feed
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:r="http://jboss.com/products/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>
```

20.3. I feed

I feed sono delle entità di primo livello che descrivono le proprietà della sorgente di informazioni. Possono contenere uno o più elementi.

`<r:feed>`	*Attributi* • `uid` — Un identificativo unico opzionale. Il valore è una stringa. • `title` — Il titolo del feed. Il valore è una stringa. • `subtitle` — Il sottotitolo del feed. Il valore è una stringa. • `updated` — Quando è stato aggiornato il feed? Il valore è una data. • `link` — Il link alla fonte dell'informazione. Il valore è una stringa. • `feedFormat` — Il formato del feed. Il valore è una stringa con default ATOM1. I valori ammessi sono RSS10, RSS20, ATOM03 e ATOM10. *Elementi contenuti* • `Zero o più elementi` *Facets* • `nessuna`

20.4. Elementi

Gli elementi rappresentano i "titoli" del feed.

`<r:feed>`	*Attributi* • `uid` — Un identificativo unico opzionale. Il valore è una stringa. • `title` — Il titolo dell'elemento. Il valore è una stringa. • `link` — Un link all'articolo. Il valore è una stringa. • `author` — L'autore dell'articolo. Il valore è una stringa. • `summary` — Il corpo dell'articolo. Il valore è una stringa.

	• `textFormat` — Il formato del corpo e del titolo dell'articolo. Il valore è una stringa e i valori ammessi sono "text" e "html". Il valore di default è "html". • `published` — Quando è stato pubblicato l'articolo? Il valore è una data. • `updated` — Quando è stato aggiornato l'articolo? Il valore è una data. *Elementi contenuti* • `nessuna` *Facets* • `nessuna`

20.5. Link e ulteriore documentazione

Alla base del supporto RSS c'è la libreria YARFRAW che si può trovare all'indirizzo http://yarfraw.sourceforge.net/ e da questa deriva la maggior parte delle caratteristiche e delle limitazioni.

Per i dettagli sul formato ATOM 1.0, si veda sulle *specifiche* [http://atompub.org/2005/07/11/draft-ietf-atompub-format-10.html]

Per i dettagli sul formato RSS 2.0, si veda su *le specifiche* [http://cyber.law.harvard.edu/rss/rss.html].

Email

Seam include ora dei componenti opzionali per modellare e spedire le email.

Il supporto email è fornito da `jboss-seam-mail.jar`. Questo JAR contiene i controlli JSF per la mail, che vengono impiegati per costruire le email, ed il componente manager `mailSession`.

Il progetto examples/mail contiene un esempio di supporto email in azione. Mostra un pacchetto idoneo e contiene esempi che mostrano le funzionalità chiave attualmente supportate.

Si può testare la propria mail usando l'ambiente di test di Seam. Si veda *Sezione 36.3.4, «Test d'integrazione di Seam Mail»*.

21.1. Creare un messaggio

Non occorre imparare un nuovo linguaggio di template per usare Seam Mail — un'email è semplicemente un facelet!

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
   xmlns:m="http://jboss.com/products/seam/mail"
   xmlns:h="http://java.sun.com/jsf/html">

   <m:from name="Peter" address="peter@example.com" />
   <m:to name="#{person.firstname} #{person.lastname}"
>#{person.address}</m:to>
   <m:subject
>Try out Seam!</m:subject>

   <m:body>
      <p
><h:outputText value="Dear #{person.firstname}" />,</p>
      <p
>You can try out Seam by visiting
      <a href="http://labs.jboss.com/jbossseam"
>http://labs.jboss.com/jbossseam</a
>.</p>
      <p
>Regards,</p>
      <p
>Pete</p>
   </m:body>

</m:message
>
```

Il tag `<m:message>` racchiude l'intero messaggio e dice a Seam di iniziare a generare la mail. Dentro al tag `<m:message>` si usa un tag `<m:from>` per impostare il mittente, un tag `<m:to>` per specificare un destinatario (si noti che EL viene impiegato come in un facelet normale), e un tag `<m:subject>`.

Il tag `<m:body>` racchiude il corpo della mail. Si possono usare tag HTML dentro il corpo così come componenti JSF.

Adesso che si ha un modello di email, come lo si spedisce? Alla fine della generazione di `m:message` viene chiamato `mailSession` per spedire l'email, e quindi semplicemente occorre chiedere a Seam di generare la vista:

```
@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());
   }
}
```

Per esempio, se si digita un indirizzo email non valido, allora viene lanciata un'eccezione che viene catturata e quindi mostrata all'utente.

21.1.1. Allegati

Seam semplifica l'uso degli allegati ad una mail. Supporta la maggior parte dei tipi java standard usati con i file.

Se si vuole spedire le email `jboss-seam-mail.jar`:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam caricherà il file dal classpath e lo allegherà alla mail. Di default verrà allegato come `jboss-seam-mail.jar`; se si vuole avere un altro nome basta aggiungere l'attributo `fileName`:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar" fileName="this-is-so-cool.jar"/>
```

Si può anche allegare un `java.io.File`, un `java.net.URL`:

```
<m:attachment value="#{numbers}"/>
```

Oppure un `byte[]` o un `java.io.InputStream`:

```
<m:attachment value="#{person.photo}" contentType="image/png"/>
```

Si noterà che per `byte[]` e `java.io.InputStream` occorrerà specificare il tipo MIME dell'allegato (poiché quest'informazione non viene trasportata come parte del file).

Ancora meglio si può allegare un PDF generato da Seam, o ogni altra vista JSF standard, semplicemente mettendo un `<m:attachment>` attorno ai normali tag:

```
<m:attachment fileName="tiny.pdf">
   <p:document
>
      A very tiny PDF
   </p:document>
</m:attachment
>
```

Con un set di file da allegare (per esempio un gruppo di fotografie caricate da un database= si utilizza un `<ui:repeat>`:

```
<ui:repeat value="#{people}" var="person">
                        <m:attachment value="#{person.photo}" contentType="image/jpeg"
 fileName="#{person.firstname}_#{person.lastname}.jpg"/>
</ui:repeat
>
```

E se si vuole mostrare un'immagine allegata inline:

```
<m:attachment
   value="#{person.photo}"
   contentType="image/jpeg"
   fileName="#{person.firstname}_#{person.lastname}.jpg"
   status="personPhoto"
   disposition="inline" />
```

```
<img src="cid:#{personPhoto.contentId}" />
```

Ci si potrebbe chiedere cosa faccia `cid:#{...}`. IETF specifica che mettendo questo come src dell'immagine, gli allegati verranno visualizzati quando si tenta di localizzare l'immagine (il `Content-ID` deve corrispondere) — magia!

Occorre dichiarare l'allegato prima di tentare di accedere allo stato dell'oggetto.

21.1.2. HTML/Text alternative part

Nonostante la maggior parte dei lettori email oggi supportino HTML, alcuni non lo fanno, quindi è possibile aggiungere un'alternativa di puro testo al corpo della mail:

```
<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
>
```

21.1.3. Destinatari multipli

Spesso si vuole spedire una mail adun gruppo di destinatari (per esempio i propri utenti). Tutti i tag dei destinatari possono essere messi dentro un `<ui:repeat>`:

```
<ui:repeat value="#{allUsers} var="user">
   <m:to name="#{user.firstname} #{user.lastname}" address="#{user.emailAddress}" />
</ui:repeat
>
```

21.1.4. Messaggi multipli

A volte, comunque, occorre spedire un messaggio leggermente diverso a ciascun utente (es. una password resettata). Il miglior modo per farlo è mettere l'interno messaggio dentro un `<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
>
```

21.1.5. Comporre template

L'esempio mail templating mostra che il templating facelets funziona solo con i tag mail di Seam.

`template.xhtml` contiene:

```
<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
>
```

`templating.xhtml` contiene:

```
<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
>
```

Si possono usare anche i tag sorgente di facelets nella propria email, ma occorre metterli in un jar dentro `WEB-INF/lib` - fare riferimento a `.taglib.xml` da `web.xml` non è molto affidabile quando

si usa Seam Mail (se si spedisce la mail in modo asincrono Seam Mail non ha accesso all'intero contesto JSF o Servlet, e quindi non conosce i parametri di configurazione di `web.xml`).

Se occorre configurare Facelets o JSF per spedire mail, servirà eseguire l'override del componente Renderer e configurarlo programmaticamente - solo per utenti avanzati!

21.1.6. Internazionalizzazione

Seam supporta l'invio di messaggi internazionalizzati. Di default, viene usata la codifica fornita da JSF, ma questa può essere sovrascritta nel template:

```
<m:message charset="UTF-8">
  ...
</m:message
>
```

Il corpo, soggetto e nome destinatario (e mittente)verranno codificati. Occorre assicurarsi che facelets usi il set di caratteri corretto per il parsing delle pagine, impostando la codifica del template:

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

21.1.7. Altre intestazioni

A volte si vogliono aggiungere altre intestazioni alla mail. Seam fornisce supporta per alcune di queste (si veda *Sezione 21.5, «Tag»*). Per esempio, si può impostare l'importanza della mail e chiedere una conferma di lettura:

```
<m:message xmlns:m="http://jboss.com/products/seam/mail"
   importance="low"
   requestReadReceipt="true"/>
```

Altrimenti si può aggiunge una qualsiasi intestazione al messaggio usando il tag `<m:header>`:

```
<m:header name="X-Sent-From" value="JBoss Seam"/>
```

21.2. Ricevere email

Se si usa EJB, allora si può impiegare MDB (Message Driven Bean) per ricevere una mail. JBoss fornisce un adattatore JCA — `mail-ra.rar` — ma la versione distribuita cone JBoss AS ha un

numero di limitazioni (e non è incorporata in alcune versioni) quindi si raccomanda di usare `mail-ra.rar` distribuito con Seam (si trova nella directory `extras/` nella distribuzione Seam). `mail-ra.rar` deve essere messo in `$JBOSS_HOME/server/default/deploy`; se la versione di JBoss già lo contiene, lo si sostituisca.

Lo si può configurare come:

```
@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());
   }

}
```

Ogni messaggio ricevuto causerà una chiamata a `onMessage(Message message)`. La maggior parte delle annotazioni di Seam funzioneranno dentro MDB, manon sarà possibile accedere al contesto di persistenza.

Si possono trovare maggiori informazioni su `mail-ra.rar` all'indirizzo *http://wiki.jboss.org/wiki/Wiki.jsp?page=InboundJavaMail*.

Se non si usa JBoss AS si può comunque usare `mail-ra.rar` oppure si cerchi se il proprio application server include un adattatore simile.

21.3. Configurazione

Per includere il supporto email nella propria applicazione, si includa `jboss-seam-mail.jar` nella directory `WEB-INF/lib`. Se si usa JBoss AS non c'è nessuna configurazione ulteriore per usare il supporto email di Seam. Altrimenti occorre assicurarsi di avere JavaMail API, un'implementazione di JavaMail API presente (l'API e l'impl usante in JBoss AS sono distribuite

con seam in `lib/mail.jar`), ed una copia di Java Activation Framework (distribuito con seam in `lib/activation.jar`.

Nota

Il module Seam Mail richiede l'uso di Facelets come tecnologia per la vista. Future versioni della libreria potranno supportare l'uso di JSP. In aggiunta si richiede l'uso del pacchetto seam-ui.

Il componente `mailSession` usa JavaMail per dialogare con un server SMTP 'reale'.

21.3.1. `mailSession`

Una sessione JavaMail può essere disponibile via ricerca JNDI se si lavora in ambiente JEE o si può usare una sessione configurata da Seam.

Le proprietà del componente mailSession sono descritte con maggior dettaglio in *Sezione 32.9, «Componenti relativi alla Mail»*.

21.3.1.1. JNDI lookup in JBoss AS

`deploy/mail-service.xml` di JBoss AS configura una sessione JavaMail legandla a JNDI. La configurazione di default del servizio deve essere modificata per la propria rete. *http://wiki.jboss.org/wiki/Wiki.jsp?page=JavaMail* descrive il servizion con maggiore dettaglio.

```
<components xmlns="http://jboss.com/products/seam/components"
   xmlns:core="http://jboss.com/products/seam/core"
   xmlns:mail="http://jboss.com/products/seam/mail">

   <mail:mail-session session-jndi-name="java:/Mail"/>

</components
>
```

Qua viene detto a Seam di ottenere da JNDI la sessione mail legata `java:/Mail`.

21.3.1.2. Sessione configurata da Seam

Una sessione mail può essere configurata via `components.xml`. Qua viene indicato a Seam di usare `smtp.example.com` come server SMTP:

```
<components xmlns="http://jboss.com/products/seam/components"
   xmlns:core="http://jboss.com/products/seam/core"
```

```
    xmlns:mail="http://jboss.com/products/seam/mail">

    <mail:mail-session host="smtp.example.com"/>

</components
>
```

21.4. Meldware

L'esempio mail di Seam usa Meldware (da *buni.org* [http://buni.org]) come server mail. Meldware è un pacchetto groupware che fornisce `SMTP`, `POP3`, `IMAP`, webmail, un calendario condiviso ed un tool di amministrazione grafico; è scritto come applicazione JEE e quindi può essere deployato in JBoss AS assieme alla propria applicazione Seam.

Attenzione

La versione di Meldware distribuita con Seam (scaricata a richiesta) è configurata per lo sviluppo - caselle di posta, utenti e alias (indirizzi email) sono creati ogni volta che si esegue il deploy dell'applicazione. Se si vuole usare Meldware in produzione occorre installare l'ultima release da *buni.org* [http://buni.org].

21.5. Tag

Le email vengono generate usando tag nel namespace `http://jboss.com/products/seam/mail`. I documenti dovrebbero sempre avere il tag `message` alla radice del messaggio. Il tag message prepara Seam a generare una email.

I tag standard per il templating di facelets possono venire usati normalmente. Dentro il corpo si può usare qualsiasi tag JSF; se viene richiesto l'accesso a risorse esterne (fogli di stile, javascript) allora ci si assicuri di impostare `urlBase`.

<m:message>
Tag radice di un messaggio mail

- `importance` — low, normal or high. Di default è normal, questo imposta l'importanza del messaggio email.
- `precedence` — imposta la precedenza del messaggio (es. bulk).
- `requestReadReceipt` — di default è false, se impostato, verrà aggiunta un richiesta di conferma di lettura, da inviare all'indirizzo `From:`.
- `urlBase` — Se impostato, il valore è messo prima di `requestContextPath` consentendo di usare nelle proprie email componenti quali `<h:graphicImage>`.

- `messageId` — Imposta esplicitamente il Message-ID

<m:from>
Si imposti l'indirizzo From: per la mail. Se ne può avere uno soltanto per email.

- `name` — il nome da cui l'email deve provenire.
- `address` — l'indirizzo email da cui la mail deve provenire.

<m:replyTo>
Si imposti l'indirizzo Reply-to: per la mail. Se ne può avere uno soltanto per email.

- `address` — l'indirizzo email da cui la mail deve provenire.

<m:to>
Si aggiunga un destinatario alla mail. Si usino più tag <m:to> per destinatari multipli. Questo tag può essere collocato in modo sicuro dentro un tag repeat quale <ui:repeat>.

- `name` — il nome del destinatario.
- `address` — l'indirizzo email del destinatario.

<m:cc>
Si aggiunga un destinatario cc alla mail. Si usino più tag <m:cc> per destinatari cc multipli. Questo tag può essere collocato in modo sicuro dentro un tag iteratore quale <ui:repeat>.

- `name` — il nome del destinatario.
- `address` — l'indirizzo email del destinatario.

<m:bcc>
Si aggiunga un destinatario ccn alla mail. Si usino più tag <m:bcc> per destinatari ccn multipli. Questo tag può essere collocato in modo sicuro dentro un tag repeat quale <ui:repeat>.

- `name` — il nome del destinatario.
- `address` — l'indirizzo email del destinatario.

<m:header>
Aggiungere un'intestazione alla mail (es. `X-Sent-From: JBoss Seam`)

- `name` — Il nome dell'intestazione da aggiungere (es. `X-Sent-From`).
- `value` — Il valore dell'intestazione da aggiungere (es. `JBoss Seam`).

<m:attachment>
Aggiungere un allegato alla mail.

- `value` — Il file da allegare.
  - `String` — Una `String` è interpretata come path al file all'interno del classpath

- `java.io.File` — Un'espressione EL può fare riferimento ad un oggetto `File`
- `java.net.URL` — Un'espressione EL può fare riferimento ad un oggetto `URL`
- `java.io.InputStream` — Un'espressione EL può fare riferimento ad un oggetto `InputStream`. In questo caso devono essere specificati entrambi i `fileName` e `contentType`.
- `byte[]` — Un'espressione EL può fare riferimento a `byte[]`. In questo caso devono essere specificati entrambi i `fileName` e `contentType`.

Se viene omesso l'attributo value:

- Se questo tag contiene un tag `<p:document>`, il documento descritto sarà generato ed allegato alla mail. Deve essere specificato un `fileName`.
- Se questo tag contiene altri tag JSF, da questi verrà generato un documento HTML ed allegato alla mail. Un `fileName` dovrebbe sempre essere specificato.

- `fileName` — Specifica il nome del file da usare per il file allegato.
- `contentType` — Specifica il tipo MIME del file allegato.

<m:subject>
Impostiamo l'oggetto della mail.

<m:body>
Si imposti il corpo della mail. Supporta un facet `alternativo` che, se viene generata una mail HTML, può contenere testo alternativo per un lettore di email che non supporta HTML.

- `type` — Se impostato a `plain` allora verrà generata una mail con semplice testo, altrimenti una mail HTML.

Asincronicità e messaggistica

Seam semplifica molto l'esecuzione di lavori asincroni da richiesta web. Quando la maggior parte delle persone pensa all'asincronicità in Java EE, pensano all'uso di JMS. Questo è certamente un modo per approcciare il problema in Seam, ed è quello giusto quando si hanno dei requisiti di qualità di servizio molto stringenti e ben definiti. Seam semplifica l'invio e la ricezione di messaggi JMS usando i componenti Seam.

Ma per molti casi d'uso, JMS è eccessivo. Seam aggiunge come nuovo layer un semplice metodo asincrono ed un meccanismo d'eventi nella scelta del *dispatcher*:

- `java.util.concurrent.ScheduledThreadPoolExecutor` (di default)
- Il servizio EJB timer (per ambienti EJB 3.0)
- Quartz

22.1. Asincronicità

Eventi asincroni e chiamate a metodi hanno le stesse aspettative di qualità di servizio del meccanismo sottostante di dispatcher. Il dispatcher di default, basato su `ScheduledThreadPoolExecutor` opera efficientemente ma non fornisce alcun supporto ai task asincroni di persistenza, e quindi non garantisce che un task verrà effettivamente eseguito. Se si lavora in un ambiente che supporta EJB 3.0 e si aggiunge la seguente linea a `components.xml`:

```
<async:timer-service-dispatcher/>
```

allora i task asincroni verranno processati dal servizio EJB timer del container. Se non si è familiari come il servizio Timer, nessuna paura, non si deve interagire direttamente con esso per usare i metodi asincroni in Seam. La cosa importante da sapere è che qualsiasi buona implementazione di EJB 3.0 avrà l'opzione di usare i timer di persistenza, che danno garanzia che i task verranno processati.

Un'altra alternativa è quella di usare la libreria open source Quartz per gestire il metodo asincrono. Occorre incorporare il JAR della libreria Quartz (che si trova nella directory `lib`) nell'EAR e dichiararla come modulo Java in `application.xml`. Il dispatcher Quartz può essere configurato aggiungendo un file di proprietà Quartz al classpath. Deve essere nominato `seam.quartz.properties`. Inoltre occorre aggiungere la seguente linea a `components.xml` per installare il dispatcher Quartz.

```
<async:quartz-dispatcher/>
```

L'API di Seam per il `ScheduledThreadPoolExecutor` di default, il `Timer` EJB3, e lo `Scheduler` Quartz sono più o meno la stessa cosa. Vengono azionati come "plug and play" aggiungendo una linea a `components.xml`.

22.1.1. Metodi asincroni

Nella forma più semplice, una chiamata asincrona consente che una chiamata di metodo venga processata asincronicamente (in un thread differente) dal chiamante. Solitamente si usa una chiamata asincrona quando si vuole ritornare una risposta immediata al client, e si lascia in background il lavoro dispendioso da processare. Questo pattern funziona molto bene nelle applicazioni che usano AJAX, dove il client può automaticamente interrogare il server per ottenere il risultato del lavoro.

Per i componenti EJB si annota l'interfaccia locale per specificare che un metodo viene processato asincronicamente.

```
@Local
public interface PaymentHandler
{
   @Asynchronous
   public void processPayment(Payment payment);
}
```

(Per i componenti JavaBean, se si vuole, si può annotare la classe d'implementazione del componente.)

L'uso dell'asincronicità è trasparente alla classe bean:

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
   public void processPayment(Payment payment)
   {
      //do some work!
   }
}
```

E è anche trasparente al 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";
   }
}
```

Il metodo asincrono viene processato in un contesto eventi completamente nuovo e non ha accesso allo stato del contesto sessione o conversazione del chiamante. Comunque, il contesto di processo di business *viene* propagato.

Le chiamate del metodo asincrono possono essere schedulate per un'esecuzione successiva usando le annotazioni `@Duration`, `@Expiration` e `@IntervalDuration`.

```
@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";
    }
}
```

Entrambi il server ed il client possono accedere all'oggetto `Timer` associato all'invocazione. L'oggetto `Timer` mostrato sotto è il timer EJB3 quando viene usato il dispatcher EJB3. Per il `ScheduledThreadPoolExecutor` di default, l'oggetto restituito è `Future` di JDK. Per il dispatcher Quartz, viene ritornato `QuartzTriggerHandle`, che verrà discusso nella prossima sessione.

```
@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";
   }
}
```

I metodi asincroni non possono ritornare al chiamante alcun altro valore.

22.1.2. Metodi asincroni con il Quartz Dispatcher

Il dispatcher Quartz (vedere indietro come viene installato) consente di usare le annotazioni `@Asynchronous`, `@Duration`, `@Expiration`, e `@IntervalDuration` come sopra. Ma possiede anche altre potenti caratteristiche. Il dispatcher Quartz supporta tre nuove annotazioni.

L'annotazione `@FinalExpiration` specifica una data finale per il task ricorrente. Si noti che si può iniettare il `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);
```

Si noti che il metodo restituisce l'oggetto `QuartzTriggerHandle`, che si può usare per arrestare, mettere in pausa e ripristinare lo scheduler. L'oggetto `QuartzTriggerHandle` è serializzabile, e quindi puoi salvarlo nel database se deve essere presente per un periodo di tempo esteso.

```
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();
```

L'annotazione `@IntervalCron` supporta la sintassi Unix del cron job per la schedulazione dei task. Per esempio, il seguente metodo asincrono gira alle 2:10pm e alle 2:44pm ogni Mercoledì del mese di Marzo.

```
  // 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);
```

L'annotazione `@IntervalBusinessDay` supporta l'invocazione sullo scenario di "Giorno Lavorativo ennesimo". Per esempio il seguente metodo asincrono gira alle ore 14.00 del secondo giorno lavorativo di ogni mese. Di default esclude tutti i weekend e le festività federali americane fino al 2010 dai giorni lavorativi.

```
// 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);
```

L'oggetto `NthBusinessDay` contiene la configurazione del trigger d'invocazione. Si possono specificare più vacanze (esempio, vacanze aziendali, festività non-US, ecc.) attraverso la proprietà `additionalHolidays`.

```
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;
    }
    ... ...
}
```

Le annotazioni `@IntervalDuration`, `@IntervalCron`, e `@IntervalNthBusinessDay` sono mutualmente esclusive. Se usate nello stesso metodo, verrà lanciata una `RuntimeException`.

22.1.3. Eventi asincroni

Eventi guidati dai componenti possono essere asincroni. Per sollevare un evento da processare in modo asincrono, occorre semplicemente chiamare il metodo `raiseAsynchronousEvent()` della classe `Events`. Per schedulare un evento a tempo, chiamare il metodo `raiseTimedEvent()` passando un oggetto *schedule* (per il dispatcher di default o il dispatcher delservizio timer, usare `TimerSchedule`). I componenti possono osservare eventi asincroni nel solito modo, ma si tenga presente che solo il contesto business process viene propagato nel thread asincrono.

22.1.4. Gestione delle eccezione da chiamate asincrone

Ogni dispatcher asincrono si comporta in modo differente quando attraverso di esso viene propagata un'eccezione. Per esempio, il dispatcher `java.util.concurrent` sospenderà ogni altra esecuzione di una chiamata che si ripete, ed il servizio timer EJB3 inghiottirà quest'eccezione. Quindi Seam cattura ogni eccezione che si propaga fuori da una chiamata asincrona prima che questa raggiunga il dispatcher.

Di default ogni eccezione che si propaga fuori da un'esecuzione asincrona verrà catturata e loggata come errore. Si può personalizzare questo comportamento facendo override del componente `org.jboss.seam.async.asynchronousExceptionHandler`.

```
@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);
  }

}
```

Per esempio, usando il dispatcher `java.util.concurrent`, viene iniettato il suo oggetto di controllo e vengono cancellate tutte le future invocazioni quando si incontra un'eccezione.

Si può alterare questo comportamento per un componente individuale, implementando sul componente il metodo `public void handleAsynchronousException(Exception exception);`. Per esempio:

```
public void handleAsynchronousException(Exception exception) {
  log.fatal(exception);
}
```

22.2. Messaggistica in Seam

Seam facilita l'invio e la ricezione di messaggi JMS da e verso componenti Seam.

22.2.1. Configurazione

Per configurare l'infrastruttura di Seam alla spedizione di messaggi JMS, occorre dire a Seam a quali topic e code si vuole spedire i messaggi, ed inoltre serve dire dove trovare `QueueConnectionFactory` e/o `TopicConnectionFactory`.

Di default Seam usa `UIL2ConnectionFactory` che è la connection factory consueta per l'uso con JBossMQ. Se si impiegano altri provider JSM, occorre impostare uno o entrambi i `queueConnection.queueConnectionFactoryJndiName` e `topicConnection.topicConnectionFactoryJndiName` in `seam.properties`, `web.xml` o `components.xml`.

Inoltre in `components.xml` occorre elencare i topic e le code per installare i `TopicPublisher` ed i `QueueSender` gestiti da Seam:

```
<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"/>
```

22.2.2. Spedire messaggi

Ora è possibile iniettare in qualsiasi componente un `TopicPublisher` e un `TopicSession` JMS:

```
@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);
    }
}
```

O lavorando con una coda:

```
@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);
    }
}
```

22.2.3. Ricezione dei messaggi usando un bean message-driven

I messaggi possono essere processati usando un qualsiasi bean message driven EJB3. I bean message-drive possono essere anche componenti Seam, in qual caso è possibile iniettare altri componenti Seam aventi scope evento e applicazione.

22.2.4. Ricezione dei messaggi nel client

Seam Remoting consente di sottoscrivere un topic JMS lato client JavaScript. Questo viene descritto in *Capitolo 25, Remoting*.

Gestione della cache

In quasi tutte le applicazioni gestionali il database è il principale collo di bottiglia e lo strato meno scalabile dell'ambiente di esecuzione. Chi utilizza ambienti PHP/Ruby cercherà di sostenere che le architetture cosiddette "shared nothing" (nessuna condivisione) hanno una buona scalabilità. Benché questo possa essere letteralmente vero, non si conoscono molte applicazioni multi utente che possano essere implementate senza la condivisione di risorse tra diversi nodi di un cluster. In effetti ciò a cui questi sprovveduti stanno pensando è un'architettura con "nessuna condivisione eccetto il database". Ovviamente condividere il database è il principale problema di scalabilità di una applicazione multi utente, perciò affermare che questa architettura è altamente scalabile è assurdo e ci dice molto sul tipo di applicazioni sul cui sviluppo questi signori spendono la maggior parte del tempo.

Quasi tutto ciò che riusciamo a fare per condividere il database *meno frequentemente* è ben fatto.

E questo richiede una cache (memoria tampone). Un'applicazione Seam ben progettata comprenderà una ricca e stratificata strategia di cache che interessi ogni strato dell'applicazione:

- Il database, chiaramente, ha la propria cache. Questo è enormemente importante, ma non può scalare come una cache nello strato applicativo.
- La soluzione ORM scelta (Hibernate o qualche altra implementazione JPA) ha una cache di secondo livello per i dati provenienti dal database. Questa è una caratteristica molto potente, ma spesso male utilizzata. In un ambiente cluster, mantenere i dati in una cache in modo consistente dal punto di vista delle transazioni con l'intero cluster e con il database, è molto dispendioso. Ha più senso per i dati condivisi tra molti utenti e che vengono aggiornati raramente. Nelle architetture tradizionali senza stato, spesso si cerca di usare la cache di secondo livello per conservare lo stato di una conversazione. Questo è sempre un male ed è particolarmente sbagliato in Seam.
- Il contesto Conversation di Seam è una cache per lo stato della conversazione. I componenti che vengono messi nel contesto conversation posso mantenere lo stato relativo all'interazione dell'utente.
- In particolare, un persistence context gestito da Seam (o un persistence context EJB gestito dal container associato ad uno stateful session bean legato ad una conversazione) agisce con una cache per i dati che sono stati letti nella conversazione attuale. Questa cache tende ad avere una frequenza di utilizzo piuttosto alta! Seam ottimizza la replica dei persistence context gestiti da Seam in un ambiente cluster e non c'è bisogno di curare la consistenza transazionale con il database (il lock ottimistico è sufficiente) perciò non è necessario preoccuparsi troppo per le implicazioni sulle prestazioni di questa cache, a meno che non si leggano migliaia di oggetti con un singolo persistence context.
- L'applicazione può conservare uno stato non transazionale nell'application context di Seam. Lo stato mantenuto nell'applicazione context ovviamente non è visibile agli altri nodi del cluster.

- L'applicazione può conservare uno stato transazionale usando il componente `cacheProvider` di Seam, il quale si integra con JBossCache, JBoss POJO Cache oppure EHCache nell'ambiente Seam. Questo stato risulterà visibile agli altri nodi se la cache gestisce il funzionamento in modalità cluster.
- Infine Seam consente di conservare dei frammenti generati di una pagina JSF. A differenza della cache di secondo livello dell'ORM, questa cache non viene automaticamente invalidata quando i dati cambiano, perciò è necessario scrivere il codice applicativo necessario per realizzare una invalidazione esplicita, oppure importare degli opportuni criteri di scadenza.

Per ulteriori informazioni sulla cache di secondo livello è necessario fare riferimento alla documentazione della soluzione ORM scelta poiché si tratta di un argomento estremamente complesso. In questo paragrafo verrà affrontato l'uso diretto della cache, tramite il componente `cacheProvider` o come cache dei frammenti di pagina, tramite il controllo `<s:cache>`.

23.1. Usare la cache in Seam

Il componente `cacheProvider` fornito dal framework gestisce una istanza di:

JBoss Cache 1.x (adatto per l'uso in JBoss 4.2.x o successivo e altri container)
`org.jboss.cache.TreeCache`

JBoss Cache 2.x (adatto per l'uso in JBoss 5.x e altri container)
`org.jboss.cache.Cache`

JBoss POJO Cache 1.x (adatto per l'uso in JBoss 4.2.x o successivo e altri container)
`org.jboss.cache.aop.PojoCache`

EHCache (adatto per l'uso in qualsiasi container)
`net.sf.ehcache.CacheManager`

E' possibile mettere con sicurezza nella cache qualsiasi oggetto Java immutabile; esso verrà conservato nella cache e replicato nel cluster (assumendo che la replica sia gestita e abilitata). Se si vuole mantenere degli oggetti mutabili nella cache occorre leggere la documentazione della cache sottostante per scoprire come notificare la cache dei cambiamenti negli oggetti.

Per usare `cacheProvider` è necessario includere nel progetto i jar dell'implementazione della cache:

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

Suggerimento

Se si sta usando JBoss Cache in un container diverso da JBoss Application Server, verificare sul *wiki* [http://wiki.jboss.org/wiki/JBossCache] di JBoss Cache per le ulteriori dipendenze.

Se l'applicazione Seam viene assemblata in un EAR si raccomanda di inserire direttamente nell'EAR la configurazione e i jar della cache.

Sarà inoltre necessario fornire un file di configurazione per JBossCache. Posizionare `treecache.xml` con un'opportuna configurazione di cache nel classpath (ad esempio nel jar ejb oppure in `WEB-INF/classes`). JBossCache ha molte impostazioni ostiche e terribili perciò non verranno discusse qui. Per ulteriori informazioni fare riferimento alla documentazione di JBossCache.

E' possibile trovare un `treecache.xml` di esempio in `examples/blog/resources/treecache.xml`.

EHCache senza un file di configurazione funzionerà nella sua configurazione di default.

Per modificare il file di configurazione in uso, configurare la cache in `components.xml`:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider configuration="META-INF/cache/treecache.xml" />
</components
>
```

A questo punto è possibile iniettare la cache in qualsiasi componente Seam:

```
@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;
   }
}
```

Se nell'applicazione si vogliono avere più configurazioni della cache basta usare `components.xml` per configurare più componenti `cacheProvider`:

```
<components xmlns="http://jboss.com/products/seam/components"
         xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider name="myCache" configuration="myown/cache.xml"/>
  <cache:jboss-cache-provider name="myOtherCache" configuration="myother/cache.xml"/>
</components
>
```

23.2. Cache dei frammenti di pagina

L'uso più interessante della cache in Seam è la tag `<s:cache>`, che è la soluzione offerta da Seam per il problema di conservare in cache frammenti di pagine JSF. Internamente `<s:cache>` usa `pojoCache`, perciò è necessario seguire i passi elencati in precedenza prima di poterla usare (mettere i jar nell'EAR, sistemare le terribili opzioni di configurazione, ecc).

`<s:cache>` viene usata per conservare quei contenuti generati che cambiano raramente. Ad esempio la pagina di benvenuto del nostro blog mostra le voci del blog più recenti:

```
<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
>
```

La chiave (`key`) consente di avere più versioni in cache di ogni frammento di pagina. In questo caso c'è una versione in cache per ogni blog. La regione (`region`) determina la cache o il nodo region in cui tutte le versioni verranno conservate. Diversi nodi possono avere diverse regole di scadenza (queste sono le cose che si impostano con le summenzionate terribili opzioni di configurazione).

Ovviamente il grosso problema di `<s:cache>` è che è troppo semplice sapere quando i dati contenuti cambiano (ad esempio quando il blogger pubblica uno nuovo contenuto). Così è necessario eliminare i frammento in cache manualmente:

```
public void post() {
   ...
   entityManager.persist(blogEntry);
   cacheProvider.remove("welcomePageFragments", "recentEntries-" + blog.getId() );
}
```

In alternativa, se non è critico che le modifiche siano immediatamente visibili agli utenti, è possibile impostare un tempo di scadenza breve nel nodo della cache.

Web Service

Seam si integra con JBossWS per consentire allo standard JEE web service di sfruttrare pienamente il framework contestuale di Seam, includendo il supporto ai web service conversazionali. Questo capitolo passa in rassegna tutti i passi richiesti per consentire ai web service di funzionare in ambiente Seam.

24.1. Configurazione ed impacchettamento

Per consentire a Seam di intercettare le richieste web service in modo tale da creare i contesti Seam necessari per la richiesta, deve essere configurato uno speciale handler SOAP; `org.jboss.seam.webservice.SOAPRequestHandler` è un'implementazione `SOAPHandler` che esegue il lavoro di gestione del ciclo di vita di Seam durante lo scope di una richiesta web service.

Uno speciale file di configurazione, `standard-jaxws-endpoint-config.xml`, deve essere collocato nella directory `META-INF` del file `jar` che contiene le classi web service. Questo file contiene la seguente configurazione handler SOAP:

```
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:javaee="http://java.sun.com/xml/ns/javaee"
              xsi:schemaLocation="urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd">
  <endpoint-config>
    <config-name
>Seam WebService Endpoint</config-name>
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings
>##SOAP11_HTTP</javaee:protocol-bindings>
          <javaee:handler>
            <javaee:handler-name
>SOAP Request Handler</javaee:handler-name>
            <javaee:handler-class
>org.jboss.seam.webservice.SOAPRequestHandler</javaee:handler-class>
          </javaee:handler>
        </javaee:handler-chain>
    </pre-handler-chains>
  </endpoint-config>
</jaxws-config
>
```

24.2. Web Service conversazionali

Quindi come vengono propagate le conversazioni tra le richieste web service? Seam usa un elemento di intestazione SOAP presente in entrambi i messaggi di richiesta e di risposta SOAP per portare l'ID della conversazione dal consumatore al servizio, e viceversa. Ecco un esempio di richiesta web service che contiene un ID di conversazione:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:seam="http://seambay.example.seam.jboss.org/">
  <soapenv:Header>
    <seam:conversationId xmlns:seam='http://www.jboss.org/seam/webservice'
>2</seam:conversationId>
  </soapenv:Header>
  <soapenv:Body>
    <seam:confirmAuction/>
  </soapenv:Body>
</soapenv:Envelope
>
```

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 `2`. 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.

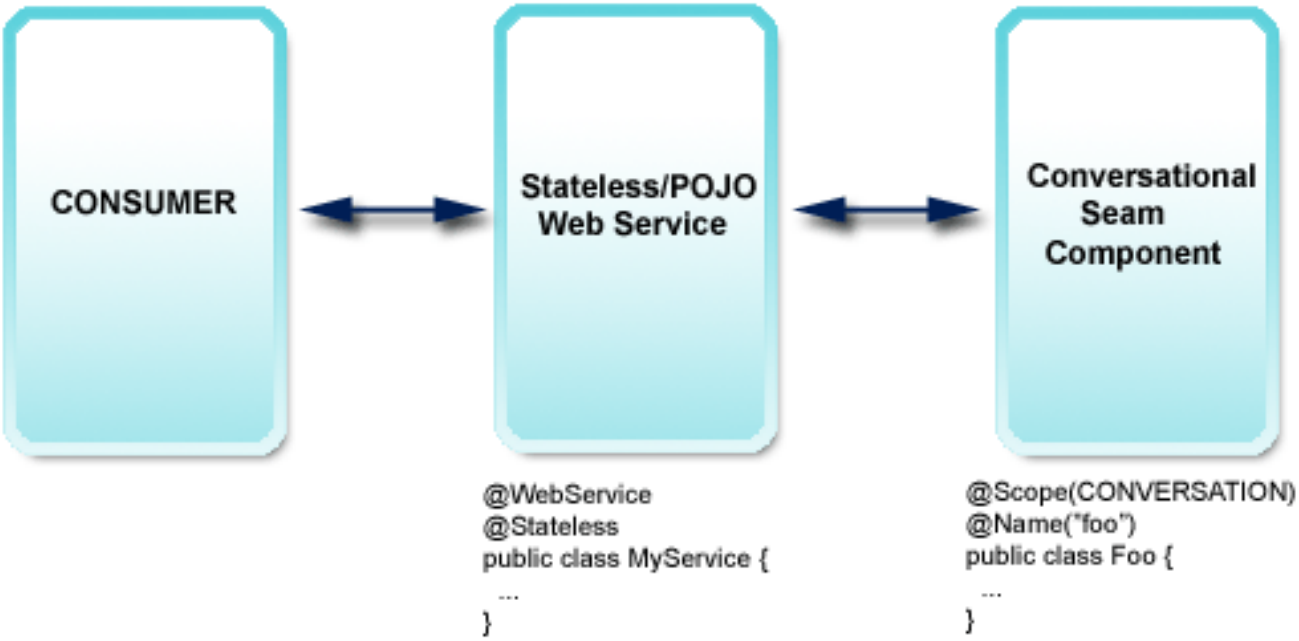
An important thing to note is that the `conversationId` header element must be qualified with a namespace of `http://www.jboss.org/seam/webservice`, otherwise Seam will not be able to read the conversation ID from the request. Here's an example of a response to the above request message:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header>
    <seam:conversationId xmlns:seam='http://www.jboss.org/seam/webservice'
>2</seam:conversationId>
  </env:Header>
  <env:Body>
    <confirmAuctionResponse xmlns="http://seambay.example.seam.jboss.org/"/>
  </env:Body>
</env:Envelope
>
```

As you can see, the response message contains the same `conversationId` element as the request.

24.2.1. Una strategia raccomandata

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.

24.3. Esempio di 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
@WebService(name = "AuctionService", serviceName = "AuctionService")
public class AuctionService implements AuctionServiceRemote
{
  @WebMethod
  public boolean login(String username, String password)
  {
    Identity.instance().setUsername(username);
```

```
    Identity.instance().setPassword(password);
    Identity.instance().login();
    return Identity.instance().isLoggedIn();
  }

  // snip
}
```

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.

24.4. Webservice RESTful HTTP con 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).

24.4.1. 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`

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 pattern mapped in `web.xml` for the `SeamResourceServlet`, e.g `/seam/resource` if you follow the common examples. 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`) 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 `/seam/resource/rest`. We recommend that you change this string in your application, 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): `/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 `/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/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 defauls are acceptable. However, you can configure RESTEasy in your Seam application. First import the `resteasy` namespace into your XML configuration file header:

```
<components
  xmlns="http://jboss.com/products/seam/components"
  xmlns:resteasy="http://jboss.com/products/seam/resteasy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    http://jboss.com/products/seam/resteasy
       http://jboss.com/products/seam/resteasy-2.1.xsd
    http://jboss.com/products/seam/components
       http://jboss.com/products/seam/components-2.1.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 `/seam/resource/restv1/{resource}` - note that your `@Path` definitions and mappings do NOT change. This is an application-wide switch usually used for versioning of the HTTP API.

You can disable stripping of the base path if you'd like to map the full path in your resources:

```
<resteasy:application strip-seam-resource-path="false"/>
```

The path of a resource is now mapped with e.g. `@Path("/seam/resource/rest/customer")`. We do not recommend disabling this feature, as your resource class mappings are then bound to a particular deployment scenario.

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
>
```

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 EJB resources will not be found even if scanning of resources is enabled, you always have to list them manually. Again, this is only relevant for EJB resources that are not also Seam components and that do not have a `@Name` annotation.

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.

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`.

24.4.2. Risorse e provider come componenti Seam

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 and providers 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. However, you get full Seam injection support and all other Seam components and contexts are available to you. Currently also supported are `SESSION`, `APPLICATION`, and `STATELESS` resource components. Remember that any HTTP request has to transmit a valid session identifier (cookie, URI path parameter) for correct handling of the server-side session context.

Conversation-scoped resource components and mapping of conversations is currently not supported but will be available soon.

Provider classes can also be Seam components, they must be `APPLICATION`-scoped or `STATELESS`.

Resources and providers can be EJBs or JavaBeans, like any other Seam component.

24.4.3. 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 requests. The following example intercepts *all* HTTP requests and enables Seam exception handling:

```
<filter>
    <filter-name
>Seam Filter</filter-name>
    <filter-class
>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
```

```
   <filter-name
>Seam Filter</filter-name>
   <url-pattern
>/*</url-pattern>
</filter-mapping
>
```

To convert the unchecked `UnsupportedOperationException` thrown by your resource methods to a `501 Not Implemented` HTTP status response, add the following to your `pages.xml` descriptor:

```
<exception class="java.lang.UnsupportedOperationException">
   <http-error error-code="501">
      <message
>The request 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
>The service is currently 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`!

Remoting

Seam fornisce un metodo per accedere in modo remoto i componenti da una pagina web, usando AJAX (Asynchronous Javascript and XML). Il framework per questa funzionalità viene fornito con quasi nessuno sforzo di sviluppo - i componenti richiedono solamente una semplice annotazione per diventare accessibile via AJAX. Questo capitolo descrive i passi richiesti per costruire una pagina web abilitata a AJAX, poi spiega con maggior dettaglio le caratteristiche del framework Seam Remoting.

25.1. Configurazione

Per usare remoting, il resource servlet di Seam deve essere innanzitutto configurato nel file `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
>
```

Il passo successivi è importare il Javascript necessario nella propria pagina web. Ci sono un minimo di due script da importare. Il primo contiene tutto il codice del framework lato client che abilita le funzionalità di remoting:

```
<script type="text/javascript" src="seam/resource/remoting/resource/remote.js"
></script
>
```

Il secondo script contiene gli stub e le definizioni tipo per i componenti da chiamare. Viene generato dinamicamente basandosi sull'interfaccia locale dei propri componenti, ed include le definizioni tipo per tutte le classi che possono essere usate per chiamare i metodi remoti dell'interfaccia. Il nome dello script riflette il nome del componente. Per esempio se si ha un bean

di sessione stateless annotato con `@Name("customerAction")`, allora il tag dello script dovrebbe essere simile a:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction"
></script
>
```

Se si vuole accedere a più di un componente dalla stessa pagina, allora li si includa tutti come parametri nel tag script:

```
<script type="text/javascript"
      src="seam/resource/remoting/interface.js?customerAction&accountAction"
></script
>
```

In alternativa si può usare il tag `s:remote` per importare il Javascript richiesto. Si separi ciascun componente o nome di classe che si vuole importare con una virgola:

```
 <s:remote include="customerAction,accountAction"/>
```

25.2. L'oggetto "Seam"

L'interazione lato client con i componenti viene eseguita tutta tramite l'oggetto Javascript `Seam`. Quest'oggetti è definito in `remote.js`, e lo si userà per fare chiamate asincrone verso il componente. E' suddiviso in due aree di funzionalità; `Seam.Component` contiene metodi per lavorare con i componenti e `Seam.Remoting` contiene metodi per eseguire le richieste remote. La via più facile per diventare familiare con quest'oggetto è cominciare con un semplice esempio.

25.2.1. Esempio Hello World

Si cominci con un semplice esempio per vedere come funziona l'oggetto `Seam`

```
@Stateless
@Name("helloAction")
public class HelloAction implements HelloLocal {
   public String sayHello(String name) {
      return "Hello, " + name;
```

```
   }
}
```

You also need to create a local interface for our new component - take special note of the `@WebRemote` annotation, as it's required to make our method accessible via remoting:

```
@Local
public interface HelloLocal {
  @WebRemote
  public String sayHello(String name);
}
```

That's all the server-side code we need to write. 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.

25.2.2. 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.

25.2.2.1. 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");
// Oppure si può impostare direttamente i campi
```

```
customer.lastName = "Smith";
```

25.2.2.2. 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);
```

25.2.2.3. 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");
```

25.2.3. 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.

25.2.3.1. 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");
```

25.2.3.2. 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.

25.3. Valutazione delle espressioni EL

Seam Remoting also supports the evaluation of EL expressions, which provides another convenient method for retrieving data from the server. Using the `Seam.Remoting.eval()` function, an EL expression can be remotely evaluated on the server and the resulting value returned to a client-side callback method. This function accepts two parameters, the first being the EL expression to evaluate, and the second being the callback method to invoke with the value of the expression. Here's an example:

```
function customersCallback(customers) {
  for (var i = 0; i < customers.length; i++) {
    alert("Got customer: " + customers[i].getName());
  }
}

Seam.Remoting.eval("#{customers}", customersCallback);
```

In this example, the expression `#{customers}` is evaluated by Seam, and the value of the expression (in this case a list of Customer objects) is returned to the `customersCallback()` method. It is important to remember that the objects returned this way must have their types imported (via `s:remote`) to be able to work with them in Javascript. So to work with a list of `customer` objects, it is required to import the `customer` type:

```
<s:remote include="customer"/>
```

25.4. Interfacce client

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.

25.5. Il contesto

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.

25.5.1. Impostazione e lettura dell'ID di conversazione

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.

25.5.2. 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} );
```

25.6. Richieste batch

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 containining 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`.

25.7. Lavorare con i tipi di dati

25.7.1. Primitives / Basic Types

This section describes the support for basic data types. On the server side these values are generally compatible with either their primitive type or their corresponding wrapper class.

25.7.1.1. String

Simply use Javascript String objects when setting String parameter values.

25.7.1.2. 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.

25.7.1.3. Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

25.7.2. 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()`.

25.7.3. Date e orari

I valori delle date sono serializzati in una rappresentazione di Strina che è accurata al millisecondo. Lato client, si usi un oggetto Date Javascript per lavorare con i valori delle date. Late server, si usi `java.util.Date` (o discendenti, come le classi `java.sql.Date` o `java.sql.Timestamp`).

25.7.4. 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.

25.7.5. Collections

25.7.5.1. 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.

25.7.5.2. 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).

25.8. 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);
```

O lo si configuri 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)`.

25.9. Gestione delle eccezioni

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.

25.10. Il messaggio di caricamento

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.

25.10.1. Cambiare il messaggio

Per cambiare il messaggio dal default "Attendere prego..." a qualcosa di differente, si imposti il valore di `Seam.Remoting.loadingMessage`:

```
Seam.Remoting.loadingMessage = "Loading...";
```

25.10.2. Nascondere il messaggio di caricamento

To completely suppress the display of the loading message, override the implementation of `displayLoadingMessage()` and `hideLoadingMessage()` with functions that instead do nothing:

```
// non mostrare l'indicatore di caricamento
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

25.10.3. A Custom Loading Indicator

It is also possible to override the loading indicator to display an animated icon, or anything else that you want. To do 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
 };
```

25.11. 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
}
```

25.11.1. 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();
```

25.11.2. 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();
```

25.11.3. 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();
```

25.11.4. Combining Constraints

Constraints can also be combined, to filter objects from multiple paths within the object graph:

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

25.12. Richieste transazionali

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);
}
```

25.13. Messaggistica JMS

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.

25.13.1. Configurazione

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"/>
```

25.13.2. Iscriversi ad un Topic JMS

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).

25.13.3. Disiscriversi da un Topic

To unsubscribe from a topic, call `Seam.Remoting.unsubscribe()` and pass in the topic name:

```
Seam.Remoting.unsubscribe("topicName");
```

25.13.4. 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:

```
// Attendere 1 secondo tra la ricezione della risposta del pool e l'invio della successiva richiesta
 di pool.
Seam.Remoting.pollInterval = 1;

// Attendere fino a 5 secondi sul server per nuovi messaggi
Seam.Remoting.pollTimeout = 5;
```

Seam e il Google Web Toolkit

Per chi preferisce utilizzare Google Web Toolkit (GWT) per sviluppare applicazioni AJAX dinamiche, Seam fornisce uno strato di integrazione che consente ai componenti GWT di interagire direttamente con i componenti Seam.

Per usare GWT, si suppone che ci sia già una certa familiarità con gli strumenti GWT. Maggiori informazioni si possono trovare in *http://code.google.com/webtoolkit/*. Questo capitolo non cercherà di spiegare come funziona GWT o come si usa.

26.1. Configurazione

Non è richiesta una particolare configurazione per usare GWT in una applicazione Seam, ad ogni modo la servlet delle risorse Seam deve essere installata. Vedi *Capitolo 30, Configurare Seam ed impacchettare le applicazioni Seam* per i dettagli.

26.2. Preparare i componenti

Il primo passo per preparare i componenti Seam che saranno richiamati tramite GWT è di creare sia l'interfaccia per il servizio sincrono che quella per il servizio asincrono per i metodi che si desidera chiamare. Entrambe queste interfacce devono estendere l'interfaccia GWT `com.google.gwt.user.client.rpc.RemoteService`:

```
public interface MyService extends RemoteService {
   public String askIt(String question);
}
```

L'interfaccia asincrona deve essere identica, salvo che contiene un parametro aggiuntivo `AsyncCallback` per ciascuno dei metodi che dichiara:

```
public interface MyServiceAsync extends RemoteService {
  public void askIt(String question, AsyncCallback callback);
}
```

L'interfaccia asincrona, `MyServiceAsync` in questo esempio, sarà implementata da GWT e non dovrà mai essere implementata direttamente.

Il passo successivo è di creare un componente Seam che implementa l'interfaccia sincrona:

```
@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);
  }
}
```

Il nome del componente Seam *deve* corrispondere al nome completo dell'interfaccia client GWT (come illustrato), altrimento la servlet delle risorse non riuscirà a trovarlo quando un client farà una chiamata GWT. I metodi che si vogliono rendere accessibili tramite GWT devono anche essere annotati con l'annotazione `@WebRemote`.

26.3. Collegare un componente GWT ad un componente Seam

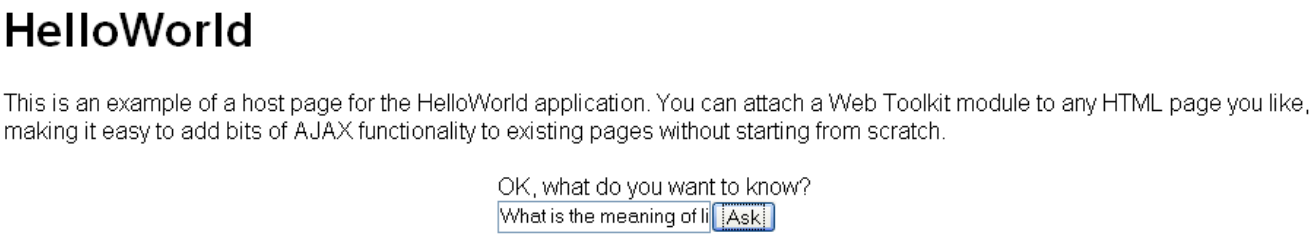
Il prossimo passo è scrivere un metodo che restituisce l'interfaccia asincrona al componente. Questo metodo può essere posizionato all'interno della classe del componente GWT e sarà usato da questo per ottenere un riferimento al client di collegamento asincrono:

```
private MyServiceAsync getService() {
  String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

  MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
  ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
  return svc;
}
```

Il passo finale è scrivere il codice del componente GWT che invoca il metodo sul client di collegamento. Il seguente esempio definisce una semplice interfaccia utente con una label, un campo di input e un bottone:

```
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);
      }
    });
  }

  ...
```

Facendo click, il bottone invoca il metodo `askServer()` passando il contenuto del campo input (in questo esempio viene fatta anche una validazione per assicurare che il testo inserito sia una domanda valida). Il metodo `askServer()` acquisisce un riferimento al client di collegamento asincrono (restituito dal metodo `getService()`) e invoca il metodo `askIt()`. Il risultato (o il messaggio di errore se la chiamata fallisce) è mostrato in una finestra di segnalazione.



Il codice completo di questo esempio si trova nella distribuzione Seam sotto la cartella `examples/remoting/gwt`.

26.4. Target Ant per GWT

Per eseguire applicazioni GWT c'è un passaggio di compilazione da Java a JavaScript (che compatta il codice e lo rende illeggibile). C'è uno strumento Ant che può essere utilizzato al posto della linea di comando o dello strumento grafico fornito con GWT. Per usarlo occorre avere il jar con lo strumento Ant nel classpath di Ant, e anche l'ambiente GWT scaricato (che sarebbe comunque necessario per far eseguire il componente in modalità hosted).

Quindi nel file Ant va scritto (verso l'inizio del file Ant):

```
<taskdef uri="antlib:de.samaflost.gwttasks"
  resource="de/samaflost/gwttasks/antlib.xml"
  classpath="./lib/gwttasks.jar"/>

  <property file="build.properties"/>
```

Creare un file `build.properties`, con il seguente contenuto:

```
gwt.home=/gwt_home_dir
```

Questo naturalmente deve puntare alla cartella dove si è installato GWT. A questo punto per usarlo creare il target:

```
<!-- Le seguenti sono istruzioni pratiche per lo sviluppo in GWT.
  Per usare GWT occorre aver scaricato GWT separatamente -->
  <target name="gwt-compile">
    <!-- in questo caso stiamo riposizionando la roba generata da gwt, perci# in questo caso
   possiamo avere solo un modulo GWT - facciamo in questo modo appositamente per mantenere
breve l'URL -->
    <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
>
```

Quando viene chiamato, questo target compila l'applicazione GWT e la copia nella cartella indicata (che dovrebbe trovarsi nella parte `webapp` del war. Si ricordi che GWT genera oggetti HTML e JavaScript). Il codice risultante dalla generazione con `gwt-compile` non va mai modificato. Le modifiche vanno fatte sulla cartella dei sorgenti GWT.

Si tenga presente che GWT contiene un browser per la modalità hosted che andrebbe utilizzato se si sviluppa con GWT. Non utilizzandolo e compilando ogni volta, si rinuncia alla maggior parte del toolkit (in effetti a chi non può o non vuole usare il browser in modalità hosted, sarebbe da dirgli che NON dovrebbe usare GWT per niente, dato che quella è la cosa migliore!).

Integrazione con il framework Spring

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

27.1. 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.com/products/seam/spring-seam"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                  http://jboss.com/products/seam/spring-seam
                  http://jboss.com/products/seam/spring-seam-2.1.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*)

27.2. 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...

27.3. 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.

27.4. 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
>
```

27.5. 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.

27.6. 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.SeamManagedEntityManagerFactoryBean">
        <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.SeamManagedEntityManagerFactoryBean">
        <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.SeamManagedEntityManagerFactoryBean">
        <property name="persistenceContextName" value="entityManager"/>
      <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
<bean
 class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor">
      <property name="defaultPersistenceUnitName" value="bookingDatabase:extended"/>
</bean
>
```

The `component.xml` might look like:

```
<persistence:managed-persistence-context name="entityManager"
      auto-create="true" entity-manager-factory="#{entityManagerFactory}"/>
```

`JpaTemplate` and `JpaDaoSupport` 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
>
```

27.7. 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`.

27.8. 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.com/products/seam/components"
        xmlns:spring="http://jboss.com/products/seam/spring"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jboss.com/products/seam/components
                    http://jboss.com/products/seam/components-2.1.xsd
                    http://jboss.com/products/seam/spring
                    http://jboss.com/products/seam/spring-2.1.xsd">

    <spring:context-loader config-locations="/WEB-INF/applicationContext.xml"/>

</components
>
```

27.9. 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}"/>
```

Integrazione con Guice

Google Guice è una libreria che fornisce una dependency injection leggera attraverso la risoluzione type-safe. L'integrazione con Guice (parte del modulo Seam IoC) consente l'uso dell'iniezione Guice per tutti i componenti Seam annotati con l'annotazione `@Guice`. In aggiunta alla regolare bijection, fornita da Seam (che diviene opzionale), Seam delega agli injector Guice noti di soddisfare le dipendenze del componente. Guice può essere utile per legare parti non-Seam di applicazione grosse o legacy assieme a Seam.

Nota

L'integrazione Guice è messa nella libreria jboss-seam-ioc. Questa dipendenza è richiesta per tutte le tecniche di integrazione coperta in questo capitolo. Occorre anche il file JAR Guice nel classpath.

28.1. Creazione di un componente ibrido Seam-Guice

L'obiettivo è creare un componente ibrido Seam-Guice. La regola per come realizzare ciò è molto semplice. Se si vuole usare l'injection Guice nel proprio componente Seam, annotarlo con l'annotazione `@Guice` (dopo l'importazione del tipo `org.jboss.seam.ioc.guice.Guice`).

```
@Name("myGuicyComponent")
@Guice public class MyGuicyComponent
{
   @Inject MyObject myObject;
   @Inject @Special MyObject mySpecialObject;
   ...
}
```

Quest'iniezione Guice avverrà ad ogni chiamata di metodo, come con la bijection. Guice inietta in base a tipo e binding. Per soddisfare le dipendenze nel precedente esempio, si possono associare le seguenti implementazioni in un modulo Guice, dove `@Special` è un'annotazione definita nella propria applicazione.

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

Bene, ma quanle injector Guice verrà usato per iniettare le dipendenze? Occorre fare prima qualche settaggio.

28.2. Configurare un injector

Dire a Seam quale injector Guice usare agganciandolo alla proprietà injection del componente Guice di inizializzazione nel descrittore del componente Seam (components.xml):

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:guice="http://jboss.org/products/seam/guice"
  xsi:schemaLocation="
    http://jboss.com/products/seam/guice
    http://jboss.com/products/seam/guice-2.1.xsd
    http://jboss.com/products/seam/components
    http://jboss.com/products/seam/components-2.1.xsd">

  <guice:init injector="#{myGuiceInjector}"/>

</components
>
```

`myGuiceInjector` deve risolvere ad un componente Seam che implementi l'interfaccia Guice `Injector`.

Tuttavia dover creare un injector è pura scrittura di codice. Ciò che si vuole essere in grado di fare è semplicemente agganciare Seam ai propri moduli Guice. Fortunamente c'è un componente Seam predefinito che implementa l'interfaccia `Injector` per fare ciò. Si può configurarlo nel descrittore del componente Seam con il seguente codice.

```
<guice:injector name="myGuiceInjector">
  <guice:modules
>
    <value
>com.example.guice.GuiceModule1</value
>
    <value
```

```
>com.example.guice.GuiceModule2</value
>
  </guice:modules
>
</guice:injector
>
```

Certamente si può anche usare un injector che viene già usato in un'altra parte anche non-Seam della proria applicazione. Questa è una delle principali motivazioni per creare quest'integrazione. Poiché l'injector è definito con un'espressione EL, si può ottenerlo in un qualsiasi modo si voglia. Per esempio si può usare il pattern del componente factory di Seam per fornire l'injector.

```
@Name("myGuiceInjectorFactory")
public InjectorFactory
{
  @Factory(name = "myGuiceInjector", scope = APPLICATION, create = true)
  public Injector getInjector()
  {
    // Your code that returns injector
  }
}
```

28.3. Uso di injector multipli

Di default viene usato un injector configurato nel descrittore di componente Seam. Se occorre usare più injector (in alternativa, si possono anche usare più moduli), si può specificare un injector per ogni componente Seam nell'annotazione `@Guice`.

```
@Name("myGuicyComponent")
@Guice("myGuiceInjector")
public class MyGuicyComponent
{
  @Inject MyObject myObject;
  ...
}
```

Ecco tutto! Si controlli l'esempio guice nella distribuzione Seam per vedere in azione l'integrazione Guice!

Hibernate Search

29.1. Introduzione

Motori di ricerca full text come Apache Lucene™ sono una potentissima tecnologia che fornisce alle applicazioni efficienti query full text. Hibernate Search, che utilizza Apache Lucene al suo interno, indicizza il modello di dominio con l'aggiunta di alcune annotazioni, si prende cura del database, della sincronizzazione degli indici e restituisce oggetti regolari gestiti che corrispondono alle query full text. Tuttavia si tenga presente che ci sono delle irregolarità quando si ha a che fare con un modello di dominio a oggetti sopra ad un indice di testo (mantenere l'indice aggiornato, irregolarità tra la struttura dell'indice ed il modello di dominio, e irregolarità di query). Ma i benefici di velocità ed efficienza superano di gran lunga queste limitazioni.

Hibernate Search è stato progettato per integrarsi nel modo più naturale possibile con JPA ed Hibernate. Essendo una naturale estensione, JBoss Seam fornisce l'integrazione con Hibernate Search.

Si prega di riferirsi alla *documentazione Hibernate Search* [http://www.hibernate.org/hib_docs/search/reference/en/html_single/] per informazioni sul progetto Hibernate Search.

29.2. Configurazione

Hibernate Search è configurato in uno dei file `META-INF/persistence.xml` o `hibernate.cfg.xml`.

La configurazione di Hibernate Search ha dei valori di default impostati per la maggior parte dei parametri di configurazione. Ecco qua una configurazione minimale di persistence unit per poter iniziare.

```
<persistence-unit name="sample">
  <jta-data-source
>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- utilizza un file system basato su indice -->
    <property name="hibernate.search.default.directory_provider"
      value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory dove gli indici verranno memorizzati -->
    <property name="hibernate.search.default.indexBase"
      value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit
>
```

Se si pensa di usare Hibernate Annotation o EntityManager 3.2.x (incorporato in JBoss AS 4.2.x e successivi), occorre configurare anche gli opportuni event listener.

```
<persistence-unit name="sample">
  <jta-data-source
>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- utilizza un file system basato su indice -->
    <property name="hibernate.search.default.directory_provider"
          value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory dove gli indici verranno memorizzati -->
    <property name="hibernate.search.default.indexBase"
          value="/Users/prod/apps/dvdstore/dvdindexes"/>

    <property name="hibernate.ejb.event.post-insert"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-update"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-delete"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>

  </properties>
</persistence-unit
>
```

Nota

Questo passo non è più necessario se vengono utilizzati Hibernate Annotation o EntityManager 3.3.x.

In aggiunta al file di configurazione, devono essere deployati i seguenti jar:

- hibernate-search.jar
- hibernate-commons-annotations.jar
- lucene-core.jar

Nota

Se vengono messi in un EAR, non si dimentichi di aggiornare `application.xml`

29.3. Utilizzo

Hibernate Search impiega annotazioni per mappare le entità ad un indice di Lucene, per maggiori informazioni si guardi alla *documentazione di riferimento* [http://www.hibernate.org/hib_docs/search/reference/en/html_single/].

Hibernate Search è pienamente integrato con la API e la semantica di JPA/Hibernate. Passare da una query basata su HQL o Criteria richiede solo poche linee di codice. La principale API con cui l'applicazione interagisce è l'API `FullTextSession` (sottoclasse della `Session` di Hibernate).

Quando Hibernate Search è presente, JBoss Seam inietta una `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();
  }
  [...]
}
```

Nota

`FullTextSession` estende `org.hibernate.Session` così che può essere usata come una regolare Hibernate Session.

Se viene usata la Java Persistence API, è proposta un'integrazione più lieve.

```
@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();
  }
  [...]
}
```

Quando Hibernate Search è presente, viene iniettato un `FulltextEntityManager`. `FullTextEntityManager` estende `EntityManager` con metodi specifici di ricerca, allo stesso modo `FullTextSession` estende `Session`.

Quando viene impiegata l'iniezione di un session bean EJB3.0 o message driven (cioè tramite l'annotazione @PersistenceContext), non è possibile rimpiazzare l'interfaccia `EntityManager` con l'interfaccia `FullTextEntityManager` nella dichiarazione. Comunque l'implementazione iniettata sarà `FullTextEntityManager`: è quindi possibile il downcasting.

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable {

  @PersistenceContext EntityManager em;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    FullTextEntityManager ftEm = (FullTextEntityManager) em;
    javax.persistence.Query query = ftEm.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
        .setMaxResults(pageSize + 1)
        .setFirstResult(pageSize * currentPage)
        .getResultList();
  }
  [...]
}
```

Attenzione

Per persone abituate a Hibernate Seam fuori da Seam, notare che l'uso di `Search.createFullTextSession` non è necessario.

Controlla gli esempi DVDStore o Blog della distribuzione di JBoss Seam per un uso pratico di Hibernate Search.

Configurare Seam ed impacchettare le applicazioni Seam

La configurazione è un argomento molto noioso ed un passatempo estremamente tedioso. Sfortunatamente sono richieste parecchie linee di XML per integrare Seam con l'implementazione JSF ed il servlet container. Non c'è alcun bisogno di soffermarsi sulle seguenti sezioni; non si dovrà mai scrivere queste cose a mano, poiché basta usare seam-gen per iniziare l'applicazione oppure basta copiare ed incollare il codice dagli esempi di applicazione!

30.1. Configurazione base di Seam

In primo luogo si comincia a guardare alla configurazione base che server ogni volta che si usa Seam con JSF.

30.1.1. Integrazione di Seam con JSF ed il servlet container

Certamente occorre un servlet faces!

```
<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
>
```

(Si può sistemare il pattern dell'URL a proprio piacimento.)

In aggiunta, Seam richiede la seguente voce nel file `web.xml`:

```
<listener>
   <listener-class
```

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

Questo listener è responsabile dell'avvio di Seam e della distruzione dei contesti di sessione e di applicazione.

Alcune implementazioni JSF hanno un'implementazione erronea della conservazione dello stato lato server, che interferisce con la propagazione della conversazione di Seam. Se si hanno problemi con la propagazione della conversazione durante l'invio di form, si provi a cambiare impostanto la conservazione lato client. Occorre impostare questo in `web.xml`:

```
<context-param>
   <param-name
>javax.faces.STATE_SAVING_METHOD</param-name>
   <param-value
>client</param-value>
</context-param
>
```

C'è una piccola parte grigia nella specifica JSF riguardante la mutabilità dei valore dello stato della vista. Poiché Seam usa lo stato della vista JSF per tornare allo scope PAGE, questo in alcuni casi può diventare un problema. Se si usa la conservazione dello stato lato server con JSF-RI e si vuole che il bean con scope PAGE mantenga l'esatto valore per una data vista di pagina, occorre specificare il seguente parametro di contesto. Altrimenti se un utente usa il pulsante "indietro" un componente con scope PAGE avrà l'ultimo valore se questo non è cambiato nella pagina "indietro". (si veda *Spec Issue* [https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=295]). Quest'impostazione non è abilitata di default a causa della performance nella serializzazione della vista JSF ad ogni richiesta.

```
<context-param>
   <param-name
>com.sun.faces.serializeServerState</param-name>
   <param-value
>true</param-value>
</context-param
>
```

30.1.2. Usare Facelets

Se si vuole seguire il consiglio ed usare Facelets invece di JSP, si aggiungano le seguenti linee a `faces-config.xml`:

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

E le seguenti linee a `web.xml`:

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

Se si usa facelets in JBoss AS, si noterà che il logging di Facelets è guasto (i messaggi di log non appaiono nel log del server). Seam fornisce un modo per sistemare questo problema, per usarlo si copi `lib/interop/jboss-seam-jul.jar` in `$JBOSS_HOME/server/default/deploy/jboss-web.deployer/jsf-libs/` e si includa `jboss-seam-ui.jar` nel `WEB-INF/lib` dell'applicazione. Le categorie di logging di Facelets sono elencate nella *Documentazione per lo sviluppatore Facelets* [https://facelets.dev.java.net/nonav/docs/dev/docbook.html#config-logging].

30.1.3. Resource Servlet di Seam

Seam Resource Servlet fornisce le risorse usate da Seam Remoting, captchas (si veda il capitolo sulla sicurezza) ed altri controlli JSF UI. Configurare Seam Resource Servlet richiede la seguente riga 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
>
```

30.1.4. Filtri servlet di Seam

Seam non ha bisogno di filtri servlet per le operazioni base. Comunque ci sono parecchie caratteristiche che dipendono dall'uso dei filtri. Per facilitare le cose, Seam lascia aggiungere e configurare i filtri servlet così come si configurano gli altri componenti predefiniti di Seam. Per sfruttare questa caratteristica occorre installa un filtro master 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
>
```

Il filtro master di Seam *deve* essere il primo filtro specificato in `web.xml`. Questo assicura che venga eseguito per primo.

I filtri Seam condividono un numero di attributi comuni, si possono impostare questi in `components.xml` in aggiunto ai parametri discussi sotto:

- `url-pattern` — Usato per specificare quali richieste vengono filtrate, il default è tutte le richieste. `url-pattern` è un pattern di stile di Tomcat che consente un suffisso wildcard.
- `regex-url-pattern` — Usato per specificare quali richieste vengono filtrate, il default è tutte le richieste. `regex-url-pattern` è una vera espressione regolare per il percorso di richiesta.
- `disabled` — Usato per disabilitare il filtro predefinito.

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.

30.1.4.1. Gestione delle eccezioni.

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.com/products/seam/components"
        xmlns:web="http://jboss.com/products/seam/web">

   <web:exception-filter url-pattern="*.seam"/>

</components
>
```

30.1.4.2. Propagazione delle conversazioni con i redirect

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"/>
```

30.1.4.3. Riscrittura dell'URL

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`.

30.1.4.4. Invio di form multipart

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).

30.1.4.5. Codifica dei caratteri

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` — La codifica da usare.
- `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`.

30.1.4.6. 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.

30.1.4.7. Log delle identità

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.com/products/seam/components"
        xmlns:web="http://jboss.com/products/seam/web">
   <web:logging-filter url-pattern="*.seam"/>
</components
>
```

30.1.4.8. 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`.

30.1.4.9. 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 thar 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")`).

30.1.5. 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. Unfortunately, there is no standard mapping to global JNDI defined in the EJB3 specification, so this mapping is vendor-specific (and may depend on your own naming conventions as well). We usually specify this option in `components.xml`.

For JBoss AS, the following pattern is correct:

```
<core:init jndi-name="earName/#{ejbName}/local" />
```

In this case, `earName` is the name of the EAR in which the bean is deployed, Seam replaces `#{ejbName}` with the name of the EJB, and the final segment represents the type of interface (local or remote).

Outside the context of an EAR (when using the JBoss Embeddable EJB3 container), the first segment is dropped since there is no EAR, leaving us with the following pattern:

```
<core:init jndi-name="#{ejbName}/local" />
```

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.

The folks at JBoss don't care much for XML, if you can't tell. So when they designed JBoss AS, they decided that EJB components would get assigned a global JNDI name automatically, using the pattern just described (i.e., EAR name/EJB name/interface type). 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
- Il semplice nome della classe bean

Guardiamo un esempio. Si assuma di avere definiti il seguente bean EJB ed un'interfaccia.

```
package com.example.myapp;

import javax.ejb.Local;

@Local
public class Authenticator
{
   boolean authenticate();
}

package com.example.myapp;

import javax.ejb.Stateless;

@Stateless
@Name("authenticator")
public class AuthenticatorBean implements Authenticator
{
   public boolean authenticate() { ... }
}
```

Assuming 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/local</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 aggrevating 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 compoenent 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`.

That covers how JNDI names are defined and used. The lesson is that with some application servers, such as GlassFish, you are going to have to specify JNDI names for all EJB components explicitly, and sometimes twice! And even if you are following the same naming convention as JBoss AS, the JNDI pattern in Seam may need to be altered. For instance, the global JNDI names are automatically prefixed with java:comp/env on GlassFish, so you need to define the JNDI pattern as follows:

```
<core:init jndi-name="java:comp/env/earName/#{ejbName}/local" />
```

Finally, let's talk about transactions. In an EJB3 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/>
```

30.1.6. Non dimenticare!

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.

30.2. Uso di provider JPA alternativi

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.

Questo è un 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 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:

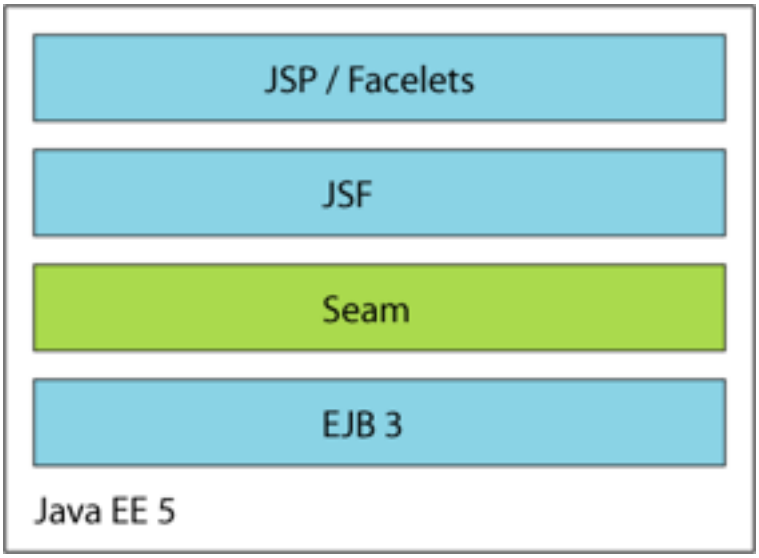
```
<component name="org.jboss.seam.persistence.persistenceProvider"
        class="org.jboss.seam.persistence.PersistenceProvider"
        scope="stateless">
</component
>
```

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.

30.3. Configurazione di Seam in java EE 5



If you're running in a Java EE 5 environment, this is all the configuration required to start using Seam!

30.3.1. 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
  my-application.war/
    META-INF/
      MANIFEST.MF
    WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
        jsf-facelets.jar
        jboss-seam-ui.jar
    login.jsp
    register.jsp
    ...
  my-application.jar/
    META-INF/
      MANIFEST.MF
      persistence.xml
    seam.properties
    org/
      jboss/
        myapplication/
          User.class
          Login.class
          LoginBean.class
          Register.class
          RegisterBean.class
          ...
```

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 facelets (our recommendation), you must include `jsf-facelets.jar` in the `WEB-INF/lib` directory of the WAR.

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`.

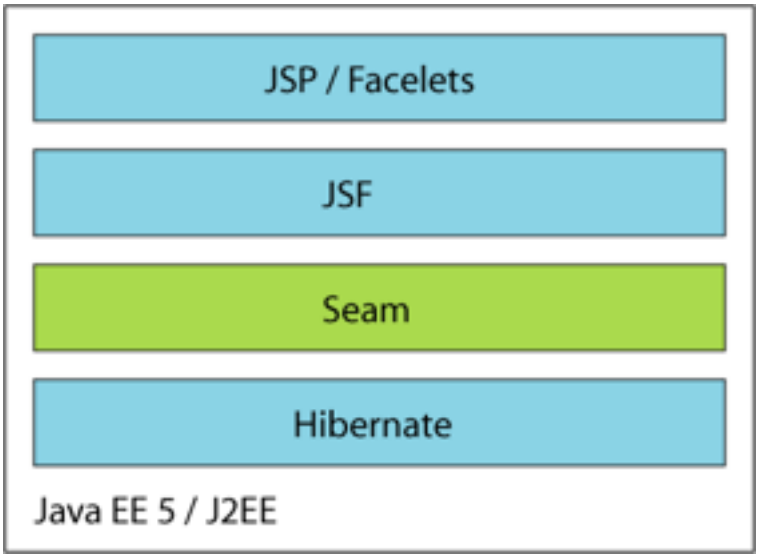
If you want to use the Seam debug page (only works for applications using facelets), you must include `jboss-seam-debug.jar` in the `WEB-INF/lib` directory of the WAR.

Seam ships with several example applications that are deployable in any Java EE container that supports EJB 3.0.

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.

30.4. Configurare Seam in J2EE

Seam is useful even if you're not yet ready to take the plunge into EJB 3.0. In this case you would use Hibernate3 or JPA instead of EJB 3.0 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 3.0 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 Hibernate3 and JavaBeans instead of EJB3, and another version that uses JPA and JavaBeans. These example applications are ready to deploy into any J2EE application server.

30.4.1. Boostrapping di 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}"/>
```

30.4.2. Boostrapping di 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}"/>
```

30.4.3. Packaging

We can package our application as a WAR, in the following structure:

```
my-application.war/
   META-INF/
      MANIFEST.MF
   WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
         jboss-seam.jar
         jboss-seam-ui.jar
         jboss-el.jar
         jsf-facelets.jar
         hibernate3.jar
         hibernate-annotations.jar
         hibernate-validator.jar
         ...
         my-application.jar/
```

```
        META-INF/
          MANIFEST.MF
        seam.properties
        hibernate.cfg.xml
        org/
          jboss/
            myapplication/
              User.class
              Login.class
              Register.class
              ...
  login.jsp
  register.jsp
  ...
```

If we want to deploy Hibernate in a non-EE environment like Tomcat or TestNG, we need to do a little bit more work.

30.5. Configurazione di Seam in java EE 5 senza JBoss Embedded

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}"/>
```

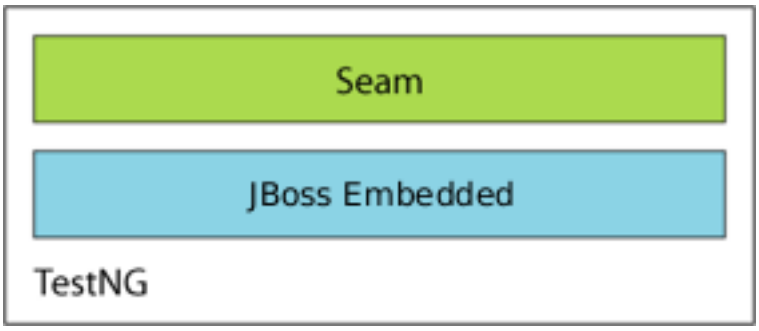
Of course, you'll also need to define a datasource.

A better alternative is to use JBoss Embedded to get access to the EE APIs.

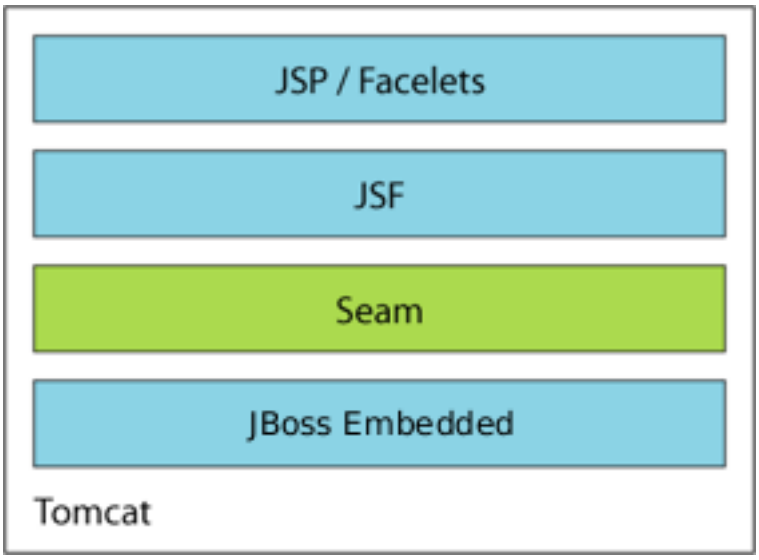
30.6. Configurazione di Seam in java EE 5 con JBoss Embedded

JBoss Embedded lets you run EJB3 components outside the context of the Java EE 5 application server. This is especially, but not only, useful for testing.

The Seam booking example application includes a TestNG integration test suite that runs on JBoss Embedded via `SeamTest`.



The booking example application may even be deployed to Tomcat.



30.6.1. Installare JBoss Embedded

Embedded JBoss must by installed into Tomcat for Seam applications to run correctly on it. Embedded JBoss runs with JDK 5 or JDK 6 (see *Sezione 41.1, «Dipendenze JDK»* for details on using JDK 6). Embedded JBoss can be downloaded *here* [http://sourceforge.net/project/showfiles.php?group_id=22866&package_id=228977]. The process for installing Embedded JBoss into Tomcat 6 is quite simple. First, you should copy the Embedded JBoss JARs and configuration files into Tomcat.

- Copy all files and directories under the Embedded JBoss `bootstrap` and `lib` directories, except for the `jndi.properties` file, into the Tomcat `lib` directory.
- Remove the `annotations-api.jar` file from the Tomcat `lib` directory.

Next, two configuration files need to be updated to add Embedded JBoss-specific functionality.

- Add the Embedded JBoss listener `EmbeddedJBossBootstrapListener` to `conf/server.xml`. It must appear after all other listeners in the file:

```
<Server port="8005" shutdown="SHUTDOWN">

  <!-- Comment these entries out to disable JMX MBeans support used for the
       administration web application -->
  <Listener className="org.apache.catalina.core.AprLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.storeconfig.StoreConfigLifecycleListener" />
```

```
  <!-- Add this listener -->
  <Listener className="org.jboss.embedded.tomcat.EmbeddedJBossBootstrapListener" />
```

- WAR file scanning should be enabled by adding the `WebinfScanner` listener to `conf/context.xml`:

```
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource
>WEB-INF/web.xml</WatchedResource>

    <!-- Uncomment this to disable session persistence across Tomcat restarts -->
    <!--
    <Manager pathname="" />
    -->
```

```
  <!-- Add this listener -->
  <Listener className="org.jboss.embedded.tomcat.WebinfScanner" />
```

```
</Context
>
```

- If you are using Sun JDK 6, you need to set the Java option `sun.lang.ClassLoader.allowArraySyntax` to `true` in the JAVA_OPTS environment variable used by the Catalina startup script (catalina.bat on Windows or catalina.sh on Unix).

Open the script appropriate for your operating system in a text editor. Add a new line immediately below the comments at the top of the file where you will define the JAVA_OPTS environment variable. On Windows, use the following syntax:

```
set JAVA_OPTS=%JAVA_OPTS% -Dsun.lang.ClassLoader.allowArraySyntax=true
```

On Unix, use this syntax instead:

```
JAVA_OPTS="$JAVA_OPTS -Dsun.lang.ClassLoader.allowArraySyntax=true"
```

For more configuration options, please see the Embedded JBoss Tomcat integration *wiki entry* [http://wiki.jboss.org/wiki/Wiki.jsp?page=EmbeddedAndTomcat].

30.6.2. Packaging

The archive structure of a WAR-based deployment on an servlet engine like Tomcat will look something like this:

```
my-application.war/
   META-INF/
      MANIFEST.MF
   WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
         jboss-seam.jar
         jboss-seam-ui.jar
         jboss-el.jar
         jsf-facelets.jar
         jsf-api.jar
         jsf-impl.jar
         ...
         my-application.jar/
            META-INF/
               MANIFEST.MF
               persistence.xml
            seam.properties
            org/
               jboss/
                  myapplication/
```

```
                User.class
                Login.class
                LoginBean.class
                Register.class
                RegisterBean.class
                ...
  login.jsp
  register.jsp
  ...
```

Most of the Seam example applications may be deployed to Tomcat by running `ant deploy.tomcat`.

30.7. Configurazione 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.

30.7.1. 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-3.1.jar
    META-INF/
       MANIFEST.MF
       application.xml
    my-application.war/
       META-INF/
          MANIFEST.MF
       WEB-INF/
          web.xml
          components.xml
```

```
        faces-config.xml
        lib/
            jsf-facelets.jar
            jboss-seam-ui.jar
    login.jsp
    register.jsp
    ...
  my-application.jar/
    META-INF/
        MANIFEST.MF
        persistence.xml
    seam.properties
    org/
        jboss/
            myapplication/
                User.class
                Login.class
                LoginBean.class
                Register.class
                RegisterBean.class
                ...
  jbpm.cfg.xml
  hibernate.cfg.xml
  createDocument.jpdl.xml
  editDocument.jpdl.xml
  approveDocument.jpdl.xml
  documentLifecycle.jpdl.xml
```

30.8. Configurazione di SFSB e dei timeout di sessione in JBoss AS

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 `server/default/conf/standardjboss.xml` (replace *default* with your own configuration).

The default SFSB timeout can be adjusted by modifying the value of `max-bean-life` in the `LRUStatefulContextCachePolicy` cache configuration:

```
<container-cache-conf>
   <cache-policy
>org.jboss.ejb.plugins.LRUStatefulContextCachePolicy</cache-policy>
   <cache-policy-conf>
```

```
      <min-capacity
>50</min-capacity>
      <max-capacity
>1000000</max-capacity>
      <remover-period
>1800</remover-period>

      <!-- SFSB timeout in seconds; 1800 seconds == 30 minutes -->
      <max-bean-life
>1800</max-bean-life
>

      <overager-period
>300</overager-period>
      <max-bean-age
>600</max-bean-age>
      <resizer-period
>400</resizer-period>
      <max-cache-miss-period
>60</max-cache-miss-period>
      <min-cache-miss-period
>1</min-cache-miss-period>
      <cache-load-factor
>0.75</cache-load-factor>
   </cache-policy-conf>
</container-cache-conf
>
```

The default HTTP session timeout can be modified in `server/default/deploy/jbossweb-tomcat55.sar/conf/web.xml` for JBoss 4.0.x, or in `server/default/deploy/jboss-web.deployer/conf/web.xml` for JBoss 4.2.x or later. The following entry in this file controls the default session timeout for all web applications:

```
<session-config>
   <!-- HTTP Session timeout, in minutes -->
   <session-timeout
>30</session-timeout>
</session-config
>
```

To override this value for your own application, simply include this entry in your application's own `web.xml`.

30.9. Esecuzione di Seam in un 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.

30.10. 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) {
     // ...
  }
}
```

Annotazioni di Seam

Quando si scrive un'applicazione Seam vengono impiegate molte annotazioni. Seam ti consente di usare annotazioni per ottenere uno stile dichiarativo di programmazione. La maggior parte delle annotazioni che si useranno sono definite dalla specifica EJB3.0. Le annotazioni per la validazione dei dati sono definite dal pacchetto Hibernate Validator. Infine Seam definisce un suo set di annotazioni che verranno descritte in questo capitolo.

Tutte queste annotazioni sono definite nel pacchetto `org.jboss.seam.annotations`.

31.1. Annotazioni per la definizione di un componente

Il primo gruppo di annotazioni consente di definire un componente Seam. Queste annotazioni appaiono sulla classe del componente.

`@Name`

```
@Name("componentName")
```

Definisce il nome del componente Seam per una classe. Quest'annotazione è richiesta per tutti i componenti Seam.

`@Scope`

```
@Scope(ScopeType.CONVERSATION)
```

Definisce il contesto di default del componente. I possibili valori sono definiti dalla enumeration `ScopeType`: `EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS, APPLICATION, STATELESS`.

Quando non viene specificato nessuno scope, quello di default dipende dal tipo di componente. Per session bean stateless, il default è `STATELESS`. Per gli entity bean ed i session bean stateful, il default è `CONVERSATION`. Per i JavaBean, il default è `EVENT`.

`@Role`

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

Consente ad un componente Seam di venir associato a variabili di contesto multiple. Le annotazioni `@Name`/`@Scope` definiscono un "ruolo di default". Ciascuna annotazione `@Role` definisce un ruolo addizionale.

- `nome` — il nome della variabile di contesto.
- `scope` — lo scope della variabile di contesto. Quando non viene specificato esplicitamente alcuno scope, il default dipende dal tipo di componente, come sopra.

`@Roles`

```
@Roles({
      @Role(name="user", scope=ScopeType.CONVERSATION),
      @Role(name="currentUser", scope=ScopeType.SESSION)
  })
```

Consente la specificazione di ruoli multipli addizionali.

`@BypassInterceptors`

```
@BypassInterceptors
```

Disabilita tutti gli interceptor di Seam per un particolare componente o metodo di un componente.

`@JndiName`

```
@JndiName("my/jndi/name")
```

Specifica il nome JNDI che Seam userà per la ricerca del componente EJB. Se non è stato specificato esplicitamente alcun nome, Seam userà il pattern JNDI specificato da `org.jboss.seam.core.init.jndiPattern`.

`@Conversational`

```
@Conversational
```

Specifica che il componente con scope conversazione è conversazionale, il che significa che nessun metodo del componente potrà essere chiamato amenoché sia attiva una conversazione long-running.

`@PerNestedConversation`

```
@PerNestedConversation
```

Limita lo scope di un componente con scope conversazione alla sola conversazione padre in cui è stato istanziato. L'istanza del componente non sarà visibile alle conversazioni figlie innestate, che otterranno la propria istanza.

Attenzione: questo è mal definito, poiché implica che un componente sarà visibile per alcune parti del ciclo di richiesta, ed invisibile dopo questo. Non è raccomandato che un'applicazioni usi questa caratteristica!

`@Startup`

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

Specifica che un componente con scope applicazione venga avviato immediatamente in fase di inizializzazione. E' usato principalmente per certi componenti predefiniti che avviano un'infrastruttura critica, quali JNDI, datasource, ecc.

```
@Scope(SESSION) @Startup
```

Specifica che un componente con scope di sessione viene avviato immediatamente alla creazione della sessione.

- `depends` — specifica che i componenti con nome vengano avviati per primi, se sono installati.

`@Install`

```
@Install(false)
```

Specifica se oppure no, un componente debba essere installato di default. La mancanza dell'annotazione `@Install` indica che un componente viene installato.

```
@Install(dependencies="org.jboss.seam.bpm.jbpm")
```

Specifica che un componente debba essere installato solo se i componenti elencati come dipendenze sono pure installati.

```
@Install(genericDependencies=ManagedQueueSender.class)
```

Specifica che un componente debba essere installato solo se un componente che è implementato da una certa classe è installato. Questo è utile quando la dipendenza non ha un singolo nome ben noto.

```
@Install(classDependencies="org.hibernate.Session")
```

Specifica che un componente debba essere installato solo se la classe (con nome) è nel classpath.

```
@Install(precedence=BUILT_IN)
```

Specifica la precedenza del componente. Se esistono più componenti con lo stesso nome, verrà installato quello con precedenza più alta. I valori definiti di precedenza sono (in ordine ascendente):

- `BUILT_IN` — Precedenza su tutti i componenti Seam predefiniti
- `FRAMEWORK` — Precedenza per l'uso di componenti di framework che estendono Seam
- `APPLICATION` — Precedenza ai componenti di applicazione (precedenza di default)
- `DEPLOYMENT` — Precedenza all'uso di componenti che fanno override dei componenti di applicazione in un particolare deploy
- `MOCK` — Precedenza per oggetti mock usati nei test

`@Synchronized`

```
@Synchronized(timeout=1000)
```

Specifica che un componente venga acceduto in modo concorrente da più client, e che Seam serializzi le richieste. Se una richiesta non è in grado di ottenere il lock sul componente in un determinato periodo di timeout, viene sollevata un'eccezione.

`@ReadOnly`

```
@ReadOnly
```

Specifica che un componente JavaBean o metodo di componente non richieda la replicazione dello stato alla fine dell'invocazione.

`@AutoCreate`

```
@AutoCreate
```

Specifica che un componente verrà automaticamente creato, anche se il client non specifica `create=true`.

31.2. Annotazioni per la bijection

Le prossime due annotazioni controllano la bijection. Questi attributi vengono impiegati nelle variabili d'istanza di un componente o nei metodi di accesso alle proprietà.

`@In`

```
@In
```

Specifica che un attributo di componente debba essere iniettato da una variabile di contesto all'inizio di ogni invocazione del componente. Se la variabile di contesto è null, viene lanciata un'eccezione.

```
@In(required=false)
```

Specifica che un attributo di componente debba essere iniettato da una variabile di contesto all'inizio di ogni invocazione del componente. La variabile di contesto può essere null.

```
@In(create=true)
```

Specifica che un attributo di componente debba essere iniettato da una variabile di contesto all'inizio di ogni invocazione del componente. Se la variabile di contesto è null, viene istanziata da Seam un'istanza del componente.

```
@In(value="contextVariableName")
```

Specifica il nome della variabile di contesto in modo esplicito, invece di usare il nome annotato della variabile d'istanza.

```
@In(value="#{customer.addresses['shipping']}")
```

Specifica che un attributo di componente debba essere iniettato valutando un'espressione JSF EL all'inizio di ogni invocazione del componente.

- `value` — specifica il nome di una variabile di contesto. Di default è il nome dell'attributo del componente. In alternativa, specifica un'espressione JSF EL, racchiusa da `#{...}`.
- `create` — specifica che Seam debba istanziare il componente con lo stesso nome della variabile di contesto se questa è indefinita (null) in tutti i contesti. Di default è false.
- `required` — specifica che Seam lanci un'eccezione se la variabile di contesto è indefinita in tutti i contesti.

`@Out`

```
@Out
```

Specifica che un attributo di componente (di Seam) venga messo in outjection nella sua variabile di contesto alla fine dell'invocazione. Se l'attributo è null, viene lanciata un'eccezione.

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

Specifica che un attributo di componente (di Seam) venga messo in outjection nella sua variabile di contesto alla fine dell'invocazione. L'attributo può essere null.

```
@Out(scope=ScopeType.SESSION)
```

Specifica che un attributo di componente che *non* è un tipo di componente di Seam venga messo in outjection in uno specifico scope alla fine dell'invocazione.

In alternativa, se non è specificato alcuno scope in modo esplicito, viene usato lo scope del componente con l'attributo `@Out` (o lo scope `EVENT` se il componente è stateless).

```
@Out(value="contextVariableName")
```

Specifica il nome della variabile di contesto in modo esplicito, invece di usare il nome annotato della variabile d'istanza.

- `value` — specifica il nome della variabile di contesto. Di default è il nome dell'attributo del componente.
- `required` — specifica che Seam debba lanciare un'eccezione se l'attributo del componente è null durante l'outjection.

Si noti che è piuttosto comune l'uso di queste annotazioni assieme, per esempio:

```
@In(create=true) @Out private User currentUser;
```

La prossima annotazione supporta il pattern *manager component*, dove un componente Seam che gestisce il ciclo di vita di un'istanza di qualche altra classe deve essere iniettata. Appare sul metodo getter del componente.

`@Unwrap`

```
@Unwrap
```

Specifica che l'oggetto ritornato dal metodo getter annotato è la cosa che viene iniettata invece dell'istanza stessa del componente.

La prossima annotazione supporta il pattern *factory component* dove un componente Seam è responsabile dell'inizializzazione del valore di una variabile di contesto. Questo è utile in particolare per inizializzare qualsiasi stato occorrente per il rendering della risposta in una richiesta non-faces. Appare su un metodo di componente.

`@Factory`

```
@Factory("processInstance") public void createProcessInstance() { ... }
```

Specifica che il metodo del componente viene usato per inizializzare il valore della variabile di contesto con nome, quando la variabile di contesto non ha valore. Questo stile viene usato con metodi che ritornano `void`.

```
@Factory("processInstance", scope=CONVERSATION) public ProcessInstance
createProcessInstance() { ... }
```

Specifica che il metodo restituisce un valore che Seam dovrebbe usare per inizializzare il valore della variabile di contesto con nome, quando la variabile di contesto non ha valore. Questo stile è usato con metodi che restituiscono un valore. Se non viene specificato esplicitamente alcuno scope, viene usato lo scope del componente con il metodo `@Factory` (amenoché il componente sia stateless, nel qual caso viene usato il contesto `EVENT`).

- `value` — specifica il nome della variabile di contesto. Se il metodo è un getter, di default è il nome della proprietà del JavaBean.

- `scope` — specifica lo scope che Seam dovrebbe associare al valore restituito. E' significativo solo per i metodi factory che restituiscono un valore.
- `autoCreate` — specifica che questo metodo factory dovrebbe automaticamente essere chiamato quando si chiama la variabile, anche se `@In` non specifica `create=true`.

Questa annotazione consente di iniettare un `Log`:

`@Logger`

```
@Logger("categoryName")
```

Specifica che un campo del componente deve essere iniettato con un istanza di `org.jboss.seam.log.Log`. Per gli entity bean, il campo deve essere dichiarato statico.

- `value` — specifica il nome della categoria di log. Di default è il nome della classe del componente.

L'ultima annotazione consente di iniettare un valore di un parametro di richiesta:

`@RequestParameter`

```
@RequestParameter("parameterName")
```

Specifica che un attributo di componente deve essere iniettato con il valore di un parametro di richiesta. Le conversioni del tipo base sono eseguite automaticamente.

- `value` — specifica il nome del parametro di richiesta. Di default è il nome dell'attributo del componente.

31.3. Annotazioni per i metodi del ciclo di vita dei componenti

Queste annotazioni consentono al componente di reagire agli eventi del proprio ciclo di vita. Accadono sul metodo del componente. Possono essere solo una per ogni classe di componente.

`@Create`

```
@Create
```

Specifica che il metodo venga chiamato quando viene istanziata da Seam un'istanza del componente. Si noti che i metodi di creazioni sono supportati solo per JavaBeans e bean di sessione stateful.

`@Destroy`

```
@Destroy
```

Specifica che il metodo deve essere chiamato quando il contesto termina e le sue variabili vengono distrutte. Si noti che i metodi distruttori sono supportati solo per JavaBeans e bean di sessione stateful.

I metodi distruttori devono essere usati solo per la pulizia. *Seam cattura, logga ed ignora (swallow) ogni eccezione che si propaga da un metodo distruttore.*

`@Observer`

```
@Observer("somethingChanged")
```

Specifica che il metodo deve essere chiamato quando avviene un evento component-driven del tipo specificato.

```
@Observer(value="somethingChanged",create=false)
```

Specifica che il metodo venga chiamato quando accade un evento del tipo specificato, ma che non venga creata un'istanza se non esiste. Se un'istanza non esiste e create è false, l'evento non verrà osservato. Di default il valore di create è true.

31.4. Annotazioni per la demarcazione del contesto

Queste annotazioni forniscono una demarcazione dichiarativa della conversazione. Appaiono sui metodi dei componenti Seam, solitamente metodi action listener.

Ogni richiesta web ha un contesto di conversazione associato ad essa. La maggior parte di queste conversazioni finisce alla fine della richesta. Se si vuole che una conversazione si espanda lungo richieste multiple, si deve "promuovere" la conversazione corrente a *conversazione long-running* chiamando un metodo marcato con `@Begin`.

`@Begin`

```
@Begin
```

Specifica che la conversazione long-running inizia quando questo metodo restuisce un esito non-null senza eccezioni.

```
@Begin(join=true)
```

Specifica che se una conversazione long-running è già in esecuzione, il contesto della conversazione viene semplicemente propagato.

```
@Begin(nested=true)
```

Specifica che se una conversazione ong-running è già in esecuzione, inizia un nuovo contesto di conversazione *innestata*. La conversazione innestata terminerà quando viene incontrato il successivo `@End`, e verrà ripristinata la conversazione più esterna. E' perfettamente legale che esistano conversazioni innestate concorrenti nella stessa conversazione più esterna.

```
@Begin(pageflow="process definition name")
```

Specifica il nome della definizione di processo jBPM che definisce il pageflow per questa conversazione.

```
@Begin(flushMode=FlushModeType.MANUAL)
```

Specifica la modalità di flush di un qualsiasi contesto di persistenza gestito da Seam. `flushMode=FlushModeType.MANUAL` supporta l'uso di *conversazioni atomiche* dove tutte le operazioni di scrittura vengono accodate nel contesto di conversazione fino ad una chiamata esplicita al metodo `flush()` (che solitamente avviene alla fine della conversazione).

- `join` — determina il comportamento quando una conversazione long-running è già attiva. Se `true`, il contesto viene propagato. Se `false`, viene lanciata un'eccezione. Di default è `false`. Quest'impostazione è ignorata quando è specificato `nested=true`.
- `nested` — specifica che una conversazione innestata sia avviata se è già presente una conversazione long-running.
- `flushMode` — imposta la modalità flush di ogni sessione Hibernate gestita da Seam o contesto di persistenza JPA che vengono creati durante la conversazione.
- `pageflow` — un nome di definizione di processo jBPM deployato via `org.jboss.seam.bpm.jbpm.pageflowDefinitions.`

`@End`

```
@End
```

Specifica che una conversazione long-running termina quando questo metodo restituisce un esito non-null senza eccezioni.

- `beforeRedirect` — di default la conversazione non verrà effettivamente distrutta prima che avvenga un redirect. Impostando `beforeRedirect=true` viene specificato che la conversazione debba essere distrutta alla fine della richiesta corrente, e che il redirect venga processato in un contesto di conversazione temporanea.
- `root` — di default terminare una conversazione innestata implica che venga tolta la conversazione dallo stack delle conversazioni e venga ripristinata la conversazione più esterna. `root=true` specifica che la conversazione radice debba essere distrutta, e ciò distrugge effettivamente l'intero stack delle conversazioni. Se la conversazione non è innestata, la conversazione semplicemente finisce.

`@StartTask`

```
@StartTask
```

"Inizia" un task jBPM. Specifica che una conversazione long-running inizi quando questo metodo restituisce un esito non-null senza eccezione. Questa conversazione è associata ad un task jBPM specificato nel parametro della richiesta. Dentro il contesto di questa conversazione è definito anche il contesto del processo di business per l'istanza del processo di business dell'istanza task.

- La `TaskInstance` di jBPM sarà disponibile in una variabile di contesto di richiesta chiamato `taskInstance`. La `ProcessInstance` di jBPM sarà disponibile in una variabile di contesto di richiesta chiamata `processInstance`. (Sicuramente questo oggetti sono disponibili per l'iniezione via `@In`.)
- `taskIdParameter` — il nome del parametro di richiesta che mantiene l'id del task. Di default è `"taskId"`, che è anche il default usato dal componente Seam JSF `taskList`.
- `flushMode` — imposta la modalità flush di ogni sessione Hibernate gestita da Seam o contesto di persistenza JPA che vengono creati durante la conversazione.

`@BeginTask`

```
@BeginTask
```

Ripristina il lavoro su un task jBPM incompleto. Specifica che la conversazione long-running inizi quando questo metoto restituisce un esito non-null senza eccezioni. Questa conversazione è associata al task jBPM specificata nel parametro di richiesta. Dentro il contesto di questa conversazione, è definito anche un contesto di business process per l'istanza del processo di business dell'istanza task.

- La `org.jbpm.taskmgmt.exe.TaskInstance` jBPM sarà disponibile in una variabile di contesto richiesta chiamata `taskInstance`. La `org.jbpm.graph.exe.ProcessInstance` jBPM sarà disponibile in una variabile di contesto richiesta chiamata `processInstance`.
- `taskIdParameter` — il nome del parametro di richiesta che mantiene l'id del task. Di default è `"taskId"`, che è anche il default usato dal componente Seam JSF `taskList`.
- `flushMode` — imposta la modalità flush di ogni sessione Hibernate gestita da Seam o contesto di persistenza JPA che vengono creati durante la conversazione.

`@EndTask`

```
@EndTask
```

"Termina" un task jBPM. Specifica che una conversazione long-running termina quando il metodo ritorna un esito non-null, e che il task corrente è completo. Lancia una transizione jBPM. La transizione lanciata sarà la transizione di default amenoché l'applicazione chiami `Transition.setName()` sul componente predefinito chiamato `transition`.

```
@EndTask(transition="transitionName")
```

Lancia la transizione jBPM data.

- `transition` — il nome della transizione jBPM da lanciare come trigger quando termina il task. Il default è la transizione di default.
- `beforeRedirect` — di default la conversazione non verrà effettivamente distrutta prima che avvenga un redirect. Impostando `beforeRedirect=true` viene specificato che la conversazione debba essere distrutta alla fine della richiesta corrente, e che il redirect venga processato in un contesto di conversazione temporanea.

`@CreateProcess`

```
@CreateProcess(definition="process definition name")
```

Crea una nuova istanza di processo jBPM quando il metodo restituisce un esito non-null senza eccezione. L'oggetto `ProcessInstance` sarà disponibile in una variabile di contesto chiamata `processInstance`.

- `definition` — il nome di una definizione di processo jBPM deployata via `org.jboss.seam.bpm.jbpm.processDefinitions`.

`@ResumeProcess`

```
@ResumeProcess(processIdParameter="processId")
```

Reinserisce lo scope di un'istanza esistente di processo jBPM quando il metodo ritorna un esisto non-null senza eccezione. L'oggetto `ProcessInstance` sarà disponibile in una variabile di contesto chiamata `processInstance`.

- `processIdParameter` — il nome del parametro di richiesta che che mantiene l'id di processo. Di default è `"processId"`.

`@Transition`

```
@Transition("cancel")
```

Marca un metodo come segnalante una transizione nell'istanza del processo jBPM corrente quando il metodo restituisce un risultato non-null.

31.5. Annotazioni per l'uso con i componenti JavaBean di Seam in ambiente J2EE

Seam fornisce un'annotazione che consente di forza un rollback di una transazione JTA per certi esiti action listener.

`@Transactional`

```
@Transactional
```

Specifica che un componente JavaBean debba avere un comportamento transazionale simile al comportamenteo di default di un componente session bean, cioèle invocazioni di metodo deve avere luogo in una transazione, e se non esiste alcuna transazione quando viene chiamato il metodo, verrà iniziata una transazione solo per quel metodo. Quest'annotazione può essere applicata o a livello di classe o di metodo.

Non usare quest'annotazione in componenti EJB 3.0, usare `@TransactionAttribute`!

`@ApplicationException`

```
@ApplicationException
```

Sinonimo di javax.ejb.ApplicationException, per l'uso in un ambiente per Java EE 5. Si applica ad un'eccezione per denotare che è un'eccezione di applicazione e deve essere riportata direttamente al client (cioè, unwrapped).

Non usare quest'annotazione in componenti EJB 3.0, usare invece `@javax.ejb.ApplicationException`.

- `rollback` — di default `false`, se `true` quest'eccezione deve impostare la transazione al solo roolback.
- `end` — di default `false`, se `true` quest'eccezione deve terminare la conversazione long-running corrente.

`@Interceptors`

```
@Interceptors({DVDInterceptor, CDInterceptor})
```

Sinonimo di javax.interceptors.Interceptors, per l'uso in ambiente pre Java EE 5. Si noti che può essere usato solo come meta-annotazione. Dichiara una lista ordinata di interceptor per una classe o un metodo.

Non usare queste annotazioni su componenti EJB 3.0, usare invece `@javax.interceptor.Interceptors`.

Queste annotazioni sono utili per i componenti Seam JavaBean. Se si usano i componenti EJB 3.0, occorre usare l'annotazione standard Java EE 5.

31.6. Annotazioni per le eccezioni

Queste annotazioni consentono di specificare come Seam debba gestire un'eccezione che si propaga fuori da un componente Seam.

`@Redirect`

```
@Redirect(viewId="error.jsp")
```

Specifica che un'eccezione annotata causi un redirect del browser verso uno specifico id di vista.

- `viewId` — specifica che l'di della vista JSF a cui fare il redirect. Si può usare EL.
- `message` — un messaggio da mostrare, di default è il messaggio d'eccezione.
- `end` — specifica che la conversazione long-running debba terminare, di default è `false`.

`@HttpError`

```
@HttpError(errorCode=404)
```

Specifica che l'eccezione annotata debba causare l'invio di un errore HTTP.

- `errorCode` — il codice d'errore HTTP, di default è `500`.
- `message` — un messaggio deve essere inviato come errore HTTP, di default è il messaggio d'eccezione.
- `end` — specifica che la conversazione long-running debba terminare, di default è `false`.

31.7. Annotazioni per Seam Remoting

Seam Remoting richiede che l'interfaccia locale di un session bean sia annotato con la seguente annotazione:

`@WebRemote`

```
@WebRemote(exclude="path.to.exclude")
```

Indica che il metodo annotato può essere chiamato da JavaScript lato client. La proprietà `exclude` è opzionale e consente agli oggetti di essere esclusi dal grafo dell'oggetto dei risultati (vedere il capitolo *Capitolo 25, Remoting* per maggiori dettagli).

31.8. Annotazioni per gli interceptor di Seam

Nelle classi interceptor di Seam appaiono le seguenti annotazioni.

Si prega di fare riferimento alla documentazione della specifica EJB 3.0 per informazioni sulle annotazioni richieste per la definizione di interceptor EJB.

`@Interceptor`

```
@Interceptor(stateless=true)
```

Specifica che quest'interceptor è stateless e Seam può ottimizzare la replicazione.

```
@Interceptor(type=CLIENT)
```

Specifica che quest'interceptor è un interceptor "lato client" che viene chiamato prima del container EJB.

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

Specifica che quest'interceptor è posizionato più in alto nello stack rispetto agli interceptor dati.

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

Specifica che quest'interceptor è posizionato più in profondità nello stack rispetto agli interceptor dati.

31.9. Annotazioni per l'asincronicità

Le seguenti annotazioni vengono usate per dichiarare un metodo asincrono, per esempio:

```
@Asynchronous public void scheduleAlert(Alert alert, @Expiration Date date) { ... }
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert,
  @Expiration Date date,
  @IntervalDuration long interval) { ... }
```

`@Asynchronous`

```
@Asynchronous
```

Specifica che la chiamata al metodo viene processata asicronicamente.

`@Duration`

```
@Duration
```

Specifica che un parametro della chiamata asincrona è la durata prima che la chiamata venga processata (o la prima processata per le chiamate ricorrenti).

`@Expiration`

```
@Expiration
```

Specifica che un parametro della chiamata asincrona è la dataora prima che la chiamata venga processata (o la prima processata per le chiamate ricorrenti).

`@IntervalDuration`

```
@IntervalDuration
```

Specifica che una chiamata di un metodo asincrono si ripete ed il parametro annotato è la durata tra le ricorrenze.

31.10. Annotazioni per l'uso di JSF

Queste annotazioni rendono più facile lavorare con JSF.

`@Converter`

Consente al componente Seam di agire come convertitore JSF. La classe annotata deve essere un componente Seam e deve implementare `javax.faces.convert.Converter`.

- `id` — l'id del convertitore JSF. Di default è il nome del componente.
- `forClass` — se specificato, registra questo componente come convertitore di default per un tipo.

`@Validator`

Consente ad un componente Seam di agire come validatore JSF. La classe annotata deve essere un componente Seam, e deve implementare `javax.faces.validator.Validator`.

- `id` — l'id del validatore JSF. Di default è il nome del componente.

31.10.1. Annotazioni per l'uso con `dataTable`

Le seguenti annotazioni facilitano l'implementazione di liste cliccabili con dietro un bean di sessione stateful. Appaiono sugli attributi.

`@DataModel`

```
@DataModel("variableName")
```

Fa l'outjection di una proprietà di tipo `List`, `Map`, `Set` o `Object[]` come un `DataModel` JSF nello scope del componente proprietario (o nello scope `EVENT` se il componente proprietario è `STATELESS`). In caso di `Map`, ogni riga di `DataModel` è una `Map.Entry`.

- `value` — nome della variabile di contesto della conversazione. Di default è il nome dell'attributo.
- `scope` — se `scope=ScopeType.PAGE` è specificato esplicitamente, il `DataModel` verrà mantenuto nel contesto `PAGE`.

`@DataModelSelection`

```
@DataModelSelection
```

Inietta il valore selezionato dal `DataModel` JSF (questo è l'elemento della collezione sottostante, o valore di mappa). Se è definito solo un attributo `@DataModel` per un componente, verrà iniettato il valore selezionato da quel `DataModel`. Altrimenti, il nome del componente di ciascun `DataModel` dovrà essere specificato nell'attributo di valore per ogni `@DataModelSelection`.

Se sul `@DataModel` associato è specificato lo scope `PAGE`, allora in aggiunta alla DataModel Selection iniettata, verrà iniettato anche il DataModel associato. In questo caso, se la proprietà annotata con `@DataModel` è un metodo getter, allora il metodo setter per la proprietà deve essere parte della Business API del componente Seam che lo contiene.

- `value` — nome della variabile di contesto della conversazione. Non occorre se c'è esattamente un `@DataModel` nel componente.

`@DataModelSelectionIndex`

```
@DataModelSelectionIndex
```

Espone l'indice della selezione del `DataModel` JSF come un attributo del componente (questo è il numero riga della collezione sottostante, o la chiave della mappa). Se per un componente è definito solo un attributo `@DataModel`, verrà iniettato il valore selezionato da quel `DataModel`. Altrimenti il nome del componente di ciascun `@DataModel` deve essere specificato nell'attributo valore per ogni `@DataModelSelectionIndex`.

- `value` — nome della variabile di contesto della conversazione. Non occorre se c'è esattamente un `@DataModel` nel componente.

31.11. Meta-annotationi per il databinding

Queste meta-annotazioni rendono possibile implementare una funzionalità simile in `@DataModel` e `@DataModelSelection` per altre strutture di dati oltre alla lista.

`@DataBinderClass`

```
@DataBinderClass(DataModelBinder.class)
```

Specifica che è un'annotazione di databinding.

`@DataSelectorClass`

```
@DataSelectorClass(DataModelSelector.class)
```

Specifica che è un'annotazione di dataselection.

31.12. Annotazioni per i pacchetti

Questa annotazione fornisce un meccaniscmo per dichiarare informazioni riguardanti un set di componente impacchettati assieme. Può essere applicata a qualsiasi pacchetto Java.

`@Namespace`

```
@Namespace(value="http://jboss.com/products/seam/example/seampay")
```

Specifica che i componenti nel pacchetto corrente sono associati al namespace dato. Il namespace dichiarato deve essere usato come namespace XML in un file `components.xml` per semplificare la configurazione dell'applicazione.

```
@Namespace(value="http://jboss.com/products/seam/core", prefix="org.jboss.seam.core")
```

Specifica un namespace da associare al pacchetto dato. In aggiunta, specifica un prefisso al nome del componente da applicare ai nomi componenti specificati nel file XML. Per esempio, un elemento XML chiamato `init`, associato a questo namespace, dovrebbe fare riferimento al componente chiamato `org.jboss.seam.core.init`.

31.13. Annotazioni per l'integrazione con un servlet container

Queste annotazioni consentono di integrare i componenti Seam con un servlet container.

`@Filter`

Si usi come filtro servlet il componente Seam (che implementa `javax.servlet.Filter`) annotato con `@Filter`. Esso verrà eseguito come filtro master di Seam.

```
@Filter(around={"seamComponent", "otherSeamComponent"})
```

Specifica che questo filtro è posizionato più in alto nello stack rispetto agli altri filtri.

```
@Filter(within={"seamComponent", "otherSeamComponent"})
```

Specifica che questo filtro è posizionato più in basso nello stack rispetto agli altri filtri.

Componenti Seam predefiniti

Questo capitolo descrive i componenti predefiniti di Seam e le loro proprietà di configurazione. I componenti predefiniti verranno creati anche se non sono elencati nel file `components.xml`, ma occorre fare l'override delle proprietà di default o specificare più di un compoenente di un certo tipo, viene usato `components.xml`.

Si noti che si può sostituire uno dei componenti predefiniti con le proprie implementazioni semplicemente specificando il nome di uno dei componenti predefiniti nella propria classe usando `@Name`.

32.1. Componenti per l'iniezione del contesto

Il primo set di componenti predefiniti esiste solamente per supportare l'injection di vari oggetti contestuali. Per esempio, la seguente variabile d'istanza di componente vedrebbe iniettato l'oggetto del contesto di sessione di Seam:

```
@In private Context sessionContext;
```

`org.jboss.seam.core.contexts`
: Componente che fornisce accesso agli oggetti del contesto di Seam, per esempio `org.jboss.seam.core.contexts.sessionContext['user']`.

`org.jboss.seam.faces.facesContext`
: Componente manager per l'oggetto del contesto `FacesContext` (non un vero contesto di Seam)

Tutti questi componenti vengono sempre installati.

32.2. Componenti JSF

Il seguente set di componenti è fornito come supplemento JSF.

`org.jboss.seam.faces.dateConverter`
: Fornisce un converter JSF di default per le proprietà di tipo `java.util.Date`.

  Questo converter è automaticamente registrato con JSF. E' fornito per risparmiare allo sviluppatore il dover specificare un DateTimeConverter su un campo d'input o su un parametro di pagina. Di default, si assume che il tipo sia una data (in opposto a tempo o tempo e data) ed usa lo stile d'input short corretto con il locale dell'utente. Per Locale.US, il pattern d'input è mm/DD/yy. Comunque in accordo con Y2K, l'anno è cambiato da due cifre a quattro (es. mm/DD/yyyy).

E' possibile eseguire l'override del pattern d'input in modo globale, usando la configurazione dei componenti. Per vedere alcuni esempi consultare la JavaDoc per questa classe.

`org.jboss.seam.faces.facesMessages`
Consentono ai messaggi faces di successo di essere propagati lungo i redirect del browser.

- `add(FacesMessage facesMessage)` — aggiunge un messaggio faces, che verrà mostrato durante la prossima fase di render response che avviene nella conversazione corrente.
- `add(String messageTemplate)` — aggiunge un messaggio faces, generato dal template di messaggio che può contenere espressioni EL.
- `add(Severity severity, String messageTemplate)` — aggiunge un messaggio faces, generato dal template di messaggio che può contenere espressioni EL.
- `addFromResourceBundle(String key)` — aggiunge un messaggio faces, generato dal template di messaggio che può contenere espressioni EL.
- `addFromResourceBundle(Severity severity, String key)` — aggiunge un messaggio faces, generato dal template di messaggio definito nel resource bundle di Seam e che può contenete espressioni EL.
- `clear()` — pulisce tutti i messaggi.

`org.jboss.seam.faces.redirect`
Un'API per l'esecuzione dei redirect con parametri (utile specialmente per le schermate con risultati di ricerca memorizzabili come segnalibro).

- `redirect.viewId` — l'id della vista JSF a cui fare il redirect.
- `redirect.conversationPropagationEnabled` — determina se la conversazione verrà propagata assieme al redirect.
- `redirect.parameters` — una mappa di nomi di parametri di richiesta da valorizzare, da passare alla richiesta di redirect.
- `execute()` — esegue immediatamente il redirect.
- `captureCurrentRequest()` — memorizza l'id della vista ed i parametri di richiesta della richesta GET corrente (nel contesto conversazione), per poi essere usato chiamando `execute()`.

`org.jboss.seam.faces.httpError`
Un'API per l'invio di errori HTTP.

`org.jboss.seam.ui.renderStampStore`
Un componente (di default con scope sessione) responsabile del mantenimento della collezione di stampe da renderizzare. Una stampa da renderizzare è un indicatore che mostra se una form renderizzata è stata inviata. Questa memorizzazione è utile quando viene il

metodo JSF di salvataggio dello stato lato client, poiché determina se la form è stata mandata sotto il controllo del server anziché nell'albero dei componenti che viene mantenuto sul client.

Per disassociare questo check dalla sessione (che è uno dei principali obiettivi di design del salvataggio di stato lato client) deve essere fornita un'implementazione che memorizzi le stampe da renderizzare nell'applicazione (valida a lungo quanto l'esecuzione dell'applicazione) o il database (valido fino al riavvio del server).

- `maxSize` — Il numero massimo di stampe da mantenere in memoria. Default: 100

Tutti questi componenti vengono installati quando la classe `javax.faces.context.FacesContext` è disponibile nel classpath.

32.3. Componenti d'utilità

Questi componenti sono molto utili.

`org.jboss.seam.core.events`
Un'API per sollevare eventi che possono essere osservati tramite i metodi `@Observer`, o binding di metodo in `components.xml`.

- `raiseEvent(String type)` — solleva un evento di un tipo particolare e lo distribuisce a tutti gli osservatori.
- `raiseAsynchronousEvent(String type)` — solleva un evento da processare in modo asincrono da parte del servizio timer di EJB3.
- `raiseTimedEvent(String type, ....)` — schedula un evento da processare in modo asincrono dal servizio timer di EJB3.
- `addListener(String type, String methodBinding)` — aggiunge un observer per un particolare tipo di evento.

`org.jboss.seam.core.interpolator`
Un'API per interpolare i valori di espressioni EL JSF in String.

- `interpolate(String template)` — analizza il template per espressioni EL JSF della forma `#{...}` e li sostituisce con i loro valori valutati.

`org.jboss.seam.core.expressions`
Un'API per creare binding di valore e di metodo.

- `createValueBinding(String expression)` — crea un oggetto per il binding di valore.
- `createMethodBinding(String expression)` — crea un oggetto per il binding di metodo.

`org.jboss.seam.core.pojoCache`
Componente manager per un'istanza `PojoCache` di JBoss Cache.

- `pojoCache.cfgResourceName` — il nome del file di configurazione. Di default è `treecache.xml`.

Tutti questi componenti vengono sempre installati.

32.4. Componenti per l'internazionalizzazione ed i temi

Il prossimo gruppo di componenti semplifica la costruzione di interfacce utente internazionalizzate usando Seam.

`org.jboss.seam.core.locale`
: Il locale di Seam.

`org.jboss.seam.international.timezone`
: La timezone di Seam. La timezone ha scope di sessione.

`org.jboss.seam.core.resourceBundle`
: Il resource bundle di Seam. Il resource bundle è stateless. Il resource bundle di Seam esegue una ricerca delle chiavi in una lista di resource bundle di Java.

`org.jboss.seam.core.resourceLoader`
: Il resource loader fornisce accesso alle risorce dell'applicazione ed ai resource bundle.

  - `resourceLoader.bundleNames` — i nomi dei resource bundle da cerca quando viene usato il resource bundle di Seam. Di default è `messages`.

`org.jboss.seam.international.localeSelector`
: Supporta la selezione del locale o a configuration time, o dall'utente a runtime.

  - `select()` — seleziona il locale specificato.
  - `localeSelector.locale` — l'attuale `java.util.Locale`.
  - `localeSelector.localeString` — la rappresentazione in stringa del locale.
  - `localeSelector.language` — il linguaggio per il locale specificato.
  - `localeSelector.country` — il paese del locale specificato.
  - `localeSelector.variant` — la variante del locale specificato.
  - `localeSelector.supportedLocales` — una lista di `SelectItem`che rappresenta il locale supportati elencati in `jsf-config.xml`.
  - `localeSelector.cookieEnabled` — specifica che la selezione del locale debba essere memorizzata in un cookie.

`org.jboss.seam.international.timezoneSelector`
: Supporta la selezione della timezone o a configuration time o da parte dell'utente a runtime.

- `select()` — seleziona il locale specificato.
- `timezoneSelector.timezone` — la `java.util.TimeZone` corrente.
- `timezoneSelector.timeZoneId` — la rappresentazione in stringa della timezone.
- `timezoneSelector.cookieEnabled` — specificare che la selezione della timezone debba essere memorizzata in un cookie.

`org.jboss.seam.international.messages`
: Una mappa contenente messaggi internazionalizzati generati da template di messaggi definiti nel resource bundle di Seam.

`org.jboss.seam.theme.themeSelector`
: Supporta la selezione di temi o a configuration time, o da parte dell'utente a runtime.

- `select()` — seleziona il tema specificato.
- `theme.availableThemes` — la lista dei temi definiti.
- `themeSelector.theme` — il tema selezionato.
- `themeSelector.themes` — una lista di `SelectItem` che rappresentano il temi definiti.
- `themeSelector.cookieEnabled` — specifica che la selezione del tema sia memorizzata in un cookie.

`org.jboss.seam.theme.theme`
: Una mappa contenente delle entry di temi.

Tutti questi componenti vengono sempre installati.

32.5. Componenti per il controllo delle conversazioni.

Il prossimo gruppo di componenti consente il controllo delle conversazioni da parte dell'applicazione o dell'interfaccia utente.

`org.jboss.seam.core.conversation`
: API per il controllo dell'applicazione degli attributi della conversazione di Seam.

- `getId()` — restituisce l'id della conversazione corrente
- `isNested()` — l'attuale conversazione è una conversazione annidata?
- `isLongRunning()` — l'attuale conversazione è una conversazione long-running?
- `getId()` — restituisce l'id della conversazione corrente
- `getParentId()` — restituisce l'id della conversazione padre
- `getRootId()` — restituisce l'id della conversazione radice

- `setTimeout(int timeout)` — imposta il timeout della conversazione attuale
- `setViewId(String outcome)` — imposta l'id della vista da usare quando si torna alla conversazione corrente dallo switcher di conversazione, dalla lista di conversazioni o dai breadcrumbs.
- `setDescription(String description)` — imposta la descrizione della conversazione corrente da mostrare nello switcher di conversazione, nella lista di conversazioni o nel breadcrumbs.
- `redirect()` — redirige all'ultimo id di vista ben-definito per questa conversazione (utile dopo i login).
- `leave()` — esce dallo scope di questa conversazione, senza terminare la conversazione.
- `begin()` — inizia una conversazione long-running (equivalente a `@Begin`).
- `beginPageflow(String pageflowName)` — inizia una conversazione long-running con un pageflow (equivalente a `@Begin(pageflow="...")`).
- `end()` — termina una conversazione long-running (equivalente a `@End`).
- `pop()` — estrae dallo stack di conversazione, restituendo la conversazione padre.
- `root()` — ritorna la conversazione root dello stack di conversazioni.
- `changeFlushMode(FlushModeType flushMode)` — cambia la modalità flush della conversazione.

`org.jboss.seam.core.conversationList`
: Componente gestore per la lista delle conversazioni.

`org.jboss.seam.core.conversationStack`
: Componente gestore per lo stack delle conversazioni (breadcrumbs).

`org.jboss.seam.faces.switcher`
: Lo switcher di conversazione.

Tutti questi componenti vengono sempre installati.

32.6. Componenti per jBPM

Questi componenti sono usati con jBPM.

`org.jboss.seam.pageflow.pageflow`
: Controllo API dei pageflow di Seam.

- `isInProcess()` — ritorna `true` se c'è un pageflow nel processo
- `getProcessInstance()` — restituisce una `ProcessInstance` jBPM per il pageflow corrente

- `begin(String pageflowName)` — inizia un pageflow nel contesto della conversazione corrente
- `reposition(String nodeName)` — riposiziona il pageflow corrente in un particolare nodo

`org.jboss.seam.bpm.actor`
API per il controllo dell'applicazione degli attributi dell'attore jBPM associato alla sessione corrente.

- `setId(String actorId)` — imposta l'id dell'attore jBPM dell'utente corrente.
- `getGroupActorIds()` — ritorna un `Set` a cui gli id degli attori jBPM per i gruppi degli utenti correnti possono essere aggiunti.

`org.jboss.seam.bpm.transition`
API per il controllo dell'applicazione della transizione jBPM per il task corrente.

- `setName(String transitionName)` — imposta il nome della transizione jBPM da usare quando il task corrente viene terminato tramite `@EndTask`.

`org.jboss.seam.bpm.businessProcess`
API per il controllo programmatico dell'associazione tra la conversazione ed il processo di business.

- `businessProcess.taskId` — l'id del task associato alla conversazione corrente.
- `businessProcess.processId` — l'id del processo associato alla conversazione corrente.
- `businessProcess.hasCurrentTask()` — l'istanza del task è associata alla conversazione corrente?
- `businessProcess.hasCurrentProcess()` — l'istanza del processo è associata alla conversazione corrente?
- `createProcess(String name)` — crea un'istanza della definizione del processo (con nome) e l'associa alla conversazione corrente.
- `startTask()` — avvia il task associato alla conversazione corrente.
- `endTask(String transitionName)` — termina il task associato alla conversazione corrente.
- `resumeTask(Long id)` — associa il task con l'id dato alla conversazione corrente.
- `resumeProcess(Long id)` — associa il processo con l'id dato alla conversazione corrente.
- `transition(String transitionName)` — avvia una transizione.

`org.jboss.seam.bpm.taskInstance`
Componente gestore per la `TaskInstance` jBPM.

`org.jboss.seam.bpm.processInstance`
: Componente gestore per la `ProcessInstance` jBPM.

`org.jboss.seam.bpm.jbpmContext`
: Componente gestore per il `JbpmContext` con scope evento.

`org.jboss.seam.bpm.taskInstanceList`
: Manager component for the jBPM task list.

`org.jboss.seam.bpm.pooledTaskInstanceList`
: Componente gestore per la lista di task pooled jBPM.

`org.jboss.seam.bpm.taskInstanceListForType`
: Componente gestore per la lista di task jBPM.

`org.jboss.seam.bpm.pooledTask`
: Action handler per l'assegnamento del task pooled.

`org.jboss.seam.bpm.processInstanceFinder`
: Manager per la lista di task delle istanze di processo.

`org.jboss.seam.bpm.processInstanceList`
: La lista dei task delle istanze di processo.

Tutti questi componenti vengono installati quando viene installato il componente `org.jboss.seam.bpm.jbpm`.

32.7. Componenti per la sicurezza

Questi componenti si relazionano alla sicurezza a livello web.

`org.jboss.seam.web.userPrincipal`
: Componente gestore per l'utente corrente `Principal`.

`org.jboss.seam.web.isUserInRole`
: Consente alla pagine JSF di scegliere di generare un controllo, dipendente dai ruoli disponibili al principal corrente. `<h:commandButton value="edit" rendered="#{isUserInRole['admin']}"/>`.

32.8. Componenti per JMS

Questi componenti sono per l'uso di `TopicPublisher` e `QueueSender` gestite (vedere sotto).

`org.jboss.seam.jms.queueSession`
: Componente gestore di una `QueueSession` JMS.

`org.jboss.seam.jms.topicSession`
: Componente gestore di una `TopicSession` JMS.

32.9. Componenti relativi alla Mail

Questi componenti sono per l'uso del supporto email di Seam

`org.jboss.seam.mail.mailSession`
: Componente gestore di una `Session` JavaMail. La sessione può essere o ricercata in un contesto JNDI (impostando la proprietà `sessionJndiName`) o creata dalle opzioni di configurazione nel qual caso è obbligatori l'`host`.

- `org.jboss.seam.mail.mailSession.host` — l'hostname del server SMTP da usare.
- `org.jboss.seam.mail.mailSession.port` — la porta del server SMTP da usare.
- `org.jboss.seam.mail.mailSession.username` — lo username da usare per connettersi al server SMTP.
- `org.jboss.seam.mail.mailSession.password` — la password da usare per connettersi al server SMTP.
- `org.jboss.seam.mail.mailSession.debug` — abilita il debugging JavaMail (molto verboso).
- `org.jboss.seam.mail.mailSession.ssl` — abilita la connessioneSSL all'SMTP (porta 465 di default).

  `org.jboss.seam.mail.mailSession.tls` — di default è true, abilita il supporto TLS nella sessione mail.
- `org.jboss.seam.mail.mailSession.sessionJndiName` — nome sotto cui una javax.mail.Session è legata a JNDI. Se fornito, tutte le proprietà verranno ignorate.

32.10. Componenti infrastrutturali

Questi componenti forniscono un'infrastruttura critica di piattaforma. Si può installare un componente che non è installato di default impostando `install="true"` nel componente in `components.xml`.

`org.jboss.seam.core.init`
: Impostazioni di inizializzazione per Seam. Sempre installato.

- `org.jboss.seam.core.init.jndiPattern` — il pattern JNDI usato per la ricerca dei bean di sessione
- `org.jboss.seam.core.init.debug` — abilita la modalità di debug di Seam. Questo può essere impostato a false in produzione. Si possono vedere gli errori se il sistema è messo sotto carico ed il debug è abilitato.

- `org.jboss.seam.core.init.clientSideConversations` — se impostato a `true`, Seam salverà le variabili del contestodi conversazione nel client anziché in `HttpSession`.

`org.jboss.seam.core.manager`
Il componente interno per la pagina Seam e la gestione del contesto di conversazione. Sempre installato.

- `org.jboss.seam.core.manager.conversationTimeout` — il timeout del contesto di conversazione espresso in millisecondi.
- `org.jboss.seam.core.manager.concurrentRequestTimeout` — massimo tempo di attesa per un thread che tenta di ottenere il locksul contesto di conversazione long-running.
- `org.jboss.seam.core.manager.conversationIdParameter` — il parametro di richiesta usato per propagare l'id di conversazione, di default è `conversationId`.
- `org.jboss.seam.core.manager.conversationIsLongRunningParameter` — il parametro di richiesta usato per propagare l'informazione se la conversazione è long-running, di default è `conversationIsLongRunning`.
- `org.jboss.seam.core.manager.defaultFlushMode` — imposta la modalità flush impostata di default su ogni contesto di persistenza gestito da Seam. Di default è `AUTO`.

`org.jboss.seam.navigation.pages`
Componente interno per la gestione del workspace di Seam. Sempre installato.

- `org.jboss.seam.navigation.pages.noConversationViewId` — impostazione globale per l'id di vista a cui fare redirect quando un'entry di conversazione non viene trovata lato server.
- `org.jboss.seam.navigation.pages.loginViewId` — impostazione globale per l'id della vista a cui fare redirect quando un utente non autenticato prova ad accedere ad una vista protetta.
- `org.jboss.seam.navigation.pages.httpPort` — impostazione globale per la porta da usare quando uno schema http viene richiesto.
- `org.jboss.seam.navigation.pages.httpsPort` — impostazione globale per la porta da usare quando uno schema https viene richiesto.
- `org.jboss.seam.navigation.pages.resources` — una lista di risorse da cercare per le risorse di stile `pages.xml`. Di default è `WEB-INF/pages.xml`.

`org.jboss.seam.bpm.jbpm`
Avvia una `JbpmConfiguration`. Installa `org.jboss.seam.bpm.Jbpm` come classe.

- `org.jboss.seam.bpm.jbpm.processDefinitions` — una lista di nomi di risorsa di file jPDL da usare per l'orchestrazione dei processi di business.
- `org.jboss.seam.bpm.jbpm.pageflowDefinitions` — una lista di nomi di risorsa di file jPDL da usare per l'orchestrazione dei pageflow di conversazione.

`org.jboss.seam.core.conversationEntries`
Componente interno con scope sessione che registra le conversazioni long-running attive tra le richieste.

`org.jboss.seam.faces.facesPage`
Componente interno con scope pagina che registra il contesto di conversazione associato alla pagina.

`org.jboss.seam.persistence.persistenceContexts`
Componente interno che registra i contesti di persistenza usati nell'attuale conversazione.

`org.jboss.seam.jms.queueConnection`
Gestisce una `QueueConnection` JMS. Installata ogni volta che viene installata una `QueueSender` gestita.

- `org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName` — il nome JNDI di un `QueueConnectionFactory` JMS. Di default è `UIL2ConnectionFactory`

`org.jboss.seam.jms.topicConnection`
Gestisce una `TopicConnection` JMS. Installata ogni volta che viene installata una `TopicPublisher` gestita.

- `org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName` — il nome JNDIdi un `TopicConnectionFactory` JMS . Di default è `UIL2ConnectionFactory`

`org.jboss.seam.persistence.persistenceProvider`
Layer d'astrazione per le funzionalità non standardizzate del provider JPA.

`org.jboss.seam.core.validators`
Mette in cache istanze di `ClassValidator` di Hibernate Validator.

`org.jboss.seam.faces.validation`
Consente all'applicazione di determinare se la validazione ha fallito o ha avuto successo.

`org.jboss.seam.debug.introspector`
Supporto per la pagina di debug di Seam.

`org.jboss.seam.debug.contexts`
Supporto per la pagina di debug di Seam.

`org.jboss.seam.exception.exceptions`
Componente interno per la gestione delle eccezioni.

`org.jboss.seam.transaction.transaction`
API per controllare le transazioni ed astrarre l'implementazione della gestione delle transazioni sottostante dietro all'interfaccia compatibile-JTA.

`org.jboss.seam.faces.safeActions`
Decide se un'espressione d'azione in un URL entrante è sicura. Questo viene fatto controllando che l'espressione d'azione esista nella vista.

32.11. Componenti misti

Questi componenti non entrano in

`org.jboss.seam.async.dispatcher`
: Bean di sessione stateless dispatcher per i metodi asincroni.

`org.jboss.seam.core.image`
: Manipolazione d'immagine ed interrogazione.

`org.jboss.seam.core.pojoCache`
: Componente gestore per un'istanza PojoCache.

`org.jboss.seam.core.uiComponent`
: Gestisce una mappa per i UIComponent aventi come chiave l'id del componente.

32.12. Componenti speciali

Alcune classi speciali di componenti Seam sono installabili più volte sotto nomi specificati nella configurazione Seam. Per esempio, le seguenti linee in `components.xml` installano e configurano due componenti Seam:

```
<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
>
```

I nomi dei componenti Seam sono `bookingDatabase` e `userDatabase`.

<entityManager>, `org.jboss.seam.persistence.ManagedPersistenceContext`
: Componente gestore per un `EntityManager` gestito con scope conversazione con un contesto di persistenza esteso.

  - *<entityManager>*.entityManagerFactory — un'espressione di value binding che valuta un'istanza di `EntityManagerFactory`.

<*entityManager*>.persistenceUnitJndiName — il nome JNDI di un entity manager factory, il valore di default è java:/<*managedPersistenceContext*>.

<*entityManagerFactory*>, `org.jboss.seam.persistence.EntityManagerFactory`
Gestisce una `EntityManagerFactory` JPA. Questo è utile quando sia usa JPA fuori dall'ambiente EJB3.0.

- `entityManagerFactory.persistenceUnitName` — il nome dell'unità di persistenza.

Si vede la API JavaDoc per ulteriori proprietà di configurazione.

<*session*>, `org.jboss.seam.persistence.ManagedSession`
Componente gestore per una `Session` HIbernate gestita con scope di conversazione.

- <*session*>.sessionFactory — un'espressione di binding che valuta un'istanza di `SessionFactory`.

  <*session*>.sessionFactoryJndiName — il nome JNDI della factory di sessione, di default è java:/<*managedSession*>.

<*sessionFactory*>, `org.jboss.seam.persistence.HibernateSessionFactory`
Gestisce una `SessionFactory` di Hibernate.

- `<sessionFactory>.cfgResourceName` — il percorso al file di configurazione. Di default è `hibernate.cfg.xml`.

Si vede la API JavaDoc per ulteriori proprietà di configurazione.

<*managedQueueSender*>, `org.jboss.seam.jms.ManagedQueueSender`
Componente gestore per un `QueueSender` JMS gestito con scope evento.

- <*managedQueueSender*>.queueJndiName — il nome JNDI della coda JMS.

<*managedTopicPublisher*>, `org.jboss.seam.jms.ManagedTopicPublisher`
Componente gestore per un `TopicPublisher` JMS gestito con scope evento.

- <*managedTopicPublisher*>.topicJndiName — il nome JNDI del topic JMS.

<*managedWorkingMemory*>, `org.jboss.seam.drools.ManagedWorkingMemory`
Componente gestore di una `WorkingMemory` di Drools gestita con scope di conversazione.

- <*managedWorkingMemory*>.ruleBase — un'espressione che valuta un'istanza di `RuleBase`.

<*ruleBase*>, `org.jboss.seam.drools.RuleBase`
Componente gestore di una `RuleBase` di Drools con scope di applicazione. *Si noti che questo non è inteso per l'uso in produzione, poiché non supporta l'installazione dinamica di nuove regole.*

- <*ruleBase*>.ruleFiles — una lista di file contenenti regole Drools.

<*ruleBase*>.dslFile — una definizione DSL di Drools.

<*entityHome*>, `org.jboss.seam.framework.EntityHome`

<*hibernateEntityHome*>, `org.jboss.seam.framework.HibernateEntityHome`

<*entityQuery*>, `org.jboss.seam.framework.EntityQuery`

<*hibernateEntityQuery*>, `org.jboss.seam.framework.HibernateEntityQuery`

Controlli JSF di Seam

Seam include un numero di controlli JSF che sono utili per lavorare con Seam. Sono intesi comecomplemento ai controlli predefiniti JSF, e ai controlli di librerie di terze parti. Si raccomanda l'uso in Seam dei tag presenti nelle librerie di JBoss RichFaces, ICEsoft ICEfaces e Apache MyFaces Trinidad. Non si raccomanda l'uso dei tag della libreria Tomahawk.

33.1. Tag

Per usare questi tag si definisce il namespace "s" nella propria pagina come segue (solo facelets):

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
>
```

L'esempio ui mostra l'uso di diversi tag.

33.1.1. Controlli di navigazione

33.1.1.1. `<s:button>`

Descrizione

Un pulsante che supporta l'invocazione di un'azione con controllo sulla propagazione della conversazione. *Non effettua il submit della form.*

Attributi

- `value` — l'etichetta.
- `action` — un metodo di binding che specifica l'action listener.
- `view` — l'id della vista JSF a cui fare riferimento.
- `fragment` — l'identificatore del frammento a cui fare riferimento.
- `disabled` — il link è disabilitato?
- `propagation` — determina lo stile della propagazione della conversazione: `begin`, `join`, `nest`, `none` o `end`.
- `pageflow` — una definizione di pageflow da avviare. (E' utile solamente quando vengono usati `propagation="begin"` oppure `propagation="join"`).

Utilizzo

```
<s:button id="cancel"
        value="Cancel"
        action="#{hotelBooking.cancel}"/>
```

Si possono specificare sia `view` sia `action` in `<s:link />`. In questo caso, l'azione verrà chiamata non appena avviene il redirect alla vista specificata.

L'uso degli action listener (incluso l'action listener di default JSF) non è supportato da `<s:button />`.

33.1.1.2. `<s:conversationId>`

Descrizione

Aggiunge l'id di conversazione al link o bottone JSF (es. `<h:commandLink />` , `<s:button />`).

Attributi

Nessuno

33.1.1.3. `<s:taskId>`

Descrizione

Aggiunge l'id del task all'output link (o controllo JSF simile), quando il task è disponibile via `#{task}`.

Attributi

Nessuno.

33.1.1.4. `<s:link>`

Descrizione

Un link che supporta l'invocazione di un'azione con controllo sulla propagazione della conversazione. *Non esegue il submit della form.*

L'uso degli action listener (incluso l'action listener di default JSF) non è supportato da `<s:link />`.

Attributi

- `value` — l'etichetta.
- `action` — un metodo di binding che specifica l'action listener.
- `view` — l'id della vista JSF a cui fare riferimento.
- `fragment` — l'identificatore del frammento a cui fare riferimento.

- `disabled` — il link è disabilitato?
- `propagation` — determina lo stile della propagazione della conversazione: `begin`, `join`, `nest`, `none` o `end`.
- `pageflow` — una definizione di pageflow da avviare. (E' utile solamente quando vengono usati `propagation="begin"` oppure `propagation="join"`).

Utilizzo

```
<s:link id="register" view="/register.xhtml"
      value="Register New User"/>
```

Si possono specificare sia `view` sia `action` in `<s:link />`. In questo caso, l'azione verrà chiamata non appena avviene il redirect alla vista specificata.

33.1.1.5. `<s:conversationPropagation>`

Descrizione

Personalizza la propagazione della conversazione per un command link/button (o controllo JSF simile). *Solo Facelets.*

Attributi

- `type` — determina lo stile della propagazione della conversazione: `begin`, `join`, `nest`, `none` oppure `end`.
- `pageflow` — una definizione di pageflow da avviare. (E' utile solamente quando vengono usati `propagation="begin"` oppure `propagation="join"`).

Utilizzo

```
<h:commandButton value="Apply" action="#{personHome.update}">
  <s:conversationPropagation type="join" />
</h:commandButton
>
```

33.1.1.6. `<s:defaultAction>`

Descrizione

Specifica l'azione di default daeseguire quando viene fatto il submit della form usando il tasto invio.

Attualmente lo si può innestare solo dentro i pulsanti (es. `<h:commandButton />`, `<a:commandButton />` o `<tr:commandButton />`).

Occorre specificare un id sulla sorgente d'azione. Si può avere soltanto un'azione di default per form.

Attributi

Nessuno.

Utilizzo

```
<h:commandButton id="foo" value="Foo" action="#{manager.foo}">
  <s:defaultAction />
</h:commandButton
>
```

33.1.2. Convertitori e Validatori

33.1.2.1. `<s:convertDateTime>`

Descrizione

Esegue conversioni di data o orario nella timezone di Seam.

Attributi

Nessuno.

Utilizzo

```
<h:outputText value="#{item.orderDate}">
  <s:convertDateTime type="both" dateStyle="full"/>
</h:outputText
>
```

33.1.2.2. `<s:convertEntity>`

Descrizione

Assegna un entity converter al componente corrente. Questo è utile per pulsanti radio e controlli dropdown.

Il converter funziona con qualsiasi entity gestita - sia semplice che composta. Il converter deve essere in grado di trovare gli item dichirati nei controlli JSF durante la sottomissione della form, altrimenti si riceverà un errore di validazione.

Attributi

Nessuno.

Configurazione

Si devono usare le *transazioni di Seam gestite* (vedere *Sezione 9.2, «Transazioni gestite da Seam»*) con `<s:convertEntity />`.

Se il *Managed Persistence Context* non viene chiamato `entityManager`, allora occorre impostarlo in components.xml:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">

  <ui:jpa-entity-loader entity-manager="#{em}" />
```

Se si usa una *Managed Hibernate Session* allora occorre impostarla in components.xml:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">

  <ui:hibernate-entity-loader />
```

Se il *Managed Hibernate Session* non viene chiamato `session`, allora occorre impostarla in components.xml:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">

  <ui:hibernate-entity-loader session="#{hibernateSession}" />
```

Se si vuole usare più di un entity manager con l'entity converter, si può creare una copia dell'entity converter per ciascun entity manager in `components.xml` - si noti come l'entity converter deleghi all'entity loader l'esecuzione delle operazioni di persistenza:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">

   <ui:entity-converter name="standardEntityConverter" entity-loader="#{standardEntityLoader}"
 />

                        <ui:jpa-entity-loader name="standardEntityLoader" entity-
manager="#{standardEntityManager}" />
```

```
  <ui:entity-converter name="restrictedEntityConverter" entity-loader="#{restrictedEntityLoader}"
 />

                                <ui:jpa-entity-loader          name="restrictedEntityLoader"          entity-
manager="#{restrictedEntityManager}" />
```

```
<h:selectOneMenu value="#{person.continent}">
  <s:selectItems value="#{continents.resultList}" var="continent"
                label="#{continent.name}" />
   <f:converter converterId="standardEntityConverter" />
</h:selectOneMenu
>
```

Utilizzo

```
<h:selectOneMenu value="#{person.continent}" required="true">
  <s:selectItems value="#{continents.resultList}" var="continent"
                label="#{continent.name}"
                noSelectionLabel="Please Select..."/>
  <s:convertEntity />
</h:selectOneMenu
>
```

33.1.2.3. `<s:convertEnum>`

Descrizione

Assegna un enum converter al componente corrente. Questo è utile per i pulsanti radio e i controlli dropdown.

Attributi

Nessuno.

Utilizzo

```
<h:selectOneMenu value="#{person.honorific}">
  <s:selectItems value="#{honorifics}" var="honorific"
                label="#{honorific.label}"
                noSelectionLabel="Please select" />
  <s:convertEnum />
</h:selectOneMenu
```

```
>
```

33.1.2.4. `<s:convertAtomicBoolean>`

Descrizione

`javax.faces.convert.Converter` for `java.util.concurrent.atomic.AtomicBoolean`.

Attributi

Nessuno.

Utilizzo

```
<h:outputText value="#{item.valid}">
  <s:convertAtomicBoolean />
</h:outputText
>
```

33.1.2.5. `<s:convertAtomicInteger>`

Descrizione

`javax.faces.convert.Converter` for `java.util.concurrent.atomic.AtomicInteger`.

Attributi

Nessuno.

Utilizzo

```
<h:outputText value="#{item.id}">
  <s:convertAtomicInteger />
</h:outputText
>
```

33.1.2.6. `<s:convertAtomicLong>`

Descrizione

`javax.faces.convert.Converter` for `java.util.concurrent.atomic.AtomicLong`.

Attributi

Nessuno.

Utilizzo

```
<h:outputText value="#{item.id}">
  <s:convertAtomicLong />
</h:outputText
>
```

33.1.2.7. `<s:validateEquality>`

Descrizione

Tag da innestare dentro un controllo 'input per validare che il valore del suo padre sia uguale (o non uguale!) al valore del controllo referenziato.

Attributi

- `for` — L'id di un controllo da validare.
- `message` — Messaggio da mostrare in presenza di errori.
- `messageId` — L'id del messaggio da mostrare in presenza di errori.
- `operator` — Quale operatore usare quando si comparano i valori. Gli operatori sono:
  - `equal` — Valida che value.equals(forValue)
  - `not_equal` — Valida che !value.equals(forValue)
  - `greater` — Valida che sia ((Comparable)value).compareTo(forValue) > 0
  - `greater_or_equal` — Valida che sia ((Comparable)value).compareTo(forValue) >= 0
  - `less` — Valida che sia ((Comparable)value).compareTo(forValue) <0
  - `less_or_equal` — Valida che sia ((Comparable)value).compareTo(forValue) <= 0

Utilizzo

```
<h:inputText id="name" value="#{bean.name}"/>
<h:inputText id="nameVerification" >
  <s:validateEquality for="name" />
</h:inputText
>
```

33.1.2.8. `<s:validate>`

Descrizione

Un controllo non visuale, valida un campo d'input JSF con la proprietà collegata usando Hibernate Validator.

Attributi

Nessuno.

Utilizzo

```
<h:inputText id="userName" required="true"
             value="#{customer.userName}">
  <s:validate />
</h:inputText>
<h:message for="userName" styleClass="error" />
```

33.1.2.9. `<s:validateAll>`

Descrizione

Un controllo non visuale, valida tutti i campi d'input JSF con le proprietà collegate usando Hibernate Validator.

Attributi

Nessuno.

Utilizzo

```
<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
>
```

33.1.3. Formattazione

33.1.3.1. `<s:decorate>`

Descrizione

"Decora" un campo d'input JSF quando la validazione fallisce o quando è impostato `required="true"`.

Attributi

- `template` — il template facelets da usare per decorare il componente
- `enclose` — se true, il template usato per decorare il campo d'input è racchiuso dall'elemento specificato con l'attributo "element". Di default questo è un elemento div.
- `element` — l'elemento con cui racchiudere il template usato per decorare il campo d'input. Di default, il template è racchiuso con un elemento div.

`#{invalid}` e `#{required}` sono disponibili dentro `s:decorate`; `#{required}` valuta `true` se si è impostato il componente input da decorare come richiesto, e `#{invalid}` valuta `true` se avviene un errore di validazione.

Utilizzo

```
<s:decorate template="edit.xhtml">
  <ui:define name="label"
>Country:</ui:define>
     <h:inputText value="#{location.country}" required="true"/>
  </s:decorate
>
```

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

```
  <div
>

    <s:label styleClass="#{invalid?'error':''}">
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}"
>*</s:span>
    </s:label>

    <span class="#{invalid?'error':''}">
      <s:validateAll>
        <ui:insert/>
      </s:validateAll>
    </span>

    <s:message styleClass="error"/>

  </div
>

</ui:composition
>
```

33.1.3.2. `<s:div>`

Descrizione

Generare un `<div>` HTML.

Attributi

Nessuno.

Utilizzo

```
<s:div rendered="#{selectedMember == null}">
  Sorry, but this member does not exist.
</s:div
>
```

33.1.3.3. `<s:span>`

Descrizione

Generare un HTML.

Attributi

- `title` — Fornisce un titolo per uno span.

Utilizzo

```
<s:span styleClass="required" rendered="#{required}" title="Small tooltip"
>*</s:span
>
```

33.1.3.4. <s:fragment>

Descrizione

Un componente non -rendering utile per abilitare/disabilitare il rendering dei suoi figli.

Attributi

Nessuno.

Utilizzo

```
<s:fragment rendered="#{auction.highBidder ne null}">
  Current bid:
</s:fragment
>
```

33.1.3.5. <s:label>

Descrizione

"Decora" un campo d'input JSF con l'etichetta. L'etichetta è messa dentro il tag HTML <label>, ed è associato al componente d'input JSF più vicino. E' spesso usato con <s:decorate>.

Attributi

- `style` — Lo stile del controllo
- `styleClass` — La classe di stile del controllo

Utilizzo

```
<s:label styleClass="label">
 Country:
```

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

33.1.3.6. `<s:message>`

Descrizione

"Decora" un campo d'input JSF con il messaggio d'errore di validazione.

Attributi

Nessuno.

Utilizzo

```
<f:facet name="afterInvalidField">
  <s:span>
     Error: 
    <s:message/>
  </s:span>
</f:facet
>
```

33.1.4. Seam Text

33.1.4.1. `<s:validateFormattedText>`

Descrizione

Controlla che il valore inviato sia un Testo Seam valido.

Attributi

Nessuno.

33.1.4.2. `<s:formattedText>`

Descrizione

Mostra come output *Seam Text*, un markup di testo utile per blog, wiki ed altre applicazioni che possono usare rich text. Vedere il capitolo di Seam Text per ulteriori informazioni.

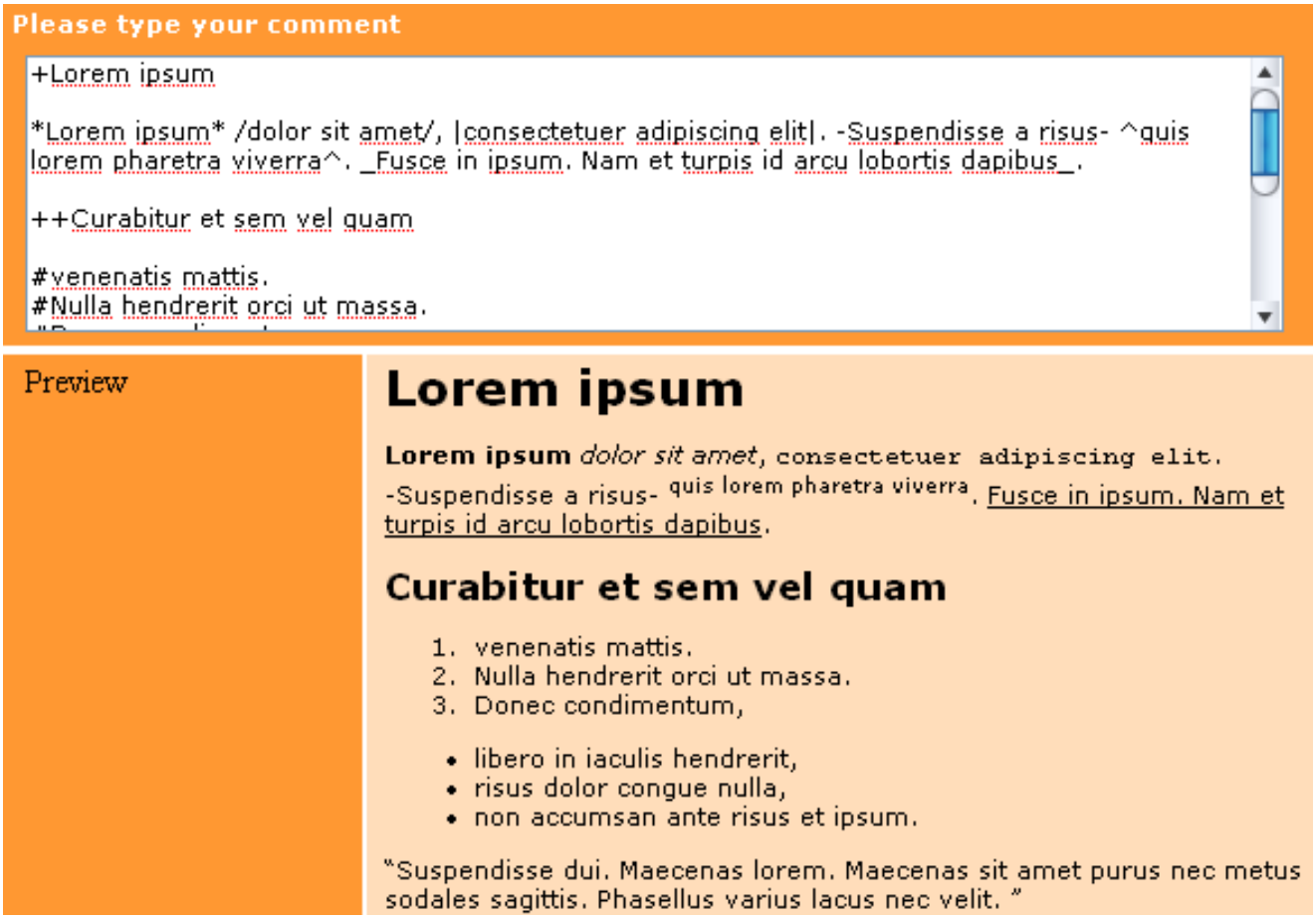
Attributi

- `value` — un'espressione EL che specifica il markup del testo rich da renderizzare.

Utilizzo

```
<s:formattedText value="#{blog.text}"/>
```

Esempio



33.1.5. Supporto per le form

33.1.5.1. `<s:token>`

Descrizione

Produce un token casuale che viene inserito in un campo di form nascosta per aiutare rendere sicuro i post della form JSF contro attacchi cross-site request forgery (XSRF). Si noti che il browser deve avere abilitati i cookie per poter eseguire il submit delle form che includono questo componente.

Attributi

- `requireSession` — indica se l'id di sessione debba essere incluso nella signature della form, da qui il binding del token alla sessione. Questo valore può essere impostato a false se viene attivata la modalità "build before restore" dei Facelets (di default in JSF 2.0). (Default: false)

- `enableCookieNotice` — indica che deve essere inserito nella pagina un check JavaScript per verificare che i cookie siano abilitati nel browser. Se i cookie non sono abilitati, viene presentata all'utente una nota riportante che l'invio della form non è avvenuto. (Default: false)
- `allowMultiplePosts` — indica se consentire che la stessa form venga inviata più volte con la stessa signature (finché non cambia la vista). Questa è una necessità comune se la form esegue chiamate Ajax ma non rigenera se stessa o, almeno, il componente UIToken. L'approccio migliore è avere il componente UIToken rigenerato su ogni chiamata Ajax in cui il componente UIToken viene processato. (Default: false)

Utilizzo

```
<h:form>
  <s:token enableCookieNotice="true" requireSession="false"/>
  ...
</h:form
>
```

33.1.5.2. `<s:enumItem>`

Descrizione

Crea un `SelectItem` da un valore enum.

Attributi

- `enumValue` — la rappresentazione stringa di un valore enum.
- `label` — l'etichetta da usare per renderizzare il `SelectItem`.

Utilizzo

```
<h:selectOneRadio id="radioList"
                  layout="lineDirection"
                  value="#{newPayment.paymentFrequency}">
  <s:convertEnum />
  <s:enumItem enumValue="ONCE"         label="Only Once" />
  <s:enumItem enumValue="EVERY_MINUTE" label="Every Minute" />
  <s:enumItem enumValue="HOURLY"       label="Every Hour" />
  <s:enumItem enumValue="DAILY"        label="Every Day" />
  <s:enumItem enumValue="WEEKLY"       label="Every Week" />
</h:selectOneRadio
>
```

33.1.5.3. `<s:selectItems>`

Descrizione

Crea una `List<SelectItem>` daList, Set, DataModel o Array.

Attributi

- `value` — un'espressione EL che specifica i dati retrostanti a `List<SelectItem>`
- `var`— definisce il nome della variabile locale che mantiene l'oggetto corrente durante l'iterazione
- `label` — l'etichetta da usare per generare `SelectItem`. Può fare riferimento alla variabile `var`.
- `itemValue` — Il valore da restituire al server se quest'opzione è selezionata. E' opzionale, di default viene usato l'oggetto `var`. Può fare riferimento alla variabile `var`.
- `disabled` — se true il `SelectItem` verrà generato disabilitato. Può fare riferimento alla variabile `var`.
- `noSelectionLabel` — specifica l'etichetta (opzionale) da mettere in cima alla lista (se è specificato anche `required="true"` allora selezionando questo valore può causare una validazione d'errore).
- `hideNoSelectionLabel` — se true, `noSelectionLabel`verrà nascosto quando viene selezionato il valore.

Utilizzo

```
<h:selectOneMenu value="#{person.age}"
                 converter="ageConverter">
  <s:selectItems value="#{ages}" var="age" label="#{age}" />
</h:selectOneMenu
>
```

33.1.5.4. `<s:fileUpload>`

Descrizione

Renderizza un controllo per l'upload del file. Questo controllo deve essere usato dentro la form con tipo di codifica `multipart/form-data`, cioè:

```
<h:form enctype="multipart/form-data"
>
```

Per richieste multiple, in `web.xml` deve essere configurato il filtro servlet Seam Multipart :

```
<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
>
```

Configurazione

Le seguenti opzioni di configurazioni per richieste multipart possono essere configurate in components.xml:

- `createTempFiles` — se quest'opzione è impostata a true, i file caricati vengono accodati in un file temporaneo invece che in memoria.
- `maxRequestSize` — dimensione massima della richiesta di upload file, in byte.

Ecco un esempio:

```
<component class="org.jboss.seam.web.MultipartFilter">
  <property name="createTempFiles"
>true</property>
  <property name="maxRequestSize"
>1000000</property>
</component
>
```

Attributi

- `data` — questo value binding riceve i dati binari. Il campo ricevente deve essere dichiarato come `byte[]` o `InputStream` (richiesto).
- `contentType` — questo value binding riceve il contenuto del file (opzionale).
- `fileName` — questo value binding riceve il nome del file (opzionale).

- `fileSize` — questo valore di binding riceve la dimensione del file (opzionale).
- `accept` — una lista separata da virgola con i tipi di contenuto da accettare, può non essere supportata dal browser. Esempio: `"images/png,images/jpg"`, `"images/*"`.
- `style` — Lo stile del controllo
- `styleClass` — La classe di stile del controllo

Utilizzo

```
<s:fileUpload id="picture" data="#{register.picture}"
              accept="image/png"
              contentType="#{register.pictureContentType}" />
```

33.1.6. Altro

33.1.6.1. `<s:cache>`

Descrizione

Mette nella cache il frammento di pagina renderizzato usando JBoss Cache. Si noti che `<s:cache>` in verità usa l'istanza di JBoss Cache gestita dal componente predefinito `pojoCache`.

Attributi

- `key` — la chiave per memorizzare (cache) il contenuto renderizzato, spesso un'espressione di valore. Per esempio, se si mette nella cache un frammento di pagina che mostra un documento, si può usare `key` —
- `enabled` — un'espressione di valore che determina se la cache debba essere usata.
- `region` — un node JBoss Cache da usare (nodi differenti possono avere differenti policy di scadenza).

Utilizzo

```
<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&amp;#160;
```

```
    <h:outputText value="#{blogEntry.date}">
      <f:convertDateTime timezone="#{blog.timeZone}" locale="#{blog.locale}"
                         type="both"/>
    </h:outputText
>]
   </p>
  </div>
</s:cache
>
```

33.1.6.2. `<s:resource>`

Descrizione

Un tag che agisce come fornitore di download dei file. Deve essere il solo nella pagina JSF. Per essere in grado di usare questo controllo, web.xml deve essere impostato come segue.

Configurazione

```
<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>
```

Attributi

- `data` — I dati da downloadare. Può essere java.util.File, InputStream o un byte array.
- `fileName` — Nome del file da servire
- `contentType` — tipo di contenuto del file da scaricare
- `disposition` — disposizione da usare. Di default è inline

Utilizzo

Ecco un esempio su come usare il tag:

```
<s:resource xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib"
      data="#{resources.data}"
      contentType="#{resources.contentType}"
      fileName="#{resources.fileName}" />
```

Il bean chiamato `resources` fornisce al file i parametri di richiesta del server, si veda `s:download`.

33.1.6.3. `<s:download>`

Descrizione

Costruisce un link RESTful a `<s:resource>`. `f:param` innestati costruiscono l'url.

- `src` — I file che servono al file di risorsa.

Attributi

```
<s:download src="/resources.xhtml">
   <f:param name="fileId" value="#{someBean.downloadableFileId}"/>
</s:download
>
```

Genererà qualcosa di simile a: `http://localhost/resources.seam?fileId=1`

33.1.6.4. `<s:graphicImage>`

Descrizione

Un `<h:graphicImage>` esteso che consente all'immagine di essere creata in un componente Seam; possono essere applicate all'immagine altre trasformazioni.

Tutti gli attributi per `<h:graphicImage>` sono supportati, così come:

Attributi

- `value` — immagine da visualizzare. Può essere un path `String` (caricato dal classpath), un `byte[]`, un `java.io.File`, un `java.io.InputStream` o un `java.net.URL`. I formati d'immagine supportati sono `image/png`, `image/jpeg` e `image/gif`.
- `fileName` — se non specificato l'immagine servita avrà un nome di file generato. Se si vuole nominare il file, occorre specificarlo in questo attributo. Il nome deve essere univoco.

Trasformazioni

Per applicare una trasformazione all'immagine, occorre innestare un tag specificando la trasformazione da applicare. Seam attualmente supporta queste trasformazioni:

`<s:transformImageSize>`

- `width` — nuova larghezza dell'immagine
- `height` — nuova altezza dell'immagine
- `maintainRatio` — se vengono specificati `true`, ed *uno* fra `width`/`height`, l'immagine sarà ridimensionata con la dimensione non specificata che viene calcolata per mantenere l'aspect ratio.
- `factor` — scala l'immagine col dato fattore

`<s:transformImageBlur>`

- `radius` — esegue una convolution blur con il raggio dato

`<s:transformImageType>`

- `contentType` — modifica il tipo d'immagine in `image/jpeg` o `image/png`

E' facile creare una trasformazione - si crei un `UIComponent` che implementi `org.jboss.seam.ui.graphicImage.ImageTransform`. Dentro il metodo `applyTransform()` si usi `image.getBufferedImage()` per recuperare l'immagine originale e `image.setBufferedImage()` per impostare l'immagine trasformata. Le trasformazioni sono applicare nell'ordine specificato nella vista.

Utilizzo

```
<s:graphicImage rendered="#{auction.image ne null}"
                value="#{auction.image.data}">
  <s:transformImageSize width="200" maintainRatio="true"/>
</s:graphicImage
>
```

33.1.6.5. `<s:remote>`

Descrizione

Genera gli stub Javascript richiesti per usare Seam Remoting.

Attributi

- `include` — una lista separata da virgola di nomi di componenti (o nome pienamente qualificati) per i quali generare stub Seam Remoting Javascript. Si veda *Capitolo 25, Remoting* per maggiori dettagli.

Utilizzo

```
<s:remote include="customerAction,accountAction,com.acme.MyBean"/>
```

33.2. Annotazioni

Seam fornisce anche annotazioni che permettono di impiegare i componenti Seam come convertitori JSF e validatori:

`@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" />
```

Registra il componente Seam come converter JSF. Qua è mostrato un converter capace di accedere all'EntityManager JPA dentro ad una transazione JTA, quando converte il valore legato alla rappresentazione del suo oggetto.

`@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" />
```

Registra il componente Seam come validatore JSF. Qua è mostrato un validatore che inietta un altro componente Seam; il componente iniettato è usato per validare il valore.

JBoss EL

Seam utilizza JBoss EL, il quale fornisce un'estensione allo standard Unified Expression Language (EL). JBoss EL apporta un numero di miglioramenti che incrementano l'espressività e la potenza delle espressioni EL.

34.1. Espressioni parametrizzate

Lo standard EL non consente di utilizzare un metodo con parametri definiti dall'utente — sicuramente metodi JSF listener (es. `valueChangeListener`) prende i parametri forniti da JSF.

JBoss EL rimuove questa restrizione. Per esempio:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel)}" value="Book Hotel"/>
```

```
@Name("hotelBooking")
public class HotelBooking {

  public String bookHotel(Hotel hotel) {
    // Book the hotel
  }
}
```

34.1.1. Utilizzo

Come nelle chiamate ai metodi in Java, i parametri sono racchiusi tra parentesi e separati da virgole:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}" value="Book Hotel"/>
```

I parametri `hotel` e `user` verranno valutati come espressioni di valore e passati al metodo `bookHotel()` del componente.

Qualsiasi valore d'espressione può essere usato come parametro:

```
<h:commandButton
  action="#{hotelBooking.bookHotel(hotel.id, user.username)}"
  value="Book Hotel"/>
```

E' importante capire bene come funziona quest'estensione a EL. Quando la pagine viene generata, i *nomi* dei parametri vengono memorizzati (per esempio `hotel.id` e `user.username`) e valutati (come espressioni di valore) quando la pagina viene inviata. Non si possono passare oggetti come parametri!

Devi assicurarti che i parametri siano disponibili non solo quando la pagina viene generata, ma anche quando ne viene fatto il submit. Se gli argomenti non possono essere risolti quando la pagina viene inviata, il metodo d'azione verrà chiamato con argomenti `null`!

Si può passare stringe letterali usando virgolette singole:

```
<h:commandLink action="#{printer.println('Hello world!')}" value="Hello"/>
```

EL unificato supporta anche le espressioni di valore, usate per associare un campo ad un bean. Le espressioni di valore utilizzano le convenzioni dei nomi di JavaBean e richiedono get e set. Spesso JSP si attende un'espressione di valore dove solo un recupero (get) è richiesto (es. l'attributo `rendered`). Molti oggetti, comunque, non hanno nominato in modo appropriato i metodi accessor alle proprietà o i parametri richiesti.

JBoss EL rimuove questa restrizione permettendo che i valori vengano recuperati usando la sintassi del metodo. Per esempio:

```
<h:outputText value="#{person.name}" rendered="#{person.name.length()
> 5}" />
```

Si può accedere alla dimensione di una collezione in maniera analoga:

```
<h:outputText value="#{person.name}" rendered="#{person.name.length()
> 5}" />
```

In generale qualsiasi espressione nella forma #{obj.property} è identica all'espressione #{obj.getProperty()}.

Sono consentiti anche i parametri. Il seguente esempio chiama `productsByColorMethod` con un argomento stringa letterale:

```
#{controller.productsByColor('blue')}
```

34.1.2. Limitazioni e suggerimenti

Nell'uso di JBoss EL dovresti tenere presente i seguenti punti:

- *Incompatibilità con JSP 2.1* — JBoss EL non può attualmente essere utilizzato con JSP 2.1 poiché il compilatore rifiuta espressioni con paramentri al suo interno. Quindi per usare quest'estensione con JSF 1.2, si dovranno utilizzare Facelets. Quest'estensione funziona correttamente con JSP 2.0.

- *Utilizzo all'interno di componenti iterativi* — Componenti quali `<c:forEach />` e `<ui:repeat />`iterano su una lista o un array, esponendo ogni item della lista ai componenti innestati. Questo funziona bene selezionando una riga con `<h:commandButton />` o `<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
>
```

Comunque si voglia usare `<s:link/>` o `<s:button/>` si *deve* esporre gli item come `DataModel` e usare `<dataTable />` (o equivalente da componente impostato come `<rich:dataTable />`). Né `<s:link />` né `<s:button />` eseguono il submit della form (e quindi producono un bookmarkable link) quindi serve un parametro "magico" per ricreare l'item quando viene chiamato l'action method. Questo parametro magico può essere aggiunto soltanto quando viene usata una data table con dietro un `DataModel`.

- *Chiamata di un `MethodExpression` da codice Java* — normalmente quando una `MethodExpression` viene creata, i tipi di parametro sono passati tramite JSF. Nel caso di un binding di metodo, JSF presume che non ci siano parametri da passare. Con quest'estensione non è possibile sapere il tipo di parametro prima che l'espressione venga valutata. Ciò ha due conseguenze:

  - Quando viene invocato un `MethodExpression` nel codice Java, i parametri passati potrebbero essere ignorati. I parametri definiti nell'espressione avranno la precedenza.

  - Solitamente è sicuro chiamare in ogni momento `methodExpression.getMethodInfo().getParamTypes()`. Per un'espressione con parametri occorre invocare il `MethodExpression` prima di chiamare `getParamTypes()`.

Entrambi questi casi sono estremamente rari e si applicano solo quando si vuole invocare manualmente `MethodExpression` nel codice Java.

34.2. Proiezione

JBoss EL supporta una limitata sintassi di proiezione. Un'espressione di proiezione mappa una sotto-espressione attraverso un'espressione a valori multipli (lista, set, ecc...). Per esempio, l'espressione:

```
#{company.departments}
```

potrebbe restituire una lista di dipartimenti. Se occorresse una lista di nomi di dipartimento, l'unica opzione è quella di iterare sulla lista per recuperare i valori. JBoss EL permette questo con l'espressione di proiezione:

```
#{company.departments.{d|d.name}}
```

La sotto-espressione è racchiusa da parentesi. In quest'esempio l'espressione `d.name` viene valutata per ogni dipartimento, usando `d` come alias per l'oggetto dipartimento. Il risultato di quest'espressione sarà una lista di valori Stringa.

Qualsiasi espressione valida può essere usata in un'espressione, e quindi sarebbe perfettamente valido scrivere la seguente, assumendo che venga usata per le lunghezze di tutti i nomi di dipartimento in un'azienda:

```
#{company.departments.{d|d.size()}}
```

Le proiezioni possono essere annidate. La seguente espressione ritorna gli ultimi nomi di ciascun impiegato in ogni dipartimento:

```
#{company.departments.{d|d.employees.{emp|emp.lastName}}}
```

Le proiezioni annidate possono comunque rivelarsi un pò difficoltose. La seguente espressione sembra ritornare una lista di tutti gli impiegati in tutti i dipartimenti:

```
#{company.departments.{d|d.employees}}
```

Comunque, restituisce una lista contenente una lista di impiegati per ogni singolo dipartimento. Per combinare questi valori è necessario usare un'espressione leggermente più lunga:

```
#{company.departments.{d|d.employees.{e|e}}}
```

E' importante notare che questa sintassi non può essere analizzata da Facelets o JSP e quindi non può essere usata in file xhtml o jsp. Anticipiamo che la sintassi di proiezione cambierà nelle future versioni di JBoss EL.

Tuning delle performance

Questo capitolo è un tentativo di documentare in un unico posto tutti i suggerimenti per ottenere migliori performance da un'applicazione Seam.

35.1. Bypassare gli interceptor

Per ripetitivi binding di valore come quelli in dataTable JSF o in altri controlli iterativi (come `ui:repeat`), l'intero stack di interceptor verrà chiamato ad ogni invocazione di un componente Seam referenziato. L'effetto di questo può essere una sostanziale degradazione di performance, specialmente se il componente viene chiamato molte volte. Un guadagno significativo di performance può essere ottenuto disabilitando lo stack degli interceptor per il componente Seam da invocare. Per disabilitare gli interceptor di un componente occorre aggiungere l'annotazione `@BypassInterceptors` alla classe del componente.

Avvertimento

E' molto importante essere consapevoli delle implicazioni che seguono la disabilitazioni degli interceptor per un componente Seam. Funzionalità quali la bijection, le restrizioni annotate sulla sicurezza, la sincronizzazione e altro vengono a mancare per un componente marcato con `@BypassInterceptors`. Mentre nella maggior parte dei casi è possibile compensare la perdita di queste funzionalità (es. invece di iniettare un componente usando `@In`, si può invece usare `Component.getInstance()`) è importante essere consapevoli delle conseguenze.

Il seguente listato di codice mostra come un componente Seam venga disabilitato con i suoi interceptor:

```
@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;
  }
}
```

Test delle applicazioni Seam

La maggior parte delle applicazioni Seam ha bisogno di almeno due tipi di test automatici: *test di unità* per testare un particolare componente Seam in isolamento, e *test d'integrazione* per provare tutti i layer java dell'applicazione (cioè tutto, tranne le pagine di vista).

Entrambi i tipi di test sono facili da scrivere.

36.1. Test d'unità dei componenti Seam

Tutti i componenti Seam sono POJO. Questo è un buon punto per partire se si vogliono eseguire dei test di unità facili. E poiché Seam enfatizza l'uso della bujection per le interazioni tra componenti e l'accesso ad oggetti contestuali, è molto facile testare un componente Seam fuori dal suo normale ambiente di runtime.

Si consideri il seguente componente Seam che crea una dichiarazione di account per un cliente:

```
@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

}
```

Si può scrivere un test d'unità per il metodo calculateTotal (che testa la business logic del componente) come segue:

```
public class StatementOfAccountTest {

    @Test
    public testCalculateTotal {
       List<Invoice
> invoices = generateTestInvoices(); // A test data generator
       double statementTotal = new StatementOfAccount().calculateTotal(invoices);
       assert statementTotal = 123.45;
    }
}
```

Si vede che non si sta testando il recupero dei dati e la persistenza dei dati da/a database; e neppure si testa alcuna funzionalità fornita da Seam. Si sta solamente testanto la logica del POJO. I componenti Seam solitamente non dipendono direttamente dall'infrastruttura del container, e quindi la maggior parte dei test d'unità sono facili come quello mostrato!

Comunque se si vuole testare l'intera applicazione, si consiglia di continuare nella lettura.

36.2. Test d'integrazione dei componenti Seam

Il test d'integrazione è leggermente più difficile. In questo caso non si può eliminare l'infrastruttura del container che invece fa parte di ciò che va testato! Allo stesso tempo non si vuole essere forzati ad eseguire un deploy dell'applicazione in un application server per fare girare i test. Occorre essere in grado di riprodurre l'infrastruttura essenziale del container dentro l'ambiente di test per poter provare l'intera applicazione senza nuocere troppo alle performance.

L'approccio preso da Seam è quello di consentire di scrivere test che possano provare i componenti mentre girano dentro un ambiente di container ridotto (Seam assieme al container JBoss Embedded; vedere *Sezione 30.6.1, «Installare JBoss Embedded»* per i dettagli di configurazione)

```
public class RegisterTest extends SeamTest
{

  @Test
  public void testRegisterComponent() throws Exception
  {

    new ComponentTest() {

      protected void testComponents() throws Exception
      {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
        assert invokeMethod("#{register.register}").equals("success");
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }

    }.run();

  }

  ...

}
```

36.2.1. Uso dei mock nei test d'intergrazione

Occasionalmente occorre essere in grado di sostituire l'implementazione di alcuni componenti Seam che dipendono da risorse non disponibili in ambiente di test. Per esempio, si supponga di avere dei componenti Seam che sono una facade di un qualche sistema di elaborazione pagamenti:

```
@Name("paymentProcessor")
public class PaymentProcessor {
   public boolean processPayment(Payment payment) { .... }
}
```

Per i test di integrazione è possibile creare un mock di un componente come segue:

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
   public boolean processPayment(Payment payment) {
      return true;
   }
}
```

Poiché la precedenza `MOCK` è più elevata della precedenza di default dei componenti di applicazione, Seam installerà l'implementazione mock quando questa è nel classpath. Con un deploy in produzione, l'implementazione mock è assente, e quindi il componente reale verrà installato.

36.3. Test d'integrazione delle interazioni utente in applicazioni Seam

Un problema ancora più difficile è quello di emulare le interazioni utente. Un terzo problema si verifica quando vengono messe le asserzioni. Alcuni framewrok di test consentono di testare un'intera applicazione riproducendo le interazioni utente con il browser. Questi framework hanno un loro spazio d'uso, ma non sono adatti per l'uso in fase di sviluppo.

"`SeamTest` consente di scrivere test *sotto forma di script (scripted)*, in un ambiente JSF simulato. Il ruolo di un test scripted è quello di riprodurre l'interazione tra la vista ed i componenti Seam. In altre parole si pretende di essere l'implementazione JSF!

Questo approccio testa ogni cosa tranne la vista.

Si consideri una vista JSP per il componente testato come unità visto sopra:

```
<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
>
```

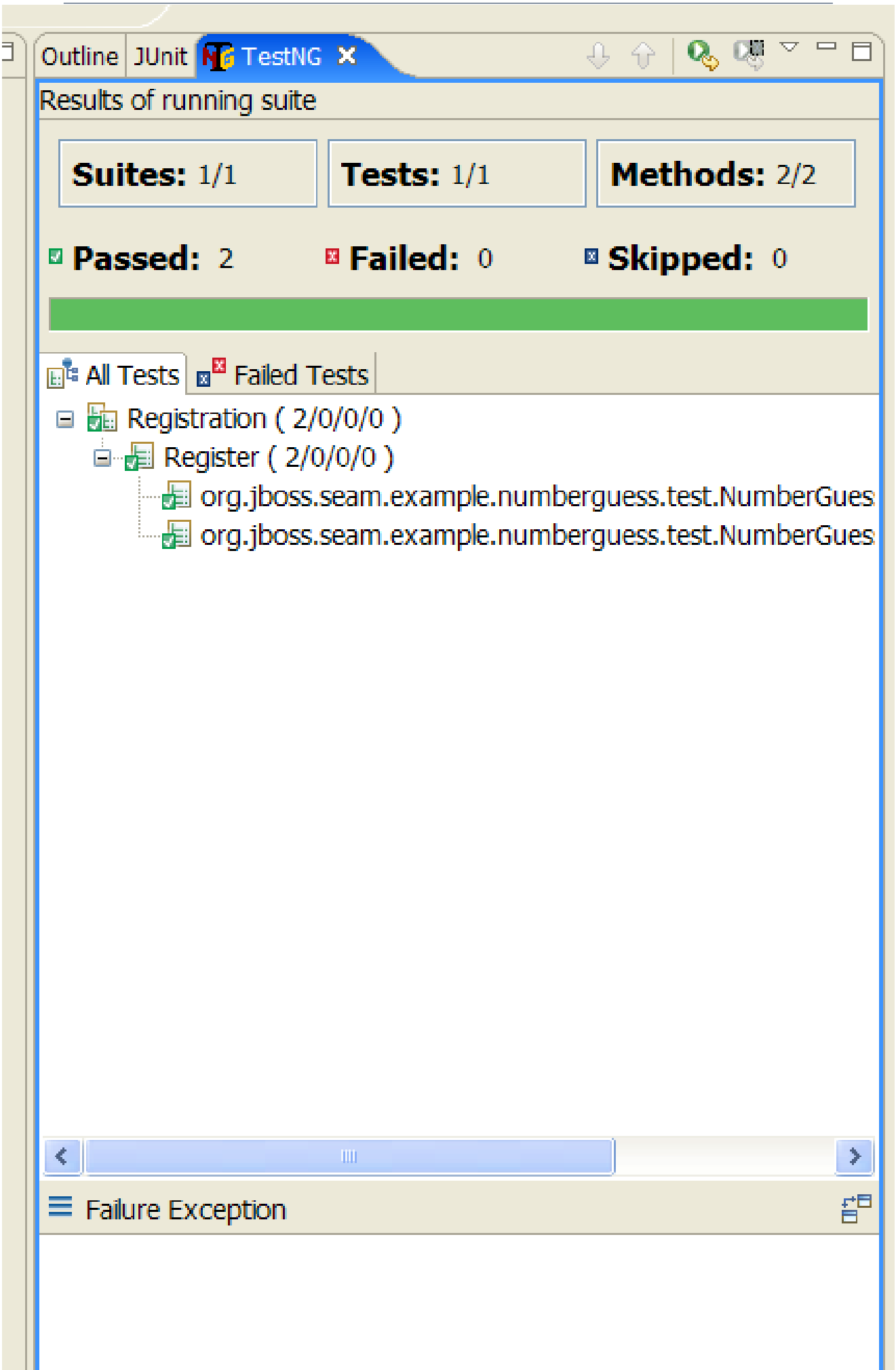
Si vuole testare la funzionalità di registrazione dell'applicazione (la cosa che succede quando l'utente clicca il pulsante Registra). Si riprodurrà ilciclo di vita della richiesta JSF in un test TestNG automatizzato:

```
public class RegisterTest extends SeamTest
{

  @Test
  public void testRegister() throws Exception
  {

    new FacesRequest() {

      @Override
      protected void processValidations() throws Exception
      {
        validateValue("#{user.username}", "1ovthafew");
        validateValue("#{user.name}", "Gavin King");
        validateValue("#{user.password}", "secret");
        assert !isValidationFailure();
      }
```

```
        @Override
        protected void updateModelValues() throws Exception
        {
          setValue("#{user.username}", "1ovthafew");
          setValue("#{user.name}", "Gavin King");
          setValue("#{user.password}", "secret");
        }

        @Override
        protected void invokeApplication()
        {
          assert invokeMethod("#{register.register}").equals("success");
        }

        @Override
        protected void renderResponse()
        {
          assert getValue("#{user.username}").equals("1ovthafew");
          assert getValue("#{user.name}").equals("Gavin King");
          assert getValue("#{user.password}").equals("secret");
        }

      }.run();

    }

    ...

}
```

Si noti che si è esteso `SeamTest`, che fornisce un ambiente Seam ai componenti, e si è scritto uno script di test come classe anonima che estende `SeamTest.FacesRequest`, la quale fornisce un ciclo di vita emulato della richiesta JSF. (C'è anche `SeamTest.NonFacesRequest` per testare le richieste GET). Si è scritto codice in metodi che vengono chiamati per le varie fasi JSF, per emulare le chiamate che JSF farebbe ai componenti. Infine si sono scritte le asserzioni.

Nelle applicazioni d'esempio di Seam ci sono molti test d'integrazione che mostrano casi ancora più complicati. Ci sono istruzioni per eseguire questi test usando Ant o usando il plugin TestNG per Eclipse:


Outline
JUnit
TestNG
Results of running suite
Suites: 1/1
Tests: 1/1
Methods: 2/2
Passed: 2
Failed: 0
Skipped: 0
All Tests
Failed Tests
Registration (2/0/0/0)
Register (2/0/0/0)
org.jboss.seam.example.numberguess.test.NumberGues
org.jboss.seam.example.numberguess.test.NumberGues
Failure Exception

36.3.1. Configurazione

Se è stato usato seam-gen per creare il progetto si è già pronti per scrivere test. Altrimenti occorre configurare l'ambiente di test all'interno del proprio tool di build (es. ant, maven, eclipse).

Prima si guardi alle dipendenze necessarie:

Tabella 36.1.

Group Id	Artifact Id	Locazione in Seam
`org.jboss.seam.embedded`	`hibernate-all`	`lib/test/hibernate-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-all`	`lib/test/jboss-embedded-all.jar`
`org.jboss.seam.embedded`	`thirdparty-all`	`lib/test/thirdparty-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-api`	`lib/jboss-embedded-api.jar`
`org.jboss.seam`	`jboss-seam`	`lib/jboss-seam.jar`
`org.jboss.el`	`jboss-el`	`lib/jboss-el.jar`
`javax.faces`	`jsf-api`	`lib/jsf-api.jar`
`javax.el`	`el-api`	`lib/el-api.jar`
`javax.activation`	`javax.activation`	`lib/activation.jar`

E' molto importante non inserire nel classpath le dipendenze di JBoss AS a compile time da `lib/` (es. `jboss-system.jar`), altrimenti questo causerà il non avvio di JBoss Embedded. Quindi si aggiungano solo le dipendenze necessarie per partire (es.Drools, jBPM).

Occorre includere nel classpath la directory `bootstrap/`; `bootstrap/` contiene la configurazione per JBoss Embedded.

E sicuramente occorre mettere nel classpath il progetto ed i test così come i jar del framework di test. Non si dimentichi di mettere nel classpath anche tutti i file di configurazione corretti per JPA e Seam. Seam chiede a JBoss Embedded di deployare le risorse (jar o directory) che hanno `seam.properties` nella root. Quindi se non si assembla una struttura di directory che assomiglia ad un archivio deployabile contenente il progetto, occorre mettere `seam.properties` in ciascuna risorsa.

Di default un progetto generato userà per i test `java:/DefaultDS` (un datasource predefinito HSQL in JBoss Embedded). Se si vuole usare un altro datasource, si metta `foo-ds.xml` nella directory `bootstrap/deploy`.

36.3.2. Uso di SeamTest con un altro framework di test

Seam include il supporto TestNG, ma a piacimento è possibile usare un altro framewrok di test quale JUnit.

Occorre fornire un'implementazione di `AbstractSeamTest` che faccia le seguenti cose:

- Chiami `super.begin()` prima di ciascun test.
- Chiami `super.end()` dopo ciascun metodo di test.
- Chiami `super.setupClass()` per configurare l'ambiente di test d'integrazione. Questo deve essere chiamato prima dei metodi di test.
- Chiami `super.cleanupClass()` per pulire l'ambiente di test d'integrazione.
- Chiami `super.startSeam()` per avviare Seam all'avvio dei test.
- Chiami `super.stopSeam()` per terminare in modo corretto Seam alla fine dei test.

36.3.3. Test d'integrazione con Dati Mock

Se occorre inserire o pulire i dati nel database prima di ogni test, si può usare l'integrazione di Seam con DBUnit. Per fare questo si estenda DBUnitSeamTest piuttosto che SeamTest.

Devi fornire un dataset per DBUnit.

Attenzione

DBUnit supporta due formati per i file di dataset, flat e XML. DBUnitSeamTest di Seam assume che venga usato il formato flat, e quindi accertarsi che il dataset sia in questo formato.

```
<dataset>

  <ARTIST
    id="1"
    dtype="Band"
    name="Pink Floyd" />

  <DISC
    id="1"
    name="Dark Side of the Moon"
    artist_id="1" />
```

```
</dataset
>
```

e comunicarlo a Seam facendo l'override di `prepareDBUnitOperations()`:"

```
protected void prepareDBUnitOperations() {
   beforeTestOperations.add(
      new DataSetOperation("my/datasets/BaseData.xml")
   );
}
```

`DataSetOperation` è impostato di default a `DatabaseOperation.CLEAN_INSERT` se non viene specificata qualchealtra operazione come argomento del costruttore. L'esempio di cui sopra pulisce tutte le tabelle definite in `BaseData.xml` e quindi inserisce tutte le richedichiarate in `BaseData.xml` prima che ogni metodo `@Test` venga invocato.

Se viene richiesta un'ulteriore pulizia prima dell'esecuzione di un metodo di test, si aggiungano operazioni alla lista `afterTestOperations`.

Occorre informare DBUnit del datasource usato impostando il parametro TestNG chiamato `datasourceJndiName`:

```
<parameter name="datasourceJndiName" value="java:/seamdiscsDatasource"/>
```

DBUnitSeamTest supporta MySQL e HSQL- occorre dire quale database viene usato:

```
<parameter name="database" value="HSQL" />
```

Questo consente anche di inserire dati binari nel set dei dati di test (N.B. questo non è stato testato sotto Windows). Occorre dire dove si trovano queste risorse:

```
<parameter name="binaryDir" value="images/" />
```

Si *devono* specificare questi tre parametri in `testng.xml`.

Per usare DBUnitSeamTest con altri database occorre implementare alcuni metodi. Si legga javadoc di `AbstractDBUnitSeamTest` per saperne di più.

36.3.4. Test d'integrazione di Seam Mail

Attenzione

Attenzione! Questa funzionalità è ancora in fase di sviluppo.

E' facilissimo eseguire il test d'integrazione con 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();
  }
}
```

Viene creata una nuova `FacesRequest` comenormale. Dentro la sezione invokeApplication mostriamo il messaggio usando `getRenderedMailMessage(viewId);`, passando il viewId del messaggio da generare. Il metodo restituisce il messaggio sul quale è possibile fare i test. Si può anche usare ogni altro metodo standard del ciclo di vita JSF.

Non c'è alcun supporto per il rendering dei componenti JSF standard, così non è possibile testare facilmente il corpo dei messaggi email.

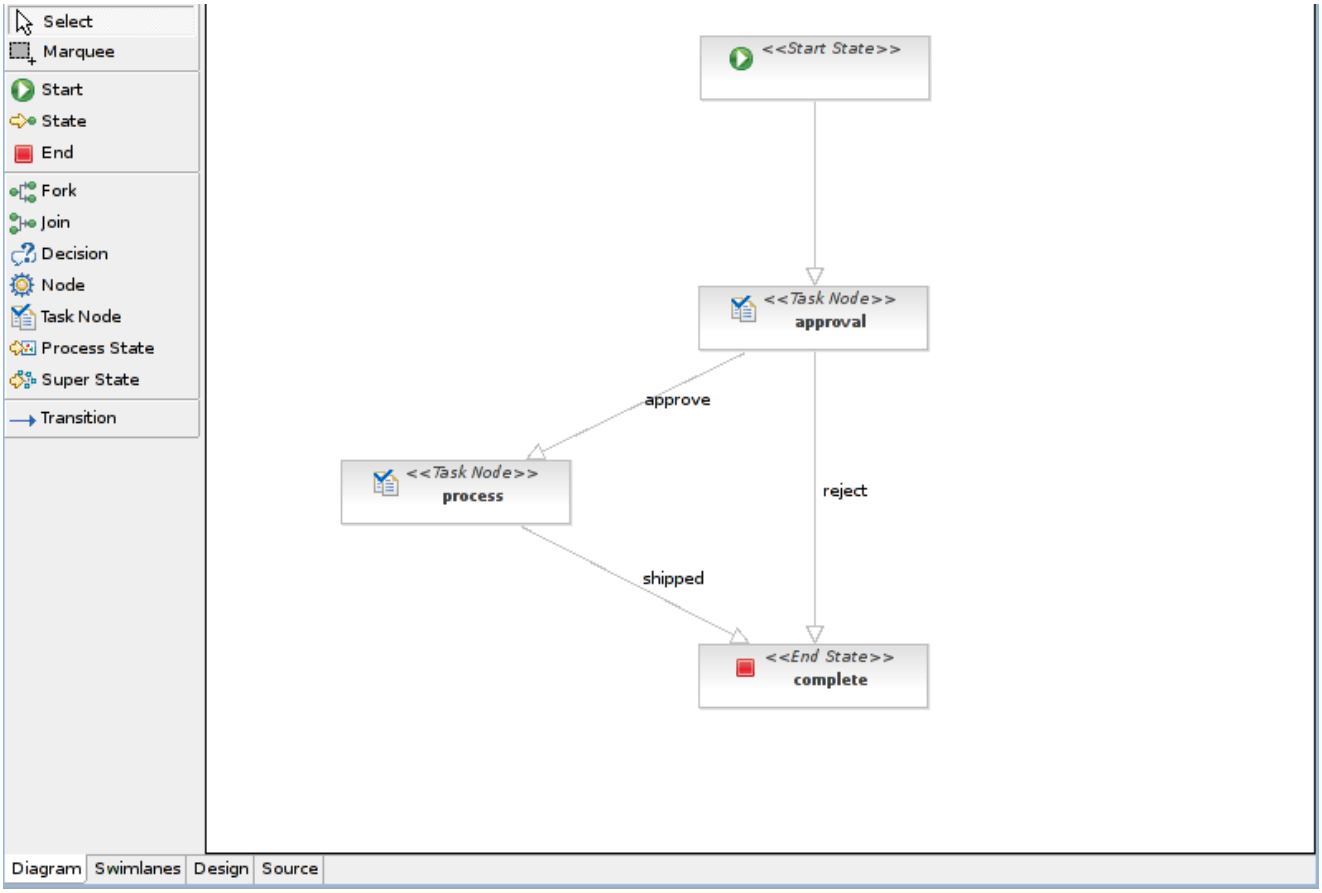
Strumenti di Seam

37.1. Visualizzatore e designer jBPM

Il designer ed il visualizzatore jBPM consentono di progettare e vedere i processi di business ed i pageflow. Quest'ottimo strumento è parte di JBoss Eclipse IDE ed ulteriori dettagli sono disponibili nella documentazione di jBPM (http://docs.jboss.com/jbpm/v3/gpd/)

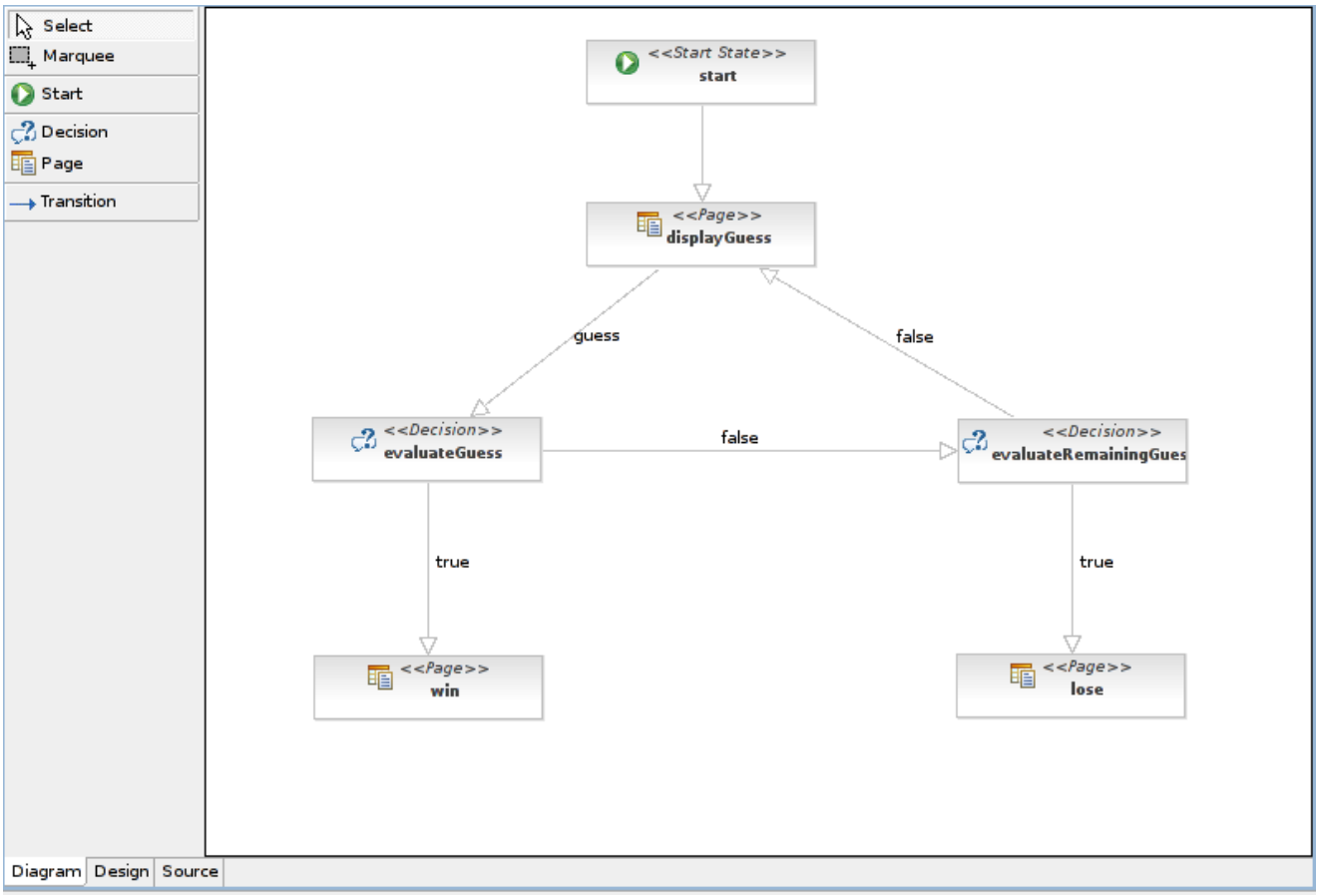
37.1.1. Designer del processo di business

Questo strumento permette di progettare il processo di business in modo grafico.



37.1.2. Visualizzatore Pageflow

Questo strumento permette di progettare i pageflow e di costruire viste grafiche per condividere e scambiare facilmente le idee su quale sia la migliore progettazione.


Select
Marquee
Start
Decision
Page
Transition
<<Start State>>
start
<<Page>>
displayGuess
guess
false
<<Decision>>
evaluateGuess
false
<<Decision>>
evaluateRemainingGues
true
true
<<Page>>
win
<<Page>>
lose
Diagram
Design
Source

Seam on BEA's Weblogic

Weblogic 10.3 is BEA's latest stable JEE5 server offering. Seam applications can be deployed and developed on Weblogic servers, and this chapter will show you how. There are some known issues with the Weblogic servers that will need to be worked around, and configuration changes that are needed specific to Weblogic.

First step is to get Weblogic downloaded, installed and running. Then we'll talk about Seam's JEE5 example and the hurdles to getting it running. After that, the JPA example will be deployed to the server. Then finally we will create a `seam-gen` application and get it up and running to provide a jump start to your own application.

38.1. Installation and operation of Weblogic

First things first we need to get the server installed. There are some outstanding issues that were not addressed in 10.3, but it does solve some of the issues discussed below without the need for BEA patches. The previous release 10.0.MP1 is also available, but requires some BEA patches to function correctly.

- `Weblogic 10.0.MP1` — *Download page* [http://www.oracle.com/technology/software/products/ias/htdocs/wls_main.html]

  10.0.MP1 has some known issues with EJBs that use `varargs` in their methods (it confuses them as `transient`), as well as some others. See *Sezione 38.2.1, «EJB3 Issues with Weblogic»* for full details on the issues and the work around.

- `Weblogic 10.3` — *Download page* [http://www.oracle.com/technology/software/products/ias/htdocs/wls_main.html]

  This is the latest stable release of the Weblogic server, and the one that will be used with the examples below. This version has addressed some of the issues with EJBs that were in 10.0.MP1. However one of the changes did not make it into this release. See *Sezione 38.2.1, «EJB3 Issues with Weblogic»* for the details, but because of this we still need to use the special Weblogic seam jar discussed below.

Special `jboss-seam.jar` for Weblogic EJB Support

Starting with Seam 2.0.2.CR2 a special Weblogic specific jar has been created that does not contain the `TimerServiceDispatcher` . This is the EJB that uses `varargs` and exposes the second EJB issue. We will be using this jar for the `jee5/booking` example, as it avoids the known BEA issues.

38.1.1. Installing 10.3

Here are the quick steps to installing Weblogic 10.3. For more details or if you are having any issues please check with the BEA docs at the *Weblogic 10.3 Doc Center* [http://edocs.bea.com/wls/docs103/] . Here we install the RHEL 5 version using the graphical installer:

1. Follow the link given above for 10.3 and download the correct version for your environment. You will need to sign up for an account with Oracle in order to do this.
2. You may need to change the the `server103_XX.bin` file to be executable:

```
chmod a+x server103_XX.bin
```

3. Execute the install:

```
./server103_XX.bin
```

4. When the graphical install loads, you need to set the BEA home location. This is where all BEA applications are installed. This location will be known as `$BEA_HOME` in this document e.g.:

```
/jboss/apps/bea
```

5. Select `Complete` as the installation type. You do not need all the extras of the complete install (such as struts and beehive libraries), but it will not hurt.
6. You can leave the defaults for the component installation locations on the next page.

38.1.2. Creating your Weblogic domain

A Weblogic domain is similar to a JBoss server configuration - it is a self contained server instance. The Weblogic server you just installed has some example domains, but we are going to create one just for the seam examples. You can use the existing domains if you wish (modify the instructions as needed).

1. Start up the Weblogic configuration wizard:

```
$BEA_HOME/wlserver_10.3/common/bin/config.sh
```

2. Choose to create a new domain, configured to support `Weblogic Server`. Note that this is the default domain option.
3. Set a username and password for this domain.

4. Next choose `Development Mode` and the default JDK when given the option.
5. The next screen asks if you want to customize any setting. Select `No`.
6. Finally set the name of the domain to `seam_examples` and leave the default domain location.

38.1.3. How to Start/Stop/Access your domain

Now that the server is installed and the domain is created you need to know how to start and stop it, plus how to access its configuration console.

- Starting the domain:

  This is the easy part - go to the `$BEA_HOME/user_projects/domains/seam_examples/bin` directory and run the `./startWeblogic.sh` script.

- Accessing the configuration console:

  Launch `http://127.0.0.1:7001/console` in your web browser. It will ask for your username and password that you entered before. We won't get into this much now, but this is the starting point for a lot of the various configurations that are needed later.

- Stopping the domain:

  There are a couple of options here:

  - The recommended way is through the configuration console:

    1. Select `seam_examples` on the left hand side of the console.
    2. Choose the `Control` tab in the middle of the page.
    3. Select the check box `AdminServer` in the table.
    4. Choose `Shutdown` just above the table, and select either `When work completes` or `Force shutdown now` as appropriate.

  - Hitting `Ctrl-C` in the terminal where you started the domain.

    No negative effects have been seen, but we would not recommend doing this while in the middle of configuration changes in the console.

-

A note on Weblogic classloading

When using the `/autodeploy` directory as described in this chapter you may see `NoClassDefFound` exceptions during redeployment's. If you see this try restarting the Weblogic server. If you still see it remove the auto-deployed EAR/WAR files, restart the server, and redeploy. We could not find a specific reason for this, but others seem to be having this issue as well.

38.1.4. Setting up Weblogic's JSF Support

These are the instructions to deploy and configure Weblogic's JSF 1.2 libraries. Out of the box Weblogic does not come with its own JSF libraries active. For complete details see *Weblogic 10.3 Configuring JSF and JSTL Libraries* [http://edocs.bea.com/wls/docs103/webapp/configurejsfandjtsl.html]

1. In the administration console navigate to the `Deployments` page using the left hand menu.
2. Then select the `Install` button at the top of the deployments table
3. Using the directory browser navigate to the `$BEA_HOME/wlserver_10.3/common/deployable-libraries` directory. Then select the `jsf-1.2.war` archive, and click the `Next` button.
4. Make sure that the `Install this deployment as a library` is selected. Click the `Next` button on the `Install Application Assistant` page.
5. Click the `Next` button on the `Optional Settings` page.
6. Make sure that the `Yes, take me to the deployment's configuration screen.` is selected. Click the `Finish` button on the `Review your choices and click Finish` page.
7. On the `Settings for jsf(1.2,1.2.3.1)` page set the `Deployment Order` to `99` so that it is deployed prior to auto deployed applications. Then click the `Save` button.

There is another step that is needed for this to work. For some reason, even with the steps above classes in the `jsf-api.jar` are not found during application deployment. The only way for this to work is to put the `javax.jsf_1.2.0.0.jar` (the jsf-api.jar) from `jsf-1.2.war` in the domains shared library. This requires a restart of the server.

38.2. The `jee5/booking` Example

Do you want to run Seam using EJB's on Weblogic? If so there are some obstacles that you will have to avoid, or some patches that are needed from BEA. This section describes those obstacles and what changes are needed to the `jee5/booking` example to get it deployed and functioning.

38.2.1. EJB3 Issues with Weblogic

For several releases of Weblogic there has been an issue with how Weblogic generates stubs and compiles EJB's that use variable arguments in their methods. This is confirmed in the Weblogic 9.X and 10.0.MP1 versions. Unfortunately the 10.3 version only partially addresses the issue as detailed below.

38.2.1.1. The `varargs` Issue

The basic explanation of the issue is that the Weblogic EJB compiler mistakes methods that use `varargs` as having the `transient` modifier. When BEA generates its own stub class from those classes during deployment it fails and the deployment does not succeed. Seam uses variable

arguments in one of its internal EJB's (`TimerServiceDispatcher`). If you see exceptions like below during deployment you are running an unpatched version of 10.0.MP1.

```
java.io.IOException: Compiler failed executable.exec:
/jboss/apps/bea/wlserver_10.0/user_projects/domains/seam_examples/servers/AdminServer
/cache/EJBCompilerCache/5yo5dk9ti3yo/org/jboss/seam/async/
TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:194: modifier transient
not allowed here
  public transient javax.ejb.Timer scheduleAsynchronousEvent(java.lang.String arg0,
  java.lang.Object[] arg1)
                              ^
/jboss/apps/bea/wlserver_10.0/user_projects/domains/seam_examples/servers/AdminServer
/cache/EJBCompilerCache/5yo5dk9ti3yo/org/jboss/seam/async/
TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:275: modifier transient
not allowed here
  public transient javax.ejb.Timer scheduleTimedEvent(java.lang.String arg0,
  org.jboss.seam.async.TimerSchedule arg1, java.lang.Object[] arg2)
```

This issue has been fixed in Weblogic 10.3, and BEA has created a patch for Weblogic 10.0.MP1 (`CR327275`) for this issue that can be requested from their support.

Unfortunately a second issue has been reported and verified by BEA.

38.2.1.2. Missing EJB Methods Issue

This issue was only found once the `CR327275` patch had been applied to 10.0.MP1. This new issue has been confirmed by BEA and they created a patch for 10.0.MP1 that addresses this issue. This patch has been referred to as both `CR370259` and `CR363182`. As with the other patch this can be requested through the BEA support.

This issue causes certain EJB methods to be incorrectly left out of Weblogic's generated internal stub classes. This results in the following error messages during deployment.

```
<<Error
> <EJB
> <BEA-012036
> <Compiling generated EJB classes produced the following Java compiler error message:
<Compilation Error
> TimerServiceDispatcher_qzt5w2_Impl.java: The type TimerServiceDispatcher_qzt5w2_Impl
must implement the inherited abstract method
TimerServiceDispatcher_qzt5w2_Intf.scheduleTimedEvent(String, Schedule, Object[])
<Compilation Error
> TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java: Type mismatch:
cannot convert from Object to Timer
```

```
<Compilation Error
> TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java: Type mismatch:
 cannot convert from Object to Timer
>
<Error
> <Deployer
> <BEA-149265
> <Failure occurred in the execution of deployment request with ID '1223409267344' for task '0'.
 Error is: 'weblogic.application.ModuleException: Exception preparing module: EJBModule(jboss-
seam.jar)
```

It appears that when Weblogic 10.3 was released the neglected to include this fix!! This means that Weblogic 10.0.MP1 with patches will function correctly, but 10.3 will still require the special Seam jar described below. Not all users have seen this and there may be certain combinations of OS/JRE that do not see this, however is has been seen many times. Hopefully Oracle/BEA will release an updated patch for this issue on 10.3. When they do we will update this reference guide as needed.

So that Seam's users can deploy an EJB application to Weblogic a special Weblogic specific jar has been created, starting with Seam 2.0.2.CR2. It is located in the `$SEAM/lib/interop` directory and is called `jboss-seam-wls-compatible.jar` . The only difference between this jar and the `jboss-seam.jar` is that it does not contain the `TimerServiceDispatcher` EJB. To use this jar simply rename the `jboss-seam-wls-compatible.jar` to `jboss-seam.jar` and replace the original in your applications `EAR` file. The `jee5/booking` example demonstrates this. Obviously with this jar you will not be able to use the `TimerServiceDispatcher` functionality.

38.2.2. Getting the `jee5/booking` Working

In this section we will go over the steps needed to get the `jee5/booking` example to up and running.

38.2.2.1. Setting up the hsql datasource

This example uses the in memory hypersonic database, and the correct data source needs to be set up. The admin console uses a wizard like set of pages to configure it.

1. Copy `hsqldb.jar` to the Weblogic domain's shared library directory: `cp $SEAM_HOME/lib/hsqldb.jar $BEA_HOME/user_projects/domains/seam_examples/lib`
2. Start up the server and navigate to the administration console following *Sezione 38.1.3, «How to Start/Stop/Access your domain»*
3. On the left side tree navigate `seam_examples - Services- JDBC - Data Sources`.
4. Then select the `New` button at the top of the data source table
5. Fill in the following:

a. Name: seam-jee5-ds

b. JNDI Name: seam-jee5-ds

c. Database Type and Driver: other

d. Select Next button

6. Select Next button on the Transaction Options page

7. Fill in the following on the Connection Properties page:

a. Database Name: hsqldb

b. Host Name: 127.0.0.1

c. Port: 9001

d. Username: sa saranno campi password vuoti.

e. Password: lasciare vuoto.

f. Select Next button

8. Fill in the following on the Connection Properties page:

a. Driver Class Name: org.hsqldb.jdbcDriver

b. URL: jdbc:hsqldb:.

c. Username: sa

d. Password: lasciare vuoto.

e. Leave the rest of the fields as is.

f. Select Next button

9. Choose the target domain for the data source in our case the only one AdminServer. Click Next.

38.2.2.2. Configuration and Build changes

OK - now we are ready to finally begin adjusting the seam application for deployment to the Weblogic server.

resources/META-INF/persistence.xml

- Change the jta-data-source to what you entered above :

```
        <jta-data-source
>seam-jee5-ds</jta-data-source
>
```

- Then comment out the glassfish properties.
- Then add these two properties for weblogic support.

```
<property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect"/>
<property name="hibernate.transaction.manager_lookup_class"
        value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

`resources/META-INF/weblogic-application.xml`

- This file needs to be created and should contain the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<weblogic-application>
   <library-ref>
      <library-name
>jsf</library-name>
      <specification-version
>1.2</specification-version>
      <implementation-version
>1.2</implementation-version>
      <exact-match
>false</exact-match>
   </library-ref>
   <prefer-application-packages>
      <package-name
>antlr.*</package-name>
   </prefer-application-packages>
</weblogic-application>
```

- These changes do two two different things. The first element `library-ref` tells weblogic that this application will be using the deployed JSF libraries. The second element `prefer-application-packages` tells weblogic that the `antlr` jars take precedence. This avoids a conflict with hibernate.

resources/META-INF/ejb-jar.xml

- The changes described here work around an issue where Weblogic is only using a single instance of the sessionBeanInterceptor for all session beans. Seam's interceptor caches and stores some component specific attributes, so when a call comes in - the interceptor is primed for a different component and an error is seen. To solve this problem you must define a separate interceptor binding for each EJB you wish to use. When you do this Weblogic will use a separate instance for each EJB.

  Modify the assembly-descriptor element to look like this:

```
<assembly-descriptor>
  <interceptor-binding
>
    <ejb-name
>AuthenticatorAction</ejb-name>
    <interceptor-class
>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding
>
    <ejb-name
>BookingListAction</ejb-name>
    <interceptor-class
>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding
>
    <ejb-name
>RegisterAction</ejb-name>
    <interceptor-class
>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding
>
    <ejb-name
>ChangePasswordAction</ejb-name>
    <interceptor-class
>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding
>
    <ejb-name
```

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

`resources/WEB-INF/weblogic.xml`

- This file needs to be created and should contain the following:

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

<weblogic-web-app
>
<library-ref>
   <library-name
>jsf</library-name>
   <specification-version
>1.2</specification-version>
   <implementation-version
>1.2</implementation-version>
   <exact-match
>false</exact-match>
 </library-ref>
</weblogic-web-app>
```

- This file and the element library-ref tells Weblogic that this application will using the deployed JSF libraries. This is needed in both this file and the weblogic-application.xml file because both applications require access.

38.2.2.3. Building and Deploying the Application

There are some changes needed to the build script and the jboss-seam.jar then we can deploy the app.

build.xml

- We need to add the follow so that the weblogic-application.xml will be packaged.

```
<!-- Resources to go in the ear -->
<fileset id="ear.resources" dir="${resources.dir}">
   <include name="META-INF/application.xml" />
   <include name="META-INF/weblogic-application.xml" />
   <include name="META-INF/*-service.xml" />
   <include name="META-INF/*-xmbean.xml" />
   <include name="treecache.xml" />
   <include name="*.jpdl.xml" />
   <exclude name=".gpd.*" />
   <include name="*.cfg.xml" />
   <include name="*.xsd" />
</fileset
>
```

$SEAM/lib/interop/jboss-seam-wls-compatible.jar

- This is the change discussed above in *Sezione 38.2.1, «EJB3 Issues with Weblogic»* . There are really two options.

- Rename this jar and replace the original $SEAM/lib/jboss-seam.jar file. This approach does not require any changes to the packaged EAR archive, but overwrites the original jboss-seam.jar

- The other option is the modify the packaged EAR archive and replace the jboss-seam.jar in the archive manually. This leaves the original jar alone, but requires a manual step when ever the archive is packaged.

Assuming that you choose the first option for handling the jboss-seam-wls-compatible.jar we can build the application by running ant archive at the base of the jee5/booking example directory.

Because we chose to create our Weblogic domain in development mode we can deploy the application by putting the EAR file in the domains autodeploy directory.

```
cp ./dist/jboss-seam-jee5.ear
            $BEA_HOME/user_projects/domains/seam_examples/autodeploy
```

Check out the application at `http://localhost:7001/seam-jee5/`

38.3. The `jpa` booking example

This is the Hotel Booking example implemented with Seam POJOs and Hibernate JPA and does not require EJB3 support to run. The example already has a breakout of configurations and build scripts for many of the common containers including Weblogic 10.X

First we'll build the example for Weblogic 10.x and do the needed steps to deploy. Then we'll talk about what is different between the Weblogic versions, and with the JBoss AS version.

Note that this example assumes that Weblogic's JSF libraries have been configured as described in *Sezione 38.1.4, «Setting up Weblogic's JSF Support»*.

38.3.1. Building and deploying `jpa` booking example

Step one setup the datasource, step two build the app, step three deploy.

38.3.1.1. Setting up the datasource

The Weblogic 10.X version of the example will use the in memory hsql database instead of the built in PointBase database. If you wish to use the PointBase database you must setup a PointBase datasource, and adjust the hibernate setting in `persistence.xml` to use the PointBase dialect. For reference the `jpa/weblogic92` example uses PointBase.

Configuring the datasource is very similar to the jee5 *Sezione 38.2.2.1, «Setting up the hsql datasource»* . Follow the steps in that section, but use the following entries where needed.

- DataSource Name: `seam-jpa-ds`
- JNDI Name: `seam-jpa-ds`

38.3.1.2. Building the example

Building it only requires running the correct ant command:

```
ant weblogic10
```

This will create a container specific distribution and exploded archive directories.

38.3.1.3. Deploying the example

When we installed Weblogic following *Sezione 38.1.2, «Creating your Weblogic domain»* we chose to have the domain in development mode. This means to deploy the application all we need to do is copy it into the autodeploy directory.

```
cp ./dist-weblogic10/jboss-seam-jpa.war
$BEA_HOME/user_projects/domains/seam_examples/autodeploy
```

Check out the application at the following `http://localhost:7001/jboss-seam-jpa/` .

38.3.2. What's different with Weblogic 10.x

- Between the the Weblogic 10.x and 9.2 examples there are several differences:
  - `META-INF/persistence.xml` — The 9.2 version is configured to use the `PointBase` database and a pre-installed datasource. The 10.x version uses the `hsql` database and a custom datasource.
  - `WEB-INF/weblogic.xml` — This file and its contents solve an issue with an older version of the `ANTLR` libraries that Weblogic 10.x uses internally. OC4J have the same issue as well. It also configures the application to use the shared JSF libraries that were installed above.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/90
                    http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd">
  <library-ref>
    <library-name
>jsf</library-name>
    <specification-version
>1.2</specification-version>
    <implementation-version
>1.2</implementation-version>
    <exact-match
>false</exact-match>
  </library-ref>
  <container-descriptor>
    <prefer-web-inf-classes
```

```
>true</prefer-web-inf-classes>
  </container-descriptor>
</weblogic-web-app
>
```

This make Weblogic use classes and libraries in the web application before other libraries in the classpath. Without this change hibernate is required to use a older, slower query factory by setting the following property in the `META-INF/persistence.xml` file.

```
<property name="hibernate.query.factory_class"
    value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory"/>
```

- `WEB-INF/components.xml` — In the Weblogic 10.x version JPA entity transactions is enabled by adding:

```
 <transaction:entity-transaction entity-manager="#{em}"/>
```

- `WEB-INF/web.xml` — Because the `jsf-impl.jar` is not in the `WAR` this listener need to be configured :

```
 <listener>
   <listener-class
>com.sun.faces.config.ConfigureListener</listener-class>
 </listener
>
```

- Between the Weblogic 10.x version and the JBoss version there are more changes. Here is the rundown:
  - `META-INF/persistence.xml` — Except for datasource name the Weblogic version sets:

```
<property name="hibernate.transaction.manager_lookup_class"
  value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

- WEB-INF/lib — The Weblogic version requires several library packages because they are not included as they are with JBoss AS. These are primarily for hibernate, and its dependencies.
  - To use Hibernate as your JPA provider you need the following jars:
    - hibernate.jar
    - hibernate-annotations.jar
    - hibernate-entitymanager.jar
    - hibernate-validator.jar
    - jboss-common-core.jar
    - commons-logging.jar
    - commons-collections.jar
    - jboss-common-core.jar
  - Various third party jars that Weblogic needs:
    - antlr.jar
    - cglib.jar
    - asm.jar
    - dom4j.jar
    - el-ri.jar
    - javassist.jar
    - concurrent.jar

38.4. Deploying an application created using seam-gen on Weblogic 10.x

seam-gen is a very useful tool for developers to quickly get an application up and running, and provides a foundation to add your own functionality. Out of box seam-gen will produce applications configured to run on JBoss AS. These instructions will show the steps needed to get it to run on Weblogic.

seam-gen was build for simplicity so, as you can imagine, deploying an application generated by seam-gen to Weblogic 10.x is not too hard. Basically it consists of updating or removing some configuration files, and adding dependent jars that Weblogic 10.x does not ship with.

This example will cover the basic `seam-gen WAR` deployment. This will demonstrate Seam POJO components, Hibernate JPA, Facelets, Drools security, RichFaces, and a configurable dataSource.

38.4.1. Running `seam-gen` setup

The first thing we need to do it tell `seam-gen` about the project we want to make. This is done by running `./seam setup` in the base directory of the Seam distribution. Note the paths here are my own, feel free to change for you environment.

```
./seam setup
Buildfile: build.xml

init:

setup:
     [echo] Welcome to seam-gen :-)
    [input] Enter your Java project workspace (the directory that contains your
Seam projects) [C:/Projects] [C:/Projects]
/home/jbalunas/workspace
    [input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]
/jboss/apps/jboss-4.2.3.GA
    [input] Enter the project name [myproject] [myproject]
weblogic-example
     [echo] Accepted project name as: weblogic_example
    [input] Select a RichFaces skin (not applicable if using ICEFaces) [blueSky]
 ([blueSky], classic, ruby, wine, deepMarine, emeraldTown, sakura, DEFAULT)

    [input] Is this project deployed as an EAR (with EJB components) or a WAR
(with no EJB support) [ear]  ([ear], war, )
war
    [input] Enter the Java package name for your session beans [org.jboss.seam.
tutorial.weblogic.action] [org.jboss.seam.tutorial.weblogic.action]
org.jboss.seam.tutorial.weblogic.action
    [input] Enter the Java package name for your entity beans [org.jboss.seam.
tutorial.weblogic.model] [org.jboss.seam.tutorial.weblogic.model]
org.jboss.seam.tutorial.weblogic.model
    [input] Enter the Java package name for your test cases [org.jboss.seam.
tutorial.weblogic.action.test] [org.jboss.seam.tutorial.weblogic.action.test]
org.jboss.seam.tutorial.weblogic.test
    [input] What kind of database are you using? [hsql]  ([hsql], mysql, oracle,
 postgres, mssql, db2, sybase, enterprisedb, h2)
```

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

    [input] Enter the filesystem path to the JDBC driver jar [/tmp/seamlib/hsqldb.jar]
[/tmp/seam/lib/hsqldb.jar]

    [input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
 [org.hsqldb.jdbcDriver]

    [input] Enter the JDBC URL for your database [jdbc:hsqldb:.] [jdbc:hsqldb:.]

    [input] Enter database username [sa] [sa]

    [input] Enter database password [] []

    [input] Enter the database schema name (it is OK to leave this blank) [] []

    [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], )

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

    [input] Enter your ICEfaces home directory (leave blank to omit ICEfaces) [] []

[propertyfile] Creating new property file:
/rhdev/projects/jboss-seam/cvs-head/jboss-seam/seam-gen/build.properties
     [echo] Installing JDBC driver jar to JBoss server
     [copy] Copying 1 file to /jboss/apps/jboss-4.2.3.GA/server/default/lib
     [echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
```

Type `./seam new-project` to create your project and `cd /home/jbalunas/workspace/weblogic_example` to see the newly created project.

38.4.2. What to change for Weblogic 10.X

First we change and delete some configuration files, then we update the libraries that are deployed with the application.

38.4.2.1. Configuration file changes

`build.xml`

- Change the default target to `archive`.

```
<project name="weblogic_example" default="archive" basedir="."
>
```

`resources/META-INF/persistence-dev.xml`

- Alter the `jta-data-source` to be `seam-gen-ds` (and use this as the `jndi-name` when creating the data source in Weblogic's admin console)
- Change the transaction type to `RESOURCE_LOCAL` so that we can use JPA transactions.

```
<persistence-unit name="weblogic_example" transaction-type="RESOURCE_LOCAL"
>
```

- Add/modify the properties below for Weblogic support:

```
<property name="hibernate.cache.provider_class"
   value="org.hibernate.cache.HashtableCacheProvider"/>
<property name="hibernate.transaction.manager_lookup_class"
   value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

- You'll need to alter `persistence-prod.xml` as well if you want to deploy to Weblogic using the prod profile.

`resource/WEB-INF/weblogic.xml`

You will need to create this file and populate it following *description of WEB-INF/weblogic.xml [611]*.

`resource/WEB-INF/components.xml`

We want to use JPA transactions so we need to add the following to let Seam know.

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

You will also need to add the transaction namespace and schema location to the top of the document.

```
xmlns:transaction="http://jboss.com/products/seam/transaction"
```

```
http://jboss.com/products/seam/transaction http://jboss.com/products/seam/transaction-2.1.xsd
```

`resource/WEB-INF/web.xml`

`WEB-INF/web.xml` — Because the `jsf-impl.jar` is not in the `WAR` this listener need to be configured :

```
 <listener>
  <listener-class
>com.sun.faces.config.ConfigureListener</listener-class>
 </listener
>
```

`resources/WEB-INF/jboss-web.xml`

You can delete this file as we aren't deploying to JBoss AS (`jboss-app.xml` is used to enable classloading isolation in JBoss AS)

`resources/*-ds.xml`

You can delete these files as we aren't deploying to JBoss AS. These files define datasources in JBoss AS, in Weblogic we will use the administration console.

38.4.2.2. Library changes

The `seam-gen` application has very similar library dependencies as the `jpa` example above. See *Sezione 38.3.2, «What's different with Weblogic 10.x»*. Below is the changes that are needed to get them in this application.

- build.xml — Now we need to adjust the `build.xml`. Find the target `war` and add the following to the end of the target.

```
    <copy todir="${war.dir}/WEB-INF/lib">
      <fileset dir="${lib.dir}">
        <!-- Misc 3rd party -->
```

```
        <include name="commons-logging.jar" />
        <include name="dom4j.jar" />
        <include name="javassist.jar" />
        <include name="cglib.jar" />
        <include name="antlr.jar" />

        <!-- Hibernate -->
        <include name="hibernate.jar" />
        <include name="hibernate-commons-annotations.jar" />
        <include name="hibernate-annotations.jar" />
        <include name="hibernate-entitymanager.jar" />
        <include name="hibernate-validator.jar" />
        <include name="jboss-common-core.jar" />
        <include name="concurrent.jar" />
      </fileset>
    </copy
>
```

38.4.3. Building and Deploying your application

All that's left is deploying the application. This involves setting up a data source, building the app, and deploying it.

38.4.3.1. Setting up the data source

Configuring the datasource is very similar to the jee5 *Sezione 38.2.2.1, «Setting up the hsql datasource»*. Except for what is listed here follow that instruction from the link.

- DataSource Name: `seam-gen-ds`
- JNDI Name: `seam-gen-ds`

38.4.3.2. Building the application

This is as easy as typing `ant` in the projects base directory.

38.4.3.3. Deploying the example

When we installed Weblogic following *Sezione 38.1.2, «Creating your Weblogic domain»* we chose to have the domain in development mode. This means to deploy the application all we need to do is copy it into the autodeploy directory.

```
cp ./dist/weblogic_example.war /jboss/apps/bea/user_projects/domains/seam_examples/autodeploy
```

Check out the application at the following `http://localhost:7001/weblogic_example/`.

Seam on IBM's Websphere AS

Websphere AS V7 is IBM's application server offering. This release is fully Java EE 5 certified.

First we will go over some basic information about the Websphere AS environment that we used for these examples. We will go over the details of those steps with the JEE5 booking example. We will also deploy the JPA example application.

39.1. Websphere AS environment and deployment information

Websphere AS is a commercial product and so we will not discuss the details of its installation other than to say follow the directions provided by your particular installation type and license. This section will detail the exact server versions used, installation tips, and some custom properties that are needed for all of the examples.

39.1.1. Installation versions

All of the examples and information in this chapter are based on the version V7 of Websphere AS at the time of this writing.

- *Websphere Application Server V7* [http://www.ibm.com/developerworks/downloads/ws/was]

After installing Websphere AS, create server profile with Profile Management Tool, if you didn't create profile in installation process.

39.2. The `jee5/booking` example

The `jee5/booking` example is based on the Hotel Booking example (which runs on JBoss AS). Out of the box it is designed to run on Glassfish, but with the steps below it can be deployed to Websphere. It is located in the `$SEAM_DIST/examples/jee5/booking` directory.

39.2.1. Configuration file changes

Below are the configuration file changes that are need to the base example.

`resources/WEB-INF/components.xml`
: We need to change the way that we look up EJBs for WAS. We need to remove the `/local` from the end of the `jndi-pattern` attribute. It should look like this:

```
<core:init jndi-pattern="java:comp/env/jboss-seam-jee5/#{ejbName}" debug="true"/>
```

`resources/META-INF/ejb-jar.xml`

We need to replace the /local string from `ejb-ref-name`. See at the following final code:

```
<enterprise-beans>
  <!-- EJB reference required when one Seam EJB component references another Seam EJB component using @In -->
  <!-- Not required if you inject using @EJB, but then you lose state management and client-side interceptors (e.g., security) -->
  <session>
    <ejb-name>RegisterAction</ejb-name>
    <ejb-local-ref>
      <ejb-ref-name>jboss-seam-jee5/AuthenticatorAction</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local>org.jboss.seam.example.booking.Authenticator</local>
    </ejb-local-ref>
  </session>
</enterprise-beans>
```

`resources/WEB-INF/web.xml`

We have to make some changes to the EJB references in the `web.xml`. These changes are what will allow WAS to bind automatically the EJB3 references in the web module to the the actual EJB3 beans in the EAR module. Replace all of the /local strings in `ejb-local-refs` when the values below.

```
<!-- JEE5 EJB3 names -->
<ejb-local-ref>
  <ejb-ref-name>jboss-seam-jee5/AuthenticatorAction</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>org.jboss.seam.example.booking.Authenticator</local>
</ejb-local-ref>
```

```
  <ejb-local-ref>
    <ejb-ref-name
>jboss-seam-jee5/BookingListAction</ejb-ref-name>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.example.booking.BookingList</local>
  </ejb-local-ref>

  <ejb-local-ref>
    <ejb-ref-name
>jboss-seam-jee5/RegisterAction</ejb-ref-name>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.example.booking.Register</local>
  </ejb-local-ref>

  <ejb-local-ref>
    <ejb-ref-name
>jboss-seam-jee5/ChangePasswordAction</ejb-ref-name>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.example.booking.ChangePassword</local>
  </ejb-local-ref>

  <ejb-local-ref>
    <ejb-ref-name
>jboss-seam-jee5/HotelBookingAction</ejb-ref-name>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.example.booking.HotelBooking</local>
  </ejb-local-ref>

  <ejb-local-ref>
    <ejb-ref-name
>jboss-seam-jee5/HotelSearchingAction</ejb-ref-name>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.example.booking.HotelSearching</local>
  </ejb-local-ref>
```

```
 <ejb-local-ref>
  <ejb-ref-name
>jboss-seam-jee5/EjbSynchronizations</ejb-ref-name>
  <ejb-ref-type
>Session</ejb-ref-type>
  <local
>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref
>
```

Note also that `EjbSynchronizations` is a built-in Seam EJB and not part of the Hotel Booking example. This means that if your application's `components.xml` specifies `transaction:ejb-transaction`, then you must include:

```
 <ejb-local-ref>
  <ejb-ref-name
>myapp/EjbSynchronizations</ejb-ref-name>
  <ejb-ref-type
>Session</ejb-ref-type>
  <local-home
></local-home>
  <local
>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref
>
```

in your web.xml. If you don't include it, you'll get the following error:

```
Name comp/env/myapp/EjbSynchronizations not found in context java:
```

`resources/META-INF/persistence.xml`

For this example we will be using the default datasource that comes with WAS. To do this change the `jta-data-source` element:

```
<jta-data-source
>DefaultDatasource</jta-data-source>
```

Then we need to adjust some of the hibernate properties. First comment out the Glassfish properties. Next you need to add/change the properties:

```
<!--<property name="hibernate.transaction.flush_before_completion" value="true"/>-->
<property name="hibernate.cache.provider_class"
          value="org.hibernate.cache.HashtableCacheProvider"/>
<property name="hibernate.dialect" value="GlassfishDerbyDialect"/>
<property name="hibernate.transaction.manager_lookup_class"
      value="org.hibernate.transaction.WebSphereExtendedJTATransactionLookup"/>
```

- `hibernate.transaction.manager_lookup_class` — Standard Hibernate transaction manager property for WAS 6.X and 7
- `hibernate.transaction.flush_before_completion` — This is commented out because we want the container to manage the transactions. Also if this is set to `true` an exception will be thrown by WAS when the EJBContext is looked up.

```
com.ibm.wsspi.injectionengine.InjectionException:
            EJBContext may only be looked up by or injected into an EJB
```

- `hibernate.dialect` — From WAS 6.1.0.9 on the embedded DB was switched to the same Derby DB as is in Glassfish v2.

`src/GlassfishDerbyDialect.java`

You will need to get the `GlassfishDerbyDialect.java` and copy it into the `/src` directory. The java class exists in the JPA example source directory and can be copied using the command below assuming you are in `jee5/booking` directory:

```
cp ../../jpa/src/GlassfishDerbyDialect.java
  ./src
```

This class will be put into the `jboss-seam-jee5.jar` file.

`resources/import.sql`

This file must also be copied from the JPA example because either the Derby DB or the dialect does not support changes to the `ID` column. The files are identical except for the column difference. Use the following command to make the copy

```
cp ../../jpa/resources-websphere7/import.sql ./resources
```

39.2.2. Building the `jee5/booking` example

In order to get the changes we have made into our application we need to make some changes to the `build.xml`. There are also some additional jars that are required by our application in order to work with WAS. This section will cover what changes are needed to the `build.xml`.

39.2.2.1. Library dependency changes

We remove the `log4j.jar` so that all of the log output from our application will be added to the WAS log. Additional steps are required to fully configure log4j and those are outside of the scope of this document.

39.2.2.2. Updating the `build.xml` file

Add the following entry to the bottom of the `build.xml` file. This overrides the default fileset that is used to populate the `jboss-seam-jee5.jar`. :

```
  <fileset id="jar.resources" dir="${resources.dir}">
    <include name="import.sql" />
    <include name="seam.properties" />
    <include name="META-INF/persistence.xml" />
    <include name="META-INF/ejb-jar.xml" />
  </fileset
>
```

Now all that is left is to execute the `ant archive` task and the built application will be in the `jee5/booking/dist` directory.

39.2.3. Deploying the application to Websphere

So now we have everything we need in place. All that is left is to deploy it - just a few steps more.

For this we will use Websphere's administration console. As before there are some tricks and tips that must be followed.

The steps below are for the WAS version stated above. The ports are default values, if you changed them substitute your values.

1. Log in to the administration console

```
https://localhost:9043/admin
```

or

```
http://localhost:9060/admin
```

2. Access the `Websphere enterprise applications` menu option under the `Applications --> Application Type` left side menu.
3. At the top of the `Enterprise Applications` table select `Install`. Below are installation wizard pages and what needs to done on each:
   - `Preparing for the application installation`
     - Browse to the `examples/jee5/booking/dist/jboss-seam-jee5.ear` file using the file upload widget.
     - Select the `Next` button.
     - Select the `Fast Path` button.
     - Select the `Next` button.
   - `Select installation options`
     - Select the `Deploy enterprise beans` and `Allow EJB reference targets to resolve automatically` check boxes. This is needed unless you used a Websphere AS tool to package the application.
     - Select the `Next` button.
   - `Map modules to servers`
     - No changes needed here as we only have one server. Select the `Next` button.
   - `Summary`
     - No changes needed here. Select the `Finish` button.
   - `Installation`
     - Now you will see it installing and deploying your application.
     - When it finishes select the `Save` link and you will be returned to the `Enterprise Applications` table.

4. Now that we have our application installed we need to make some adjustments to it before we can start it:
   - Starting from the `Enterprise Applications` table select the `Seam Booking` link.
   - Select the `Manage Modules` link.
   - Select the `jboss-seam-jee5-booking.war` link.
   - Change the `Class loader order` combo box to `Classes loaded with application class loader first (parent last)`.
   - Select `Apply` and then `Save` options.
   - Return to the `Seam Booking` page.
   - On this page select the `Class loading and update detection` link.
   - Select the radio button for `Classes loaded with application class loader first`.
   - Select `Apply` and then `Save` options.
5. To start the application return to the `Enterprise Applications` table and select our application in the list. Then choose the `Start` button at the top of the table.
6. You can now access the application at `http://localhost:9080/seam-jee5-booking/index.html`.

39.3. The `jpa` booking example

Thankfully getting the `jpa` example to work is much easier than the `jee5` example. This is the Hotel Booking example implemented in Seam POJOs and using Hibernate JPA with JPA transactions. It does not use EJB3.

The example already has a breakout of configurations and build scripts for many of the common containers including Websphere.

First thing we are going to do is build and deploy that example. Then we'll go over some key changes that we needed.

39.3.1. Building the `jpa` example

Building it only requires running the correct ant command:

```
ant websphere7
```

This will create container specific distribution and exploded archive directories with the `websphere7` label.

39.3.2. Deploying the jpa example

This is similar to the jee5 example at *Sezione 39.2.3, «Deploying the application to Websphere»*, but without so many steps.

- From the Enterprise Applications table select the Install button.
  - Preparing for the application installation
    - Browse to the examples/jpa/dist-websphere7/jboss-seam-jpa.war file using the file upload widget.
    - Select the Fast Path button.
    - Select the Next button.
  - Select the Next button for the next three pages, no changes are needed.
  - Map context roots for Web modules
    - In the Context root text box enter jboss-seam-jpa.
    - Select the Next button.
  - Summary page
    - Review the settings if you wish and select the Finish button to install the application. When installation finished select the Save link and you will be returned to the Enterprise Applications table.
- As with the jee5 example there are some class loader changes needed before we start the application. Follow the instructions at *installation adjustments for jee5 example* but exchange jboss-seam-jpa_war for Seam Booking.
- Finally start the application by selecting it in the Enterprise Applications table and clicking the Start button.
- You can now access the application at the http://localhost:9080/jboss-seam-jpa/index.html.

39.3.3. What's different for Websphere AS V7

The differences between the JPA examples that deploys to JBoss 4.2 and Websphere AS V7 are mostly expected; library and configuration file changes.

- Configuration file changes
  - META-INF/persistence.xml — the main changes here are for the datasource JNDI path, switching to the Websphere transaction manager look up class, and changing the hibernate dialect to be GlassfishDerbyDialect .

- `WEB-INF/components.xml` — the change here is `jndi-pattern` without /local string.
- `META-INF/ejb-jar.xml` — the same change in `ejb-ref-name`, where is replace /local string in `jboss-seam-jee5/AuthenticatorAction`.
- `src/GlassfishDerbyDialect.java` — this class is needed for the hibernate dialect change to `GlassfishDerbyDialect`
- `import.sql` — either for the dialect or Derby DB the `ID` column can not be populated by this file and was removed.

- Changes for dependent libraries

  The Websphere version requires several library packages because they are not included as they are with JBoss AS. These are primarily for hibernate and their dependencies. Below are listed only the additional jars needed above and beyond the JBoss `JPA` example.

  - To use Hibernate as your JPA provider you need the following jars:
    - `hibernate.jar`
    - `hibernate-annotations.jar`
    - `hibernate-commons-annotations.jar`
    - `hibernate-entitymanager.jar`
    - `hibernate-validator.jar`
    - `commons-collections.jar`
    - `jboss-common-core.jar`
  - Various third party jars that Websphere needs:
    - `antlr.jar`
    - `cglib.jar`
    - `asm.jar`
    - `dom4j.jar`
    - `javassist.jar`
    - `concurrent.jar`

39.4. Deploying an application created using seam-gen on Websphere V7

seam-gen is a very useful tool for developers to quickly get an application up and running, and provides a foundation to add your own functionality. Out of box seam-gen will produce applications configured to run on JBoss AS. These instructions will show the steps needed to get it to run on Websphere. As stated above in *Sezione 39.2, «The jee5/booking example »* there are some tricky changes needed to get an EJB3 application running. This section will take you through the exact steps.

39.4.1. Running seam-gen Setup

The first step is setting up seam-gen to construct the base project. There are several choices made below, specifically the datasource and hibernate values that we will adjust once the project is created.

```
./seam setup
Buildfile: build.xml

init:

setup:
     [echo] Welcome to seam-gen :-)
    [input] Enter your Java project workspace (the directory that contains your
Seam projects) [C:/Projects] [C:/Projects]
/home/jbalunas/workspace
    [input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]
/home/jbalunas/jboss/jboss-4.2.3.GA
    [input] Enter the project name [myproject] [myproject]
websphere_example
     [echo] Accepted project name as: websphere_example
    [input] Do you want to use ICEFaces instead of RichFaces [n] (y, [n], )

    [input] skipping input as property icefaces.home.new has already been set.
    [input] Select a RichFaces skin [blueSky] ([blueSky], classic, ruby, wine,
deepMarine, emeraldTown, sakura, DEFAULT)

    [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 [org.jboss.seam.
```

```
tutorial.websphere.action] [org.jboss.seam.tutorial.websphere.action]
org.jboss.seam.tutorial.websphere.action
    [input] Enter the Java package name for your entity beans [org.jboss.seam.
tutorial.websphere.model] [org.jboss.seam.tutorial.websphere.model]
org.jboss.seam.tutorial.websphere.model
    [input] Enter the Java package name for your test cases [org.jboss.seam.
tutorial.websphere.action.test] [org.jboss.seam.tutorial.websphere.action.test]
org.jboss.seam.tutorial.websphere.test
    [input] What kind of database are you using? [hsql]  ([hsql], mysql, oracle,
 postgres, mssql, db2, sybase, enterprisedb, h2)

    [input] Enter the Hibernate dialect for your database [org.hibernate.
dialect.HSQLDialect] [org.hibernate.dialect.HSQLDialect]

    [input] Enter the filesystem path to the JDBC driver jar [/tmp/seam/lib/hsqldb.jar]
[/tmp/seam/lib/hsqldb.jar]

    [input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
[org.hsqldb.jdbcDriver]

    [input] Enter the JDBC URL for your database [jdbc:hsqldb:.]
[jdbc:hsqldb:.]

    [input] Enter database username [sa] [sa]

    [input] Enter database password [] []

    [input] Enter the database schema name (it is OK to leave this blank) [] []

    [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], )

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

[propertyfile] Creating new property file:
/rhdev/projects/jboss-seam/svn-seam_2_0/jboss-seam-2_0/seam-gen/build.properties
     [echo] Installing JDBC driver jar to JBoss server
     [copy] Copying 1 file to /home/jbalunas/jboss/jboss-4.2.3.GA/server/default/lib
     [echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
```

```
Total time: 3 minutes 5 seconds
```

Type `./seam new-project` to create your project and `cd /home/jbalunas/workspace/websphere_example` to the newly created structure.

39.4.2. Changes needed for deployment to Websphere

We now need to make some changes to the generated project.

39.4.2.1. Configuration file changes

`resources/META-INF/persistence-dev.xml`

- Alter the `jta-data-source` to be `DefaultDatasource`. We are going to be using the integrated Websphere DB.
- Add or change the properties below. These are described in detail at *Sezione 39.2, «The `jee5/booking` example »*:

```
<property name="hibernate.dialect" value="GlassfishDerbyDialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.cache.provider_class"
      value="org.hibernate.cache.HashtableCacheProvider"/>
<property name="hibernate.transaction.manager_lookup_class"
      value="org.hibernate.transaction.WebSphereExtendedJTATransactionLookup"/>
```

- Remove the JBoss AS specific method of exposing the EntityManagerFactory:

```
<property
 name="jboss.entity.manager.factory.jndi.name"
 value="java:/websphere_exampleEntityManagerFactory"
>
```

- You'll need to alter `persistence-prod.xml` as well if you want to deploy to Websphere using the prod profile.

`src/GlassfishDerbyDialect.java`

As with other examples we need to include this java class for DB support. It can be copied from the `jpa` example into the `websphere_example/src` directory.

```
cp $SEAM/examples/jpa/src/GlassfishDerbyDialect.java
  ./src
```

`resources/META-INF/jboss-app.xml`
: You can delete this file as we aren't deploying to JBoss AS (`jboss-app.xml` is used to enable classloading isolation in JBoss AS)

`resources/*-ds.xml`
: You can delete these file as we aren't deploying to JBoss AS (these files define datasources in JBoss AS, we are using Websphere's default datasource)

`resources/WEB-INF/components.xml`

- Enable container managed transaction integration - add the `<transaction:ejb-transaction />` component, and it's namespace declaration `xmlns:transaction="http://jboss.com/products/seam/transaction"`
- Alter the `jndi-pattern` to `java:comp/env/websphere_example/#{ejbName}`
- We do not need `managed-persistence-context` for this example and so can delete its entry.

```
<persistence:managed-persistence-context name="entityManager"
        auto-create="true"
       persistence-unit-jndi-name="java:/websphere_exampleEntityManagerFactory"/>
```

`resources/WEB-INF/web.xml`
: As with the `jee5/booking` example we need to add EJB references to the web.xml. These references require replacing /local string in `ejb-ref-name` to flag them for Websphere to perform the proper binding.

```
 <ejb-local-ref>
  <ejb-ref-name
>websphere_example/AuthenticatorAction</ejb-ref-name>
  <ejb-ref-type
>Session</ejb-ref-type>
  <local
>org.jboss.seam.tutorial.websphere.action.Authenticator</local>
 </ejb-local-ref
>
```

```
  <ejb-local-ref>
    <ejb-ref-name
>websphere_example/EjbSynchronizations</ejb-ref-name
>
    <ejb-ref-type
>Session</ejb-ref-type>
    <local
>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
  </ejb-local-ref
>
```

39.4.2.2. Creating the `AuthenticatorAction` EJB

We want to take the existing `Authenticator` Seam POJO component and create an EJB3 out of it.

1. Change the generated Authenticator class
   - Rename the class to `AuthenticatorAction`
   - Add the `@Stateless` annotation to the new `AuthenticatorAction` class.
   - Create an interface called `Authenticator` which `AuthenticatorAction` implements (EJB3 requires session beans to have a local interface). Annotate the interface with `@Local` , and add a single method with same signature as the `authenticate` in `AuthenticatorAction` .

```
@Name("authenticator") @Stateless public class
        AuthenticatorAction implements Authenticator {
```

```
@Local public interface Authenticator {
  public boolean authenticate();
}
```

2. We've already added its reference to the `web.xml` file so are good to go.

39.4.2.3. Extra jar dependencies and other changes to the `build.xml`

This application has similar requirements as the `jee5/booking` example.

- Change the default target to `archive` (we aren't going to cover automatic deployment to Websphere).

```
<project name="websphere_example" default="archive" basedir="."
>
```

- Websphere looks for the drools `/security.drl` file in the root of the `war` file instead of the root of the `websphere_example.jar` so we need to have the `build.xml` move it to the correct location at build time. The following must be added at the top of the `<target name="war" depends="compile" description="Build the distribution .war file">` target.

```
<copy todir="${war.dir}">
  <fileset dir="${basedir}/resources" >
     <include name="*.drl" />
  </fileset>
</copy
>
```

- Now we need to get extra jars into the `build.xml`. Look for the `<fileset dir="${basedir}">` section of the task below. Add the new includes at the bottom of the fileset.

```
<target name="ear" description="Build the EAR">
   <copy todir="${ear.dir}">
        <fileset dir="${basedir}/resources">
              <include name="*jpdl.xml" />
              <include name="*hibernate.cfg.xml" />
              <include name="jbpm.cfg.xml" />
        </fileset>

        <fileset dir="${basedir}">
              <include name="lib/jbpm*.jar" />
              <include name="lib/jboss-el.jar" />
              <include name="lib/drools-*.jar"/>
              <include name="lib/core.jar"/>
              <include name="lib/janino*.jar"/>
              <include name="lib/antlr-*.jar"/>
              <include name="lib/mvel*.jar"/>
             <include name="lib/richfaces-api*.jar" />
        </fileset>
   </copy>
   <copy todir="${ear.dir}/META-INF">
```

```
        <fileset dir="${basedir}/resources/META-INF">
            <include name="application.xml" />
            <include name="jboss-app.xml" />
        </fileset>
    </copy>
</target
>
```

- Dipendenze Hibernate

```
    <!-- Hibernate and deps -->
    <include name="lib/hibernate.jar"/>
    <include name="lib/hibernate-commons-annotations.jar"/>
    <include name="lib/hibernate-annotations.jar"/>
    <include name="lib/hibernate-entitymanager.jar"/>
    <include name="lib/hibernate-validator.jar"/>
    <include name="lib/jboss-common-core.jar" />
```

- Dipendenze di terze parti

```
    <!-- 3rd party and supporting jars -->
    <!--<include name="lib/log4j.jar" />-->
    <include name="lib/javassist.jar"/>
    <include name="lib/dom4j.jar" />
    <include name="lib/concurrent.jar" />
    <include name="lib/cglib.jar"/>
    <include name="lib/asm.jar"/>
    <include name="lib/antlr.jar" />
    <include name="lib/commons-logging.jar" />
    <include name="lib/commons-collections.jar" />
```

- `jboss-seam.jar` - this is needed in the `ear` base directory.

```
    <!-- seam jar -->
    <include name="lib/jboss-seam.jar" />
```

You should end up with something like:

```
<fileset dir="${basedir}">

  <include name="lib/jbpm*.jar" />
  <include name="lib/jboss-el.jar" />
  <include name="lib/drools-*.jar"/>
  <include name="lib/core.jar"/>
  <include name="lib/janino*.jar"/>
  <include name="lib/antlr-*.jar"/>
  <include name="lib/mvel*.jar"/>
  <include name="lib/richfaces-api*.jar" />

  <!-- Hibernate and deps -->
  <include name="lib/hibernate.jar"/>
  <include name="lib/hibernate-commons-annotations.jar"/>
  <include name="lib/hibernate-annotations.jar"/>
  <include name="lib/hibernate-entitymanager.jar"/>
  <include name="lib/hibernate-validator.jar"/>
  <include name="lib/jboss-common-core.jar" />

  <!-- 3rd party and supporting jars -->
  <include name="lib/javassist.jar"/>
  <include name="lib/dom4j.jar" />
  <include name="lib/concurrent.jar" />
  <include name="lib/cglib.jar"/>
  <include name="lib/asm.jar"/>
  <include name="lib/antlr.jar" />
  <include name="lib/commons-logging.jar" />
  <include name="lib/commons-collections.jar" />

  <!-- seam jar -->
  <include name="lib/jboss-seam.jar" />

</fileset
>
```

39.4.2.4. Building and deploying the seam-gen'd application to Websphere

- Build your application by calling `ant` in the base directory of your project (ex. `/home/jbalunas/workspace/websphere_example`). The target of the build will be `dist/websphere_example.ear` .

- To deploy the application follow the instructions here : *Sezione 39.2.3, «Deploying the application to Websphere»* but use references to this project `websphere_example` instead of `jboss-seam-jee5`.
- Checkout the app at: `http://localhost:9080/websphere_example/index.html`

Seam on GlassFish application server

GlassFish is an open source application server which fully implements Java EE 5. The latest stable release is v2 UR2.

First, we'll discuss the GlassFish environment. Then we will go over the how you deploy the jee5 example. Next, we will deploy the JPA example application. Finally we show how to get a seam-gen's generated application running on GlassFish.

40.1. GlassFish environment and deployment information

40.1.1. Installazione

All of the examples and information in this chapter are based on the the latest version of GlassFish at the time of this writing.

- *GlassFish v2 UR2 - download page* [https://glassfish.dev.java.net/downloads/v2ur2-b04.html]

After downloading GlassFish, install it:

```
$ java -Xmx256m -jar glassfish-installer-v2ur2-b04-linux.jar
```

After installing, setup GlassFish:

```
$ cd glassfish; ant -f setup.xml
```

The created domain's name is `domain1`.

Next, we start the embedded JavaDB server:

```
$ bin/asadmin start-database
```

Nota

JavaDB is an embedded database that is included with GlassFish, just as HSQLDB is included in JBoss AS.

Now, start the GlassFish server:

```
$ bin/asadmin start-domain domain1
```

The web administration console is available at `http://localhost:4848/`. You can access the web admin console with the default username (`admin`) and password (`adminadmin`). We will be using the the admin console to deploy our examples. You can also copy EAR/WAR files to the `glassfish/domains/domain1/autodeploy` directory to deploy them, although we are not going to cover that.

You can stop the server and database using:

```
$ bin/asadmin stop-domain domain1; bin/asadmin stop-database
```

40.2. The `jee5/booking` example

The `jee5/booking` example is based on the Hotel Booking example (which runs on JBoss AS). Out of the box it is also designed to run on GlassFish. It is located in `$SEAM_DIST/examples/jee5/booking`.

40.2.1. Building the `jee5/booking` example

To build the example, simply execute the default `ant` target:

```
$ ant
```

in the `examples/jee5/booking` directory. This will create the `dist` and `exploded-archives` directories.

40.2.2. Deploying the application to GlassFish

We will deploy the application on GlassFish using the GlassFish admin console.

1. Log in to the admin console at `http://localhost:4848`
2. Access the `Enterprise Applications` in the menu option under the `Applications` left side menu.
3. At the top of the `Enterprise Application` table select `Deploy`. Follow through the wizard, using these hints:
   - `Preparing for the application installation`
     - Browse to `examples/jee5/booking/dist/jboss-seam-jee5.ear`.

- Selezionare il pulsante OK.

4. You can now access the application at `http://localhost:8081/seam-jee5/`.

40.3. The jpa booking example

This is the Hotel Booking example implemented in Seam POJOs and using Hibernate JPA with JPA transactions. It does not require EJB3 support to run on application server.

The example already has a break-out of configurations and build scripts for many of the common containers including GlassFish.

40.3.1. Building the jpa example

To build the example, use the `glassfish` target:

```
$ ant glassfish
```

This will create the container specific `dist-glassfish` and `exploded-archives-glasfish` directories.

40.3.2. Deploying the jpa example

This is very similar to the `jee5` example at *Sezione 40.2.2, «Deploying the application to GlassFish»* except that this is a `war` and not an `ear`.

- Log in to the administration console:

```
http://localhost:4848
```

- Access the `Web Applications` in the menu option under the `Applications` left side menu.
  - `Preparing for the application installation`
    - Browse to `examples/jpa/dist-glassfish/jboss-seam-jpa.war`.
    - Selezionare il pulsante OK.
- You can now access the application at `http://localhost:8081/jboss-seam-jpa/`.

Using Derby instead of Hypersonic SQL DB

In order for the app to work out of the box with GlassFish, we have used the Derby (aka JavaDB) database in GlassFish. However, we strongly recommend that you use another database (e.g. HSQL). `examples/jpa/resources-glassfish/WEB-INF/classes/GlassfishDerbyDialect.class` is a hack to get around a Derby

bug in GlassFish server. You must use it as your Hibernate dialect if you use Derby with GlassFish.

40.3.3. What's different for GlassFish v2 UR2

- Cambiamenti al file di configurazione
  - `META-INF/persistence.xml` — the main changes needed are the datasource JNDI, switching to the GlassFish transaction manager lookup class, and changing the hibernate dialect to be `GlassfishDerbyDialect`.
  - `WEB-INF/classes/GlassfishDerbyDialect.class` — this class is needed for the Hibernate dialect change to `GlassfishDerbyDialect`
  - `import.sql` — either for the dialect or Derby DB the `ID` column can not be populated by this file and was removed.

40.4. Deploying an application generated by `seam-gen` on GlassFish v2 UR2

`seam-gen` is a very useful tool for developers to quickly get an application up and running, and provides a foundation to add your own functionality. Out of box `seam-gen` will produce applications configured to run on JBoss AS. These instructions will show the steps needed to get it to run on GlassFish.

40.4.1. Eseguire il setup in `seam-gen`

The first step is setting up `seam-gen` to construct the base project. There are several choices made below, specifically the datasource and hibernate values that we will adjust once the project is created.

```
$ ./seam setup
Buildfile: build.xml

init:

setup:
[echo] Welcome to seam-gen :-)
[input] Enter your Java project workspace (the directory that contains your
Seam projects) [C:/Projects] [C:/Projects]
/projects
[input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]
```

```
[input] Enter the project name [myproject] [myproject]
seamgen_example
[echo] Accepted project name as: seamgen_example
[input] Do you want to use ICEfaces instead of RichFaces [n] (y, [n])

[input] skipping input as property icefaces.home.new has already
been set.
[input] Select a RichFaces skin [blueSky] ([blueSky], classic, ruby, wine,
deepMarine, emeraldTown, japanCherry, DEFAULT)

[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.seamgen_example] [com.mydomain.seamgen_example]
org.jboss.seam.tutorial.glassfish.action
[input] Enter the Java package name for your entity beans
[org.jboss.seam.tutorial.glassfish.action]
[org.jboss.seam.tutorial.glassfish.action]
org.jboss.seam.tutorial.glassfish.model
[input] Enter the Java package name for your test cases
[org.jboss.seam.tutorial.glassfish.action.test]
[org.jboss.seam.tutorial.glassfish.action.test]
org.jboss.seam.tutorial.glassfish.test
[input] What kind of database are you using? [hsql] ([hsql], mysql, oracle,
postgres, mssql, db2, sybase, enterprisedb, h2)

[input] Enter the Hibernate dialect for your database
[org.hibernate.dialect.HSQLDialect]
[org.hibernate.dialect.HSQLDialect]

[input] Enter the filesystem path to the JDBC driver jar
[/tmp/seam/lib/hsqldb.jar] [/tmp/seam/lib/hsqldb.jar]

[input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
[org.hsqldb.jdbcDriver]

[input] Enter the JDBC URL for your database [jdbc:hsqldb:.]
[jdbc:hsqldb:.]

[input] Enter database username [sa] [sa]

[input] Enter database password [] []
```

```
[input] Enter the database schema name (it is OK to leave this blank) [] []

[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])

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

[propertyfile] Creating new property file:
/home/mnovotny/workspaces/jboss/jboss-seam/seam-gen/build.properties
[echo] Installing JDBC driver jar to JBoss server
[copy] Copying 1 file to
/home/mnovotny/workspaces/jboss/jboss-seam/seam-gen/C:/Program
Files/jboss-4.2.3.GA/server/default/lib
[echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
Total time: 4 minutes 5 seconds
```

Type `$ ./seam new-project` to create your project and then `cd /projects/seamgen_example` to the newly created structure.

40.4.2. Changes needed for deployment to GlassFish

We now need to make some changes to the generated project.

40.4.2.1. Cambiamenti al file di configurazione

`resources/META-INF/persistence-dev.xml`

- Alter the `jta-data-source` to be `jdbc/__default`. We are going to be using the integrated GlassFish Derby DB.
- Replace all of the properties with the following. The key differences are briefly described in *Sezione 40.3.3, «What's different for GlassFish v2 UR2»*:

```
<property name="hibernate.dialect" value="GlassfishDerbyDialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property                                    name="hibernate.cache.provider_class"
 value="org.hibernate.cache.HashtableCacheProvider"/>
```

```
<property name="hibernate.transaction.manager_lookup_class"
 value="org.hibernate.transaction.SunONETransactionManagerLookup"/>
```

- You'll need to alter `persistence-prod.xml` as well if you want to deploy to GlassFish using the prod profile.

`resources/GlassfishDerbyDialect.class`

As with other examples we need to include this class for DB support. It can be copied from the `jpa` example into the `seamgen_example/resources` directory.

```
$ cp \
$SEAM_DIST/examples/jpa/resources-glassfish/WEB-INF/classes/
GlassfishDerbyDialect.class \
./resources
```

`resources/META-INF/jboss-app.xml`

You can delete this file as we aren't deploying to JBoss AS (`jboss-app.xml` is used to enable classloading isolation in JBoss AS)

`resources/*-ds.xml`

You can delete these file as we aren't deploying to JBoss AS (these files define data sources in JBoss AS, we are using GlassFish's default data source)

`resources/WEB-INF/components.xml`

- Enable container managed transaction integration - add the `<transaction:ejb-transaction/>` component, and it's namespace declaration `xmlns:transaction="http://jboss.com/products/seam/transaction"`
- Alter the `jndi-pattern` to `java:comp/env/seamgen_example/#{ejbName}`

`resources/WEB-INF/web.xml`

As with the `jee5/booking` example, we need to add EJB references to web.xml. Technically, the reference type is not required, but we add it here for good measure. Note that these references require the presence of an empty `local-home` element to retain compatibility with a JBoss AS 4.x deployment.

```
<ejb-local-ref
>
   <ejb-ref-name
>seamgen_example/AuthenticatorAction</ejb-ref-name
>
   <ejb-ref-type
>Session</ejb-ref-type
```

```
>
   <local-home/>
   <local
>org.jboss.seam.tutorial.glassfish.action.Authenticator</local
>
 </ejb-local-ref>

 <ejb-local-ref>
   <ejb-ref-name
>seamgen_example/EjbSynchronizations</ejb-ref-name
>
   <ejb-ref-type
>Session</ejb-ref-type>
   <local-home/>
   <local
>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref
>
```

Keep in mind that if you are deploying to JBoss AS 4.x, and have defined the EJB references shown above in your web.xml, you will need to also define local JNDI names for each of them in jboss-web.xml, as shown below. This step is not required when deploying to GlassFish, but it's mentioned here in case you are also deploying the application to JBoss AS 4.x (not required for JBoss AS 5).

```
<ejb-local-ref
>
   <ejb-ref-name
>seamgen_example/AuthenticatorAction</ejb-ref-name
>
   <local-jndi-name
>AuthenticatorAction</local-jndi-name
>
 </ejb-local-ref>

 <ejb-local-ref>
   <ejb-ref-name
>seamgen_example/EjbSynchronizations</ejb-ref-name
>
   <local-jndi-name
>EjbSynchronizations</local-jndi-name>
 </ejb-local-ref
```

```
>
```

40.4.2.2. Creating the `AuthenticatorAction` EJB

We want to take the existing `Authenticator` Seam POJO component and create an EJB3 out of it.

1. • Rename the class to `AuthenticatorAction`
   • Add the `@Stateless` annotation to the new `AuthenticatorAction` class.
   • Create an interface called `Authenticator` which `AuthenticatorAction` implements (EJB3 requires session beans to have a local interface). Annotate the interface with `@Local`, and add a single method with same signature as the `authenticate` in `AuthenticatorAction`.

```
@Name("authenticator")
@Stateless
public class AuthenticatorAction implements Authenticator {
```

```
@Local
public interface Authenticator {

   public boolean authenticate();
}
```

2. We've already added its reference to the `web.xml` file so we are good to go.

40.4.2.3. Extra jar dependencies and other changes to the `build.xml`

This application has similar requirements as the `jee5/booking` example.

- Change the default target to `archive` (we aren't going to cover automatic deployment to GlassFish).

```
<project name="seamgen_example" default="archive" basedir="."
>
```

- We need to get the `GlassfishDerbyDialect.class` into our application jar. To do that find the `jar` task and add the `GlassfishDerbyDialect.class` line as shown below:

```
<target name="jar" depends="compile,copyclasses" description="Build the distribution .jar
 file">
```

```
  <copy todir="${jar.dir}">
    <fileset dir="${basedir}/resources">
      <include name="seam.properties" />
      <include name="*.drl" />
      <include name="GlassfishDerbyDialect.class" />
    </fileset
>
  </copy>
  ...
```

- Now we need to get extra jars into the `ear` file. Look for the `<copy todir="${ear.dir}/lib" >` section of the `ear` target. Add the following to the child `<fileset dir="${lib.dir}" >` element.

- • Aggiungere dipendenze Hibernate

```
<!-- Hibernate and deps -->
<include name="hibernate.jar"/>
<include name="hibernate-commons-annotations.jar"/>
<include name="hibernate-annotations.jar"/>
<include name="hibernate-entitymanager.jar"/>
<include name="hibernate-validator.jar"/>
<include name="jboss-common-core.jar"/>
```

  - Aggiungere dipendenze di terze parti

```
<!-- 3rd party and supporting jars -->
<include name="javassist.jar"/>
<include name="dom4j.jar"/>
<include name="concurrent.jar" />
<include name="cglib.jar"/>
<include name="asm.jar"/>
<include name="antlr.jar" />
<include name="commons-logging.jar" />
<include name="commons-collections.jar" />
```

  You should end up with something like:

```
<fileset dir="${lib.dir}">
  <includesfile name="deployed-jars-ear.list" />

  <!-- Hibernate and deps -->
```

```
    <include name="hibernate.jar"/>
    <include name="hibernate-commons-annotations.jar"/>
    <include name="hibernate-annotations.jar"/>
    <include name="hibernate-entitymanager.jar"/>
    <include name="hibernate-validator.jar"/>
    <include name="jboss-common-core.jar" />

    <!-- 3rd party and supporting jars -->
    <include name="javassist.jar" />
    <include name="dom4j.jar" />
    <include name="concurrent.jar" />
    <include name="cglib.jar" />
    <include name="asm.jar" />
    <include name="antlr.jar" />
    <include name="commons-logging.jar" />
    <include name="commons-collections.jar" />
  </fileset
  >
```

40.4.2.4. Building and deploying the seam-gen'd application to GlassFish

- Build your application by calling `ant` in the base directory of your project (for example `/projects/seamgen-example`). The target of the build will be `dist/seamgen-example.ear`.
- To deploy the application follow the instructions here *Sezione 40.2.2, «Deploying the application to GlassFish»* but use references to this project `seamgen-example` instead of `jboss-seam-jee5`.
- Controllare l'applicazione all'indirizzo `http://localhost:8081/seamgen_example/`

Dipendenze

41.1. Dipendenze JDK

Seam non funziona con JDK 1.4 e richiede JDK 5 o superiore, poiché impiega annotazioni ed altre caratteristiche di JDK 5.0. Seam è stato testato usando i JDK di Sun. Comunque con Seam non ci sono problemi noti usando altri JDK.

41.1.1. Considerazioni su JDK 6 di Sun

Le prime versioni di JDK 6 di Sun contenevano un versione incompatibile di JAXB e richiedevano l'override di questa usando la directory "endorsed". La release JDK6 Update 4 di SUn ha aggiornato JAXB 2.1 e rimosso questo requisito. In fase di building, di testing o di esecuzione assicurarsi di utilizzare questa versione o successive.

Seam usa JBoss Embedded nei propri test di unità e di integrazione. JBoss Embedded ha un requisito aggiuntivo con JDK 6; occorre impostare il seguente argomento JVM:

```
-Dsun.lang.ClassLoader.allowArraySyntax=true
```

. Il sistema di build interno a Seam imposta di default questo valore quando si esegue la suit di test. Comunque usando JBoss Embedded per i test va impostato questo valore.

41.2. Dipendenze del progetto

Questa sezione elenca le dipendenze di Seam sia a compile-time sia a runtime. Laddove il tipo viene elencato come `ear`, la libreria deve essere inclusa nella directory /lib del proprio ear dell'applicazione. Laddove il tipo viene elencato come `war`, la libreria deve essere collocata nella directory `/WEB-INF/lib` del proprio file war. Lo scope della dipendenze è tutto, runtime o provided (da JBoss AS 4.2 o 5.0).

Le informazioni sulla versione e sulle dipendenze non sono incluse nella documentazione, ma sono fornite in `/dependency-report.txt` che viene generato dai POM di Maven memorizzati in `/build`. E' possibile generare questo file eseguendo `ant dependencyReport`.

41.2.1. Core

Tabella 41.1.

Nome	Scope	Tipo	NOte
`jboss-seam.jar`	tutti	ear	La libreria base di Seam è sempre richiesta.

Nome	Scope	Tipo	NOte
`jboss-seam-debug.jar`	runtime	war	I
`jboss-seam-ioc.jar`	runtime	war	Richiesto con l'uso di Seam con Spring
`jboss-seam-pdf.jar`	runtime	war	Richiesto con l'uso delle funzionalità PDF di Seam
`jboss-seam-excel.jar`	runtime	war	Richiesto con l'uso delle funzionalità di Microsoft® Excel® in Seam
`jboss-seam-rss.jar`	runtime	war	Richiesto con l'uso delle funzionalità di generazione RSS in Seam
`jboss-seam-remoting.jar`	runtime	war	Richiesto con l'uso di Seam Remoting
`jboss-seam-ui.jar`	runtime	war	Richiesto con l'uso dei controlli JSF in Seam
`jsf-api.jar`	fornito		JSF API
`jsf-impl.jar`	fornito		Implementazione di riferimento JSF
`jsf-facelets.jar`	runtime	war	Facelets
`urlrewrite.jar`	runtime	war	Libreria URL Rewrite
`quartz.jar`	runtime	ear	Richiesto per usare Quartz assieme alle funzionalità asincrone di Seam

41.2.2. RichFaces

Tabella 41.2. Dipendenze di RichFaces

Nome	Scope	Tipo	NOte
`richfaces-api.jar`	tutti	ear	Richiesto per l'uso di RichFaces. Fornisce classi API per l'uso nella propria applicazione, per esempio per creare un albero
`richfaces-impl.jar`	runtime	war	Richiesto per l'uso di RichFaces.
`richfaces-ui.jar`	runtime	war	Richiesto per l'uso di RichFaces. Fornisce tutti i componenti UI.

41.2.3. Seam Mail

Tabella 41.3. Dipendenze di Seam Mail

Nome	Scope	Tipo	NOte
`activation.jar`	runtime	ear	Richiesto per il supporto agli allegati
`mail.jar`	runtime	ear	Richiesto per il supporto alle email d'uscita
`mail-ra.jar`	solo compilazione		Richiesto per il supporto alle email d'ingresso mail-ra.rar dovrebbe essere deployato nell'application server a runtime
`jboss-seam-mail.jar`	runtime	war	Seam Mail

41.2.4. Seam PDF

Tabella 41.4. Dipendenze di Seam PDF

Nome	Tipo	Scope	NOte
`itext.jar`	runtime	war	Libreria PDF
`jfreechart.jar`	runtime	war	Libreria grafici
`jcommon.jar`	runtime	war	Richiesto da JFreeChart
`jboss-seam-pdf.jar`	runtime	war	Libreria principale PDF di Seam

41.2.5. Seam Microsoft® Excel®

Tabella 41.5. Dipendenze di Microsoft® Excel® in Seam

Nome	Tipo	Scope	NOte
`jxl.jar`	runtime	war	Libreria JExcelAPI
`jboss-seam-excel.jar`	runtime	war	Libreria base Microsoft® Excel® in Seam

41.2.6. Supporto Seam RSS

Tabella 41.6. Dipendenze di Seam RSS

Nome	Tipo	Scope	NOte
`yarfraw.jar`	runtime	war	Libreria YARFRAW RSS

Nome	Tipo	Scope	NOte
JAXB	runtime	war	Librerie JAXB XML per il parsing
http-client.jar	runtime	war	Librerie Apache HTTP Client
commons-io	runtime	war	Libreria Apache commons IO
commons-lang	runtime	war	Libreria Apache commons lang
commons-codec	runtime	war	Libreria Apache commons codec
commons-collections	runtime	war	Libreria Apache commons collections
jboss-seam-rss.jar	runtime	war	Libreria Seam RSS core

41.2.7. JBoss Rules

Le librerie di JBoss Rules libraries sono nella directory `drools/lib` di Seam.

Tabella 41.7. Dipendenze di JBoss Rules

Nome	Scope	Tipo	NOte
antlr-runtime.jar	runtime	ear	Libreria ANTLR Runtime
core.jar	runtime	ear	Eclipse JDT
drools-compiler.jar	runtime	ear	
drools-core.jar	runtime	ear	
janino.jar	runtime	ear	
mvel.jar	runtime	ear	

41.2.8. JBPM

Tabella 41.8. Dipendenze JBPM

Nome	Scope	Tipo	NOte
jbpm-jpdl.jar	runtime	ear	

41.2.9. GWT

Queste librerie sono richieste se si desidera usare Google Web Toolkit (GWT) nella propria applicazione Seam.

Tabella 41.9. Dipendenze GWT

Nome	Scope	Tipo	NOte
gwt-servlet.jar	runtime	war	Librerie GWT Servlet

41.2.10. Spring

Queste librerie sono richieste se si desidera usare Spring Framework nella propria applicazione Seam.

Tabella 41.10. Dipendenze in Spring Framework

Nome	Scope	Tipo	NOte
`spring.jar`	runtime	ear	Libreria Spring Framework

41.2.11. Groovy

Queste librerie sono richieste se si desidera usare Groovy nella propria applicazione Seam.

Tabella 41.11. Dipendenze di Groovy

Nome	Scope	Tipo	NOte
`groovy-all.jar`	runtime	ear	Librerie di Groovy

41.3. Gestione delle dipendenze usando Maven

Maven offre un supporto per la gestione transitiva delle dipendenze e può essere usato per gestire le dipendenze nei progetti Seam. Maven Ant Tasks intergra Maven nel build di Ant, e Maven può essereimpiegato per fare ilbuild ed il deploy dei propri progetti.

Qui non si discute l'uso di Maven, ma soltanto un utilizzo base del POM.

Le versioni rilasciate di Seam sono disponibili all'indirizzo *http://repository.jboss.org/maven2* e gli snapshot notturni sono disponibili all'indirizzo *http://snapshots.jboss.org/maven2*.

Tutti gli artifact di Seam sono disponibili 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-remoting</artifactId>
</dependency
>
```

```
<dependency>
  <groupId
>org.jboss.seam</groupId>
  <artifactId
>jboss-seam-ioc</artifactId>
</dependency
>
```

```
<dependency>
  <groupId
>org.jboss.seam</groupId>
  <artifactId
>jboss-seam-ioc</artifactId>
</dependency
>
```

Questo POM d'esempio fornirà Seam, JPA (tramite Hibernate) e Hibernate Validator:

```
<?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 Repository</name>
      <url
>http://repository.jboss.org/maven2</url>
    </repository>
  </repositories>

  <dependencies>

    <dependency>
      <groupId
>org.hibernate</groupId>
      <artifactId
>hibernate-validator</artifactId>
      <version
>3.0.0.GA</version>
    </dependency>

    <dependency>
      <groupId
>org.hibernate</groupId>
      <artifactId
```

```
>hibernate-annotations</artifactId>
      <version
>3.3.0.ga</version>
    </dependency>

    <dependency>
      <groupId
>org.hibernate</groupId>
      <artifactId
>hibernate-entitymanager</artifactId>
      <version
>3.3.1.ga</version>
    </dependency>

    <dependency>
      <groupId
>org.jboss.seam</groupId>
      <artifactId
>jboss-seam</artifactId>
      <version
>2.0.0.GA</version>
    </dependency>

  </dependencies>

</project
>
```