GUIDA PER LO SVILUPPATORE

Regola kit

Autori:
Nicola Santi

18 settembre 2008

Indice

Capitolo 1

Getting Started

Questo capitolo è una cura per gli impazienti; seguendo le istruzioni dei paragrafi seguenti sarete in grado di installare e predisporre una prima applicazioni web con Regola.

1.1 Installare Regola kit

Le applicazioni realizzate con Regola kit utilizzano Maven 2 per tutta la gestione del ciclo di build (compilazione, esecuzione dei test, creazione dei file war, ...). Quindi come prima cosa bisogna installare Maven 2 scaricandolo dal sito maven.apache.org e seguendo le istruzioni.

Finita l'installazione di Maven 2 verificate che tutto sia andato a buon fine aprendo una console di terminale e lanciando il seguente comando.

```
1 nicola@casper:~# mvn -version
  Maven version: 2.0.8
3 Java version: 1.6.0_03
  OS name: "linux" version: "2.6.22-14-generic" arch: "i386" Family: "unix"
```

Se tutto è corretto Maven 2 risponde al comando restituendo la sua versione, quella di Java ed infine alcune informazioni circa il sistema operativo in uso.

Regola kit non richiede nessuna installazione particolare (anche se è possibile scaricare un pacchetto contente documentazione e comandi di utilità): quindi finita l'installazione di Maven 2 siete pronti già per utilizzare Regola kit.

■ Per maggiori informazioni su come installare Regola kit sulle vostre macchine di sviluppo si rimanda al capitolo 2 nella pagina 17. ■

1.2 Predisporre il database

Per questo tutorial ipotizziamo di avere a disposizione un database di tipo MySql già installato sulla macchina dove intendiamo creare il progetto. Assicuratevi che il database stia funzionando e digitate il comando seguente:

```
nicola@casper:~# mysql -u root -p
```

Vi troverete dentro la shell di amministrazione di MySql. Approfittatene per creare un nuovo database che sarà utilizzato dall'applicazione digitando il comando.

```
1 mysql> create database clienti;
```

(Attenzione al punto e virgola in fondo al comando). A questo punto non ci resta che creare anche l'utente utilizzato dalla nostra applicazione per accedere al database (nell'esempio porta il mio nome *nicola*).

```
1 mysql> grant all on clienti.* to 'nicola'@'localhost';
```

Bene, con il database abbiamo finito. Digitate questo ultimo comando per uscire dalla shell di amministrazione di MySql e passare al paragrafo seguente.

```
1 mysql> exit
```

■ Regola kit è in grado di utilizzare diversi DBMS (ad esempio Oracle, Microsoft Sql, PostgreSQL, Hypersonic, ...). Per sapere come configurare la vostra applicazione per utilizzare DBMS diversi da MySql si rimanda al capitolo 5 nella pagina 27. ■

1.3 Creare un progetto con Regola kit

Posizionatevi nella cartella dove volete creare il vostro nuovo progetto e digitate un comando simile al seguente con l'accortezza però di modificare il parametro gruopId (nell'esempio *com.acme*) con il nome del vostro package di default ed il parametro artifactId (nell'esempio *clienti*) con il nome del nuovo progetto.

```
1 nicola@casper:~# mvn archetype:create -DarchetypeGroupId=org.regola  \
  -DarchetypeArtifactId=regola-jsf-archetype \
3 -DarchetypeVersion=1.1-SNAPSHOT -DgroupId=com.acme \
  -DartifactId=clienti
```

Attenzione: il comando qui sopra è riparito su diverse righe per chiarezza tipografica, deve invece essere digitato su una sola riga.

La prima volta che lanciate questo comando Maven 2 scarica tutte le librerie necessarie (la cosa potrebbe prendere un po' di tempo) e crea una sotto cartella col nome del progetto (nell'esempio la cartella clienti). Il progetto è

questo punto è già stato creato, posizioniatevi all'interno della cartella *clienti* col comando:

```
nicola@casper:~# cd clienti
```

1.4 Collegarsi al database

Prima di lanciare la nostra nuova applicazione è necessario informarla circa le coordinate del database da utilizzare, per farlo bisogna apportare un modifica al file src/test/resources/jetty/env.xml con il vostro editor di testo preferito. Dovete inserire cambiare solo il nome del database (alla riga 6) e lo username (riga 8) da utilizzare per ottenere qualcosa di simile al frammento di xml seguente:

```
1  ...
<New id="jira-ds" class="org.mortbay.jetty.plus.naming.Resource">
3    <Arg>jdbc/Datasource</Arg>
    <Arg>
5      <New class="org.enhydra.jdbc.standard.StandardConnectionPoolDataSource">
        <Set name="Url">jdbc:mysql://localhost/clienti</Set>
7       <Set name="DriverName">com.mysql.jdbc.Driver</Set>
        <Set name="User">nicola</Set>
9      </New>
    </Arg>
11 </New>
...
```

■ Per avere maggiori informazioni sulle diverse configurazioni relative alle connessioni al database si rimanda al capitolo 4 ■

1.5 Avviare l'applicazione

Ora tutto è pronto per avviare l'applicazione, se avete lasciato la cartella principale del progetto tornateci e da lì lanciate il comando seguente (e lasciate aperta la console):

```
nicola@casper:~/projects/clienti# mvn jetty:run
```

Sullo schermo si susseguiranno diverse righe per informarvi che l'applicazione è stata inizializzata e quando, infine, apparirà la dicitura *Started Jetty Server* saprete che tutto è pronto.

Lasciando sempre aperta la console aprite un'istanza del vostro browser e collegatevi all'indirizzo localhost:8080/clienti per vedere la pagina di benvenuto della vostra applicazione.

Complimenti, avete appena fatto il primo passo nel mondo delle applicazioni Regola kit!

■ Per visualizzare l'applicazione state utilizzando un piccolo (ma molto completo) application server di nome Jetty. Per la messa in produzione però si consiglia di utilizzare dei container diversi (ad esempio Tomcat o JBoss). Per imparare a come creare i pacchetti per questi application server si rimanda al capitolo 6 nella pagina 29 ■

1.6 Struttura di un progetto Regola kit

La struttura della cartella di un progetto Regola kit si impronta alla struttra standard di un progetto web di Maven 2. Al primo livello troviamo:

pom.xml	il file di configurazione di Maven 2
src/	la cartella dei sorgenti
target/	contiene i file compilati ed i pacchetti per le consegne

La cartella target contiene quanto i pacchetti pronti per la consegna con la classi compilate ed i descrittori. Si tratta di una cartella il cui contenuto è ricreato ogni volta si lanci il comando:

```
1 nicola@casper:~/projects/clienti# mvn package
```

La cartella src contiene i sorgenti (html, js, css e java) dell'applicazione. Al suo interno potete trovare:

main/	contiene i sorgenti dell'appplicazione
test/	contiene i sorgenti dei test
site/	reportistica generata da Maven 2

La cartella main e la cartella test contengono entrambe i sorgenti java (nella sottocartella java) e le altre risorse (nella cartella risorse). Queste ultime sono i file di configurazione, i mappaggi orm e, in generale, tutto quello che non sono sorgenti Java ma devono finire comunque nel classpath. La differenza tra la cartella main e quella test e che il contenuto di quest'ultima non finisce mai nei pacchetti destinati alla produzione ma è usato esclusivamente per l'esecuzione dei test. Infine la cartella main contiene anche la sottocartella webapp dove si trova la webroot, ovvero le pagine web dell'applicazione ed il file web.xml.

■ Per una descrizione completa dei file e delle cartelle standard di un progetto Regola kit si rimanda al capitolo 3 nella pagina 19 ■

1.7 Persistenza su database

Regola kit abbraccia una metodologia di sviluppo incentrata sull'analisi del dominio del problema (Domain Driven Development) per cui il primo passo è quello di creare le classi di modello. Spesso queste classi sono persistite sul database per cui si potrebbe iniziare scrivendo la classe e poi creando la corrispondente tabella sul database. Oppure, al contrario come faremo tra poco, creando prima la tabella del database e facendoci poi creare in automatico la classe Java (nel prossimo paragrafo 1.9 nella pagina seguente). Colleghiamoci nuovamente al database clienti.

```
nicola@casper:~/projects/clienti# mysql -u root -p clienti
```

Creiamo una piccola tabella con la sola chiave primaria (id) ed un campo di descrizione (label).

```
mysql> create table prodotti (id int(11) not null auto_increment, label
varchar(80) not null, primary key (id) );
mysql> describe prodotti;
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| label  | varchar(80)  | NO   |     |         |                |
+--------+--------------+------+-----+---------+----------------+
2 rows in set (0.02 sec)
```

Inseriamo qualche dato nella tabella, ad esempio alcune descrizioni di esempio per verificare poi il funzionamento dell'applicazione.

```
mysql> insert into prodotti values (null, 'book');
Query OK, 1 row affected (0.05 sec)

mysql> insert into prodotti values (null, 'bottle');
Query OK, 1 row affected (0.00 sec)

mysql> insert into prodotti values (null, 'paper');
Query OK, 1 row affected (0.00 sec)

mysql> select * from prodotti;
+----+---------+
| id | label   |
+----+---------+
|  1 | book    |
|  2 | bottle  |
|  3 | paper   |
+----+---------+
3 rows in set (0.01 sec)
```

Adesso il database contiene una tabella con dei dati che possiamo usare per persistere le nostre classi di modello. Usciamo dal database e torniamo all'applicazione.

```
mysql> exit
```

1.8 (Ri)collegarsi al database

Al paragrafo 1.4 nella pagina 9 abbiamo configurato l'applicazione per utilizzare il nostro database in fase di esecuzione. Adesso dobbiamo fare in modo che anche in fase di sviluppo si possa accedere al database (ad esempio per usare i generatori di codice o lanciare la batteria di test). Il file da modificare è src/test/resources/designtime.properties e deve essere aggiornato in modo da contenere lo username, la password ed il nome del database. Il risultato finale deve risultare simile al seguente:

```
1 ...
  hibernate.dialect=org.hibernate.dialect.MySQLDialect
3 hibernate.connection.driver_class = com.mysql.jdbc.Driver
  hibernate.connection.url = jdbc:mysql://localhost/clienti
5 hibernate.connection.username = nicola
  hibernate.connection.password =
7 ...
```

1.9 Classi di modello

Siete ora in grado di scrivere la classe di modello che sarà persistita sulla tabella prodotti... oppure potete farvela generare automaticamente e poi modificare convenientemente le classi prodotte. Userete gli Hibernate Tools che sono già configurati all'interno delle applicazioni prodotte con Regola kit e trovano nell'unico file src/test/resources/hibernate.reveng.xml la configurazione di tutto il processo di generazione inversa, a partire cioè dal database. Specificate il nome della tabella da cui partire (*prodotti* alla riga 2), il nome della classe (*Prodotto*, al singolare e con la prima lettera maiuscola nella riga 4), il package da utilizzare (*com.acme.model* sempre alla riga 2).

```
1 ...
   <table-filter match-name="prodotti"  package="com.acme.model"
   exclude="false"/>
3
    <table name="prodotti" class="Prodotto" >
5      <primary-key property="id" />
    </table>
7 ...
```

Ora avviate la generazione: posizionantevi nella cartella principale del vostro progetto ed utilizzate il plugin di Maven 2 Hibernate3 che consente di generare le classi java (il goal hbm2java), i file di mappaggio di hibernate (hbm2hbmxml) e la configurazione generale di hibernate (hbm2cfgxml).

```
1 nicola@casper:~/projects/clienti# mvn hibernate3:hbm2java
  hibernate3:hbm2hbmxml hibernate3:hbm2cfgxml
```

Il primo file generato si trova nella posizione src/main/java/com/acme/-model/Prodotto.java e contiene la classe Java:

```
1  public class Prodotto  implements java.io.Serializable {
2
3      private Integer id;
4
5      public Prodotto() {
6      }
7
8      public Integer getId() {
9          return this.id;
10     }
11
12     public void setId(Integer id) {
13         this.id = id;
14     }
15 }
```

Poi è stato creato il file con i mappaggi di hibernate src/main/resources/com/acme/model/Prodotto.hbm.xml, molto semplice in questo caso:

```
1  <?xml version="1.0"?>
   <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
   3.0//EN"
3  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
   <!-- Generated 14-apr-2008 12.23.38 by Hibernate Tools 3.2.0.CR1 -->
5  <hibernate-mapping>
       <class name="com.acme.model.Prodotto" table="prodotti"
       catalog="clienti">
7          <id name="id" type="java.lang.Integer">
               <column name="id" />
9              <generator class="identity" />
           </id>
11     </class>
   </hibernate-mapping>
```

Ed infine è stato inserito un riferimento a quest'ultimo file di mappaggio dentro la configuraizone principale di Hibernate src/main/resources/hibernate.cfg.xml (alla riga 13).

```
   <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
   <hibernate-configuration>
6      <session-factory>
           <property
           name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
8          <property name="hibernate.connection.password"></property>
           <property
           name="hibernate.connection.url">jdbc:mysql://localhost/clienti</property>
10         <property name="hibernate.connection.username">nicola</property>
           <property
           name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
12
          <mapping resource="com/acme/model/Prodotti.hbm.xml" />
14     </session-factory>
   </hibernate-configuration>
```

Attenzione: il goal hibernate3:hbm2cfgxml cancella e riscrive ogni volta questo file ed inoltre vi aggiunge delle configurazioni che a runtime non sono usate (come username, password, url e driver class). Nell'impiego di tutti

i giorni di Regola kit il nostro team di sviluppo non utilizza il goal hibernate3:hbm2cfgxml e si occupa di aggiungere manualmente i mappaggi delle risorse. Naturalmente le configurazioni non usate non costituiscono problema, per cui alla fine la scelta di impiegare o meno hibernate3:hbm2cfgxml è lasciata alla vostra discrezione.

■ Esistono altri goal disponibili, ad esempio la generazione degli script sql per creare le tabella a partire dalla configurazione delle classi. Si rimanda, per approfondimenti, al capitolo 5 nella pagina 27. ■

1.10 Dal modello alla presentazione

Adesso che avete creato la classe di modello è il momento di realizzare il codice per leggere e scrivere oggetti (della classe Prodotto) sul database, le pagine web che elenchino questi oggetti così come la pagine di dettaglio per effettuare modifiche. Questo codice può essere scritto a mano oppure potete partire facendovi generare automaticamente della classi di default che utilizzerete come modello di partenza per le vostre modifiche.

Prima di lanciare il generatore di Regola kit bisogna assicurarsi che il progetto sia compilato e poi invocare il goals exec:java che avvia il generatore.

```
1 nicola@casper:~/projects/clienti# mvn compile
  nicola@casper:~/projects/clienti# mvn exec:java -Dexec.args="-c
  com.acme.model.Prodotto -m"
```

Noterete tra i parametri passati al comando il nome della classe attorno a cui costruire i vari livelli e l'opzione *m* che specifica di utilizzare un'ampia catena di generatori, in particolare:

dao	produce il custom dao
modelPattern	produce la classe necessaria a Model Pattern
properties	aggiunge le chiavi per la localizzazione
list-handler	genera il controller dietro la pagina di lista
list	genera la pagina di lista
form-handler	genera il controller dietro la pagina di dettaglio
form	genera la pagina di dettaglio

■ I generatori possono anche essere avviati individualmente. Per scoprire come e conoscere anche altri generatori forniti con Regola kit si rimanda al capitolo 7.4 nella pagina 36. ■

1.11 Usare Eclipse

Per lavorare sul progetto appena creato con Eclipse basta lanciare il comando seguente che provvede a creare tutti i file necessari a quell'ambiente:

```
nicola@casper:~/projects/clienti# mvn eclipse:eclipse
```

A questo punto basta importare il progetto nel workspace con la voce di menu File | Import : magari senza settare l'opzione di copia e quindi lavorando sul progetto nella sua cartella originale.

Capitolo 2

Installazione

2.1 Maven 2

TODO: Qui si spiega che Regola kit si fonda su Maven 2 per tutta la gestione del ciclo di build.

2.2 Librerie

TODO: Dove si elencano i moduli di cui si compone

2.3 Dipendenze

TODO: Dove si elencano le dipendenze, ricordando però che sono gestite da Maven 2

2.4 Eclipse IDE

Per lavorare su un progetto Regola kit con Eclipse basta lanciare il comando seguente che provvede a creare tutti i file necessari a quell'ambiente:

```
1 nicola@casper:~/projects/yourProject# mvn eclipse:eclipse
```

A questo punto basta importare il progetto nel workspace con la voce di menu File | Import : magari senza settare l'opzione di copia e quindi lavorando sul progetto nella sua cartella originale.

Per installare i plugin necessari al funzionamento dell'assistente alla scrittura del codice (vedi il paragrafo a pagina 7.4 nella pagina 36) basta sem-

plicemente ricopiare il file ... dentro la cartella eclipse/plugin e riavviare Eclipse.

Capitolo 3

Struttura di un progetto

Questo capitolo funziona un po' come una cartina stradale e rimanda ad altri capitoli per l'approfondimento.

/	pom.xml	il file principale di Maven 2
src/main/resources/	runtime.properties log4j.xml hibernate.cfg.xml ApplicationResources.properties applicationContext-*.xml <stesse cartelle dei packages>	proprietà di runtime la verbosità dei log la configurazione principale di Hibernate i testi tradotti nelle varie lingue le configurazioni di Spring i mappaggi di Hibernate
src/test/resources/	designtime.properties log4j.xml hibernate.reveng.xml applicationContext-*-test.xml jetty/env.xml	proprietà a design time la verbosità dei log Hibernate tools le configurazioni di Spring datasource per Jetty

3.1 Lo standard Maven 2

Fra i tanti vantaggi offerti di cui si può beneficiare utilizzando Maven 2 vi sono quelli derivanti dall'adesione ad uno standard (che i teorici dei giochi definiscono equilibrio di Nash); infatti disponendo i diversi file di un progetto in modo convenuto consente agli sviluppatori (e i sistemisiti) di orientarsi da subito su un progetto estraneo, a sistemi di supporto alla scrittura di trovare senza configurazione le risorse di cui abbisognano ed in generale consentono a soggetti diversi e lontani (nello spazio e nel tempo) di cooperare sullo stesso

progetto senza conflitti.

La struttura esatta di progetto web è reperibile su (mav), però in estrema sintesi, la gerarchie delle cartelle prevede al primo livello la cartella dei sorgenti (src) e quella con gli artifatti prodotti (target), ad esempio le classi compilate ed i war file. La cartella src è primariamente suddivisa in due sottocartelle speculari, una contiene sorgenti di produzione (main) ed una quelli di test. Queste sottocartelle (di main e test) contengono i sorgenti java (cartella java) e le risorse (dentro resources), prevalentemente i file di configurazione. Da rilevare che la cartella main contiene anche la sottocartella webapp dove si trova la webroot, ovvero le pagine web dell'applicazione ed il file web.xml.

3.2 L'iniezione delle dipendenze

TODO: La distinzione tra i tre livelli di bean

3.3 La localizzazione

La localizzazione consiste nelle tecniche per tradurre i testi, l'aspetto e le form di input di un'applicazione nelle diverse lingue. Regola kit utilizza i Resource Bundle di Java per risolvere questo aspetto come indicato in (i18). Nella cartella src/main/resources sono presenti diversi file di proprietà il cui nome presenta la radice comune ApplicationResources, ad esempio:

- ApplicationResources en.properties
- ApplicationResources it.properties

Queste file replicano la stessa chiave traducendola nella varie lingue. Ad esempio dentro ApplicationResources en possiamo trovare la chiave errors.required a cui è associato un testo inglese:

```
1 errors.required=the field is required.
```

Nel file ApplicationResources it sarà possibile fornire una traduzione per la chiave errors.required semplicemente ridefinendola.

```
1 errors.required=campo necessario.
```

Affinché questo meccanismo di localizzazione funzioni Regola kit provvede automaticamente a selezionare il giusto file in base alle impostazioni del browser che utilizza l'applicazione. Inoltre è necessario rammentarsi di non includere mai direttamente dei testi all'interno delle pagine web ma di

richiamare le chiavi definite dentro i file ApplicationResources. Ad esempio nelle pagine jsp bisogna inserire qualcosa di simile a:

```
1 <fmt:message key="errors.required"/>
```

Per le pagine jsf è invece possibile utilizzare il managed bean mgs.

```
1 #{msg[errors.required]}
```

3.4 Le connessioni al database

Le connessioni al database sono di due tipi a seconda dell'ambiente in cui sta girando l'applicazione. Per l'ambiente di run-time è necessario indicare il nome JNDI del datsource fornito dall'application server mentre a design-time bisogna proprio impostare tutte le caratteristiche di una connessione. In entrami i casi la configurazione riguarda l'impostazione di proprietà di configurazione dentro i file design-time.properties e run-time.properties.
Per maggiori dettagli ed esempi di configurazione per i diversi application server si rimanda a 4 nella pagina 23.

3.5 Verbosità dei log

TODO: log4j e come interagisce con JBoss

3.6 La sezione dei test

TODO: come configurare i test e quali file utilizzano.

3.7 Applicazione Servlet

TODO: Davvero in breve la struttura

Capitolo 4

Database

4.1 Configurazione di run-time

Le applicazioni JEE generalmente lasciano la gestione delle connessioni database al container. In questo modo è possibile per i sistemisti modificare, ad esempio, la url di connessione oppure il numero di connessioni in pool senza modificare l'applicazione né dovere effettuare una riconsegna (redeploy). Ogni container configura in modo diverso però ogni datasource è caratterizzato necessariamenta da un nome JDNI, ad esempio java:comp/env/jdbc/miodatabase. Questo nome è utilizzato dall'applicazione per recuperare la connessione e, in definitiva, connettersi al database. Nelle applicazioni Regola kit il nome JNDI del datasource è specificato nella proprietà di runtime jee.datasrouce (nel file src/main/resource/runtime.properties).

```
1 jee.datasource=java:comp/env/jdbc/Datasource
```

Con questo la configurazione di runtime della vostra applicazione è terminata, non ci sono altre variabili da impostare. Un problema tipico di configurazione riportato di diversi utenti è l'impossibilità di connettersi al datasource per avere utilizzato un indirizzo JNDI sbagliato. La causa del problema è che il nome con cui l'application server espone la connessione non è *esattamente* quello specificato nella configurazione: perché ad esempio viene preposta la stringa 'java:' o a volte 'java:comp/env/'.

Per facilitare la soluzione di questo tipo di problema concludiamo il paragrafo riportando qualche configurazione di datasource per gli application server più diffusi. Ad esempio JBoss richiede di ricopiare nella cartella di deploy del server utilizzato un file con il nome del tipo *-ds.xml (ad esempio miadatasource-ds.xml) con un contenuto simile al seguente:

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

```
<datasources>
 <local-tx-datasource>
    <jndi-name>jdbc/services</jndi-name>

    <connection-url>jdbc:oracle:thin:@133.222.0.1:1522:SID</connection-url>
    <user-name>username</user-name>
    <password>*****</password>

    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <exception-sorter-class-name>
       org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
    </exception-sorter-class-name>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>20</max-pool-size>

 </local-tx-datasource>
</datasources>
```

In questo caso nella configurazione dell'applicazione dovete indicare il nome indicato però preceduto da 'java:' come indicato di seguito.

```
jee.datasource=java:jdbc/services
```

Per quanto riguarda Jetty il file di configurazione env.xml contiene l'indicazione del datasource.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
"http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">

  <New id="jira-ds" class="org.mortbay.jetty.plus.naming.Resource">
     <Arg>jdbc/Datasource</Arg>
       <Arg>
         <New
         class="org.enhydra.jdbc.standard.StandardConnectionPoolDataSource">
           <Set name="Url">jdbc:mysql://localhost/clienti</Set>
           <Set name="DriverName">com.mysql.jdbc.Driver</Set>
           <Set name="User">nicola</Set>
         </New>
       </Arg>
  </New>

</Configure>
```

Jetty aggiunge al nome configurato la stringa 'java:comp/env/' per cui l'indirizzo JNDI da utilizzare è il seguente:

```
jee.datasource=java:comp/env/jdbc/Datasource
```

Lo stesso indirizzo JNDI può essere utilizzato anche per un datasource fornito da Tomcat, utilizzando una configurazione simile alla seguente per il file context.xml:

```
<Context >
  ...
  <Resource name="jdbc/Datasource" auth="Container"
  type="javax.sql.DataSource"
             maxActive="100" maxIdle="30" maxWait="10000"
```

```
5               username="nicola" password=""
                driverClassName="com.mysql.jdbc.Driver"
                url="jdbc:mysql://localhost/clienti"/>
7 </Context>
```

TODO: Mi chiedo come mai il dialetto sia specificato come proprietà di designtime ma non come proprietà di runtime.

4.2 Configurazione di design-time

Tra le caratteristiche invidiabili del framework Spring vi è la possibilità di testare l'applicazione fuori dal container con notevoli risparmio di tempo e relativo aumento di produttività. I servizi necessari ai test sono offerti direttamente da Spring che si sostituisce al container, ad esempio, nella gestione delle transazioni e nei datasource. In quest'ultimo caso è necessario specificare tutte le proprietà di una connessione (come il driver, la url, la username e la password) come proprietà di designtime (nel file src/test/resources/designtime.properties).

```
1 ...
  hibernate.dialect=org.hibernate.dialect.MySQLDialect
3 hibernate.connection.driver_class = com.mysql.jdbc.Driver
  hibernate.connection.url = jdbc:mysql://localhost/clienti
5 hibernate.connection.username = nicola
  hibernate.connection.password =
7 ...
```

Da notare che è necessario specificare anche la variabile hibernate.dialect con il tipo di *dialetto* di database utilizzato. I tipi disponibili sono molti ed elencati nella documentazione di Hibernate, alcuni esempi comuni sono MySQL5Dialect, OracleDialect, PostgreSQLDialect, HSQLDialect e SQLServerDialect.

Capitolo 5

Persistenza

5.1 Hibernate

TODO: Qui si dice che l'orm predefinito da Regola kit è Hibernate, perché è estremamente flessibile e su database preesistenti consente di gestire anche i casi più anomali. Perché non usiamo ancora JPA (immaturo).

5.2 Configurazione

TODO: I file hibernate.cfg.xml ed i mappaggi. Alcuni esempi di base per questi file.

5.3 Generatori automatici

TODO: Esempi di configurazione di hibernate.reveng.xml ed esempi (tutti) dei vari goals disponibili a riga di comando.

5.4 Altri ORM

TODO: Si parla del supporto sperimentale per gli altri orm.

Capitolo 6

Messa in produzione

6.1 Produrre i pacchetti war ed ear

TODO: Qui si spiega come produrre i pacchetti e cosa contengano al loro interno

6.2 Application Server

TODO: Perché non usare Jetty ed esempi di configurazione per Tomcat e JBoss. Si rimanda al capitolo dei database per la configurazione dei datasource.

Per utilizzare CXF all'intenro di JBOSS è necessario impostare il parametro di avvio della virtual machine javax.xml.soap.MessageFactory al valore com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl.

6.3 Integrazione continua

TODO: Perché serve e come configurare i file standard dei progetti di Regola kit

Capitolo 7

Sviluppo

Questo capitolo apre la parte dedicata alle funzionalità offerte da Regola kit relativamente allo sviluppo. Lasciandosi alle spalle le configurazioni.

7.1 Domain Driven Development

TODO: Molto in breve si spiega come ci si incentri sul modello.

7.2 Livelli

TODO: Si presentano i vari livelli e si rimanda a capitoli successivi per i dettagli

7.3 Model Pattern

Regola kit nasce per agevolare la realizzazione di applicazioni web destinate a gestire moli piuttosto ampie di dati generalmente persistite su un database relazionale. Un problema ricorrente in questi contesti è quello che si potrebbe sinteticamente indicare come come la questione dei sottoinsiemi, ovvero fornire una risposta generale alle domande: come estrarre in modo conveniente sottoinsiemi di una certa entità dal database? E come rappresentare in modo sintetico questi sottoinsiemi all'interno di un form html oppure in un'applicazione desktop?
La soluzione proposta da Regola kit riteniamo che possa essere ritenuta sufficientemente assodata e generica da ambire al rango di design pattern (magari un minore); se così fosse pensiamo che la descrizione del problema unitamente

alla soluzione individuata possa essere identificata col nome di Model Pattern (o in alternativa Selection).

Nei prossimi paragrafi cercheremo di formalizzare questo nuovo pattern descrivendone intenti e forze in gioco. Si avvisa fin d'ora che materremo il discorso a livello teorico presentando solo esempi astratti utili unicamente per comprendere la portata della soluzione proposta. Rimandiamo al paragrafo 8.4 nella pagina 37 per illustrare l'implementazione di riferimento di Model Pattern (quella contenuta in Regola kit) e spiegarne l'utilizzo all'interno delle vostre applicazioni.

7.3.1 Intento

Model Pattern vuole fornire un design appropriato per individuare un sottoinsieme di oggetti del modello di dominio e rappresentare questo sottoinsieme in modo conveniente all'interno del livello di presentazione.

7.3.2 Forze

In una categoria piuttosto ampia di applicazioni (che comprende la quasi totalità di applicazioni web) gli oggetti del modello salvano il loro stato in modo persistente su database, magari attraverso dei framework ORM (ad esempio Hibernate, JPA, ecc.). Non di rado il numero di istanze persistenti, per tipologia di classe, è veramente elevato cosa che complica in primis l'estrazione (1) (ovvero il filtraggio) delle istanze. Sempre dalla mole dei dati deriva la necessità di ordinare (2) e paginare (3) gli oggetti estratti: ovvero raggrupparli in sotto gruppi contenenti al più un certo numero di elementi. Un altra richiesta tipica delle applicazioni di cui stiamo parlando è quella di rappresentare gli oggetti del modello estratti in modo parziale, ovvero mostrare solo un sotto gruppo di proprietà, in modo da fornire, in un solo colpo d'occhio, tutti gli elementi necessari per prendere delle decisioni sui dati eliminando le proprietà superflue. Questo sezionamento delle classi del modello si chiama proiezione (4).
Del sottoinsieme degli elementi del modello individuato e estratto dal database (reidratato) è inoltre necessario fornire delle rappresentazioni sintetiche da utilizzare in diversi punti dell'applicazione. Si prenda per esempio la classe Studente ed il form html che consente di effetture le operazione di editing sulle sue proprietà quali nome, cognome ed altre. Si consideri poi che tra le proprietà della classe Studente vi sia anche la collezione di classi del tipo Esami che contenga gli esami sostenuti fino alla data corrente. Bene, il problema è spesso quello di mostrare nello stesso form html contenete le proprietà di

Studente anche la collezione di Esami: in questo caso la rappresentazione corretta potrebbe variare in base alle finalità applicative. Ad esempio le segreterie di facoltà potrebbero essere interessati all'elenco completo di tutti gli esami (nessuna sintesi), i docenti potrebbero volere esclusivamente la media delle valutazione degli esami (applicare una metrica), l'ufficio delle imposte potrebbe essere interessato solo al fatto di aver sostenuto almeno un esame (e quindi un intero che esprime il numero di esami superati). Tutte queste rappresentazioni (elenco, metrica ed intero) sono una descrizione del sottoinsieme di oggetti del modello Esami.
Inoltre si consideri il caso in cui la collezione di Esami fosse vuota o nulla, in questo caso diverse rappresentazioni tradurrebbero il valore null nel modo più appropriato rispetto al contesto (ad esempio con una stringa che indica l'assenza di esami, o con una segnalazione di errore o quant'altro). In ogni caso per ogni sottoinsieme di oggetti del modello emerge prepotente la necessità di provvedere ad una o più rappresentazione sintetica (5).
Quindi ricapitolando le forze che caratterizzano il problema:

1. estrazione di dati efficiente da una mole ampia o ampissima
2. ordinare le istanze di modello estratte
3. paginare le istanze di modello estratte
4. effettuare delle proiezioni sulle istanze di modello estratte
5. fornire delle rappresentazioni sintetiche della totalità di istanze estratte (cioè del sottoinsieme di tutta la popolazione di istanze estratto)

7.3.3 Esempio

Per amore di concretezza riportiamo un esempio che possa aiutare a comprendere meglio il problema descritto nel paragrafo precedente e la soluzione proposta da Model Pattern. Immaginiamo di avere una collezione di oggetti persista su database, tutti appartenenti alla medesima classe Prodotto, che presentano solo due proprietà: id, un intero che tiene la chiave primaria, e la descrizione del prodotto stesso.

```
1 public class Prodotto {

3     private Integer id;
      private String descrizione;
5
      //getter and setter per tutti i campi
7     ...
  }
```

Il problema è quello di descrivere in modo sintetico sottoinsiemi di oggetti della classe Prodotto. Procederemo per gradi iniziando con il sottoinsieme diciamo Σ definito in modo sintetico, dalla descrizione seguente: il sottoinsieme

di tutti gli oggetti la cui descrizione inizia con la parola 'manuale'. L'equivalente informatico di questa definizione è costiuito da un'instanza della classe seguente (che svolge il ruolo di Model Pattern):

```
public class ProdottoPattern {

    private String inizioDescrizione;

    //getter and setter per tutti i campi
    ...
}
```

In particolare il sottoinsieme Σ può essere definito da un oggetto della classe ProdottoPattern che ha come proprietà inizioDescrizione la stringa 'manuale'. Si può notare che ProdottoPattern non contiene né il sottoinsieme Σ né il codice per estrarlo da database; si limita solo a fornirne una descrizione sintetica del sottoinsieme. Arricchiamo questo primo esempio includendo anche l'ordine degli oggetti contenuti nel sottoinsieme Σ, ad esempio richiedendo che siano ordinati in base alla proprietà id. In questo caso la classe Model Pattern potrebbe modificarsi così:

```
public class ProdottoPattern {

    private String inizioDescrizione = ''manuale'';
    private String[] ordine = { ''id'' };

    //getter and setter per tutti i campi
    ...
}
```

In questo modo sarebbe in grado di descrivere il sottoinsieme ordinato Σ anche se in modo un po' rozzo infatti si trascura la possibilità di ordinare in modo ascendente o discendente. Comunque l'esempio dovrebbe rendere l'idea di come le classi Model Pattern si limitino a descrivere i sottoinsiemi senza contenere codice per l'estrazione dello stesso. Questo servizio è infatti fornito dalle classi a corredo come quelle fornite da Regola kit che consentono, dato un oggetto del tipo Model Pattern, di ottenere il sottoinsieme voluto. La sfida consiste nella predisposizione di un architettura che consenta di utilizzare dei Model Pattern semplici come quelli mostrati in questi esempi ma in grado di funzionare correttamente (ad esempio indicando che la proprietà inizioDescrizione si riferisce alla proprietà descrizione di Prodotto) su diversi orm. Si rimanda al paragrafo successivo per una descrizione esatta dell'architettura. Fino a questo momento abbiamo trattato solo la capacità di Model Pattern di esprimere una selezione, adesso ci occuperemo della funzione di rappresentazione del sottoinsieme selezionato. Immaginiamo di dover mostrare in una pagina web il sottoinsieme Σ in modo sintetico (ovvero senza elencare in una colonna tutti i suoi elementi). Modifichiamo Model Pattern per aggiungere la proprietà sintesi.

```
public class ProdottoPattern  {

    public getSintesi()
    {
       if (inizioDescrizione == null)
       {
          return ``Prodotti con descrizione iniziante con: '' +
          inizioDescrizione;
       }
       else
       {
          return "Tutti di prodotti.";
       }
    }

}
```

Come si vede la proprietà sintesi contiene la logica per fornire una semplice rappresentazione del sottoinsieme Σ anche nel caso in cui inizioDescrizione sia nullo.

7.3.4 Architettura

Model Pattern propone di avvilire le forze del problema utilizzando una classe (Pattern) per ogni tipo nel modello che si occupi di due distinti aspetti:

1. individuare il sottoinsieme selezionato
2. fornire (zero o più) rappresentazione del sottoinsieme selezionato

Per quanto riguarda il primo punto bisogna precisare fin da subito che la classe Pattern non contiene in nessun modo il codice necessario all'estrazione della selezione o la selezione stessa. Invece si occupa di contenere le informazioni necessarie a descrivere in modo esatto il sottoinsieme, ovvero descriverlo in modo esatto rispetto ai seguenti punti:

1. quali istanze sono comprese nel sottoinsieme e quali esluse
2. con quali ordine le istanze entrano nel sottoinsieme
3. in che modo il sottoinsieme è paginato
4. quali proiezioni effettuare su ogni singola istanza del sottoinsieme

La classe Pattern si occupa anche di contenere zero o più rappresentazioni del sottoinsieme che descrive da utilizzare in altrettanti punti dell'applicazione (tipicamente un qualche livello di presentazione, o la produzione di report, ecc.). A tal proposito predispone il metodo

```
    void init(Collection<Model> subset)
```

questo metodo prende in ingresso un sottoinsieme e provvede ad inizializzare le rappresentazioni di subset secondo la logica contenuta in pattern;

[TODO: nda Lorenzo, in futuro potrebbe essere utile utilizzare questo stesso metodo per inizializzare anche la parte di descrizione 1) in base ad un subset, ovvero produrre il filtro che estrarrebbe eventualmente subset?]

7.4 Generatori

TODO: Qui si elencano i generatori disponibili e cosa scrivano. Se il paragrafo diventasse troppo lungo allora lo mettiamo su un capitolo a parte.

Capitolo 8

Dao

8.1 Scopo

TODO: A cosa serve DAO?

8.2 GenericDao

TODO: Se ne descrivono interfacce e si presentano esempi d'uso (dei test d'unità) che si snodano tra i vari paragrafi.

8.3 Creare un custom dao

TODO: l'interfaccia, l'implementazione ed infine la configurazione di Spring. Si ricorda che esiste un generatore per questo.

8.4 Ricerche con Model Pattern

Nel paragrafo 7.3 nella pagina 31 è stato presentato in modo formale un nuovo design pattern chiamato Model Pattern; è stato spiegato come intenda risolvere il probema dell'estrazione di un sottoinsieme di oggetti e di come intenda fornire una rappresentazione sintetica di questo oggetto. La soluzione proposta, si è visto, prevede l'utilizzo di una classe che svolga il ruolo di ModelPattern, ovvero sia contemporaneamente in grado di individuare quali elementi estrarre e di rappresentarli senza però contenere al suo interno né il sottoinsieme né la logica per l'estrazione. La discussione è rimasta però a livello astratto rimandando la descrizione di come Model Pattern possa

essere utilizzato concretamente all'interno di un'applicazione Regola kit; nei prossimi paragrafi vedremo quindi l'implementazione di riferimento di Model Pattern contenuta in Regola kit (precisamente nei moduli regola-core, regola-dao e sottomoduli) per scoprire come progettare un classe ModelPattern con diversi criteri di filtraggio e ordinamento e la utilizzeremo lungo i diversi livelli applicativi, dal DAO alla presentazione. Per maggiore concretezza immagineremo di doverci occupare, diciamo, della classe Prodotto riportata di seguito.

```
class Prodotto {
    Integer id;
    String  descrizione;

    //seguono getter/setter per ogni campo
    ...
}
```

8.4.1 ModelPattern

Regola kit fornisce una classe di base org.regola.model.ModelPattern da cui derivare il nostro ModelPattern; questa classe presenta alcune facilitazioni per specificare gli ordinamenti e la paginazione ed è accettata quasi in ogni livello applicativo, dai GenericDao fino a controllori web che si occupano di disegnare la tabella con il sottoinsieme. Tenendo a mente la classe Prodotto una prima versione del nostro ModelPattern potrebbe essere la seguente:

```
class ProdottoPattern extends org.regola.model.ModelPattern
    implements Serializable {

    Integer  chiave;

    @Equals("id")
    public Integer getChiave() {
        return chiave;
    }

    ...
}
```

Soffermiamoci un attimo su alcuni elementi che trasformano la nostra classe in un ModelPattern utilizzabile con Regola kit:

nome della classe una convenzione di Regola kit prevede che ogni Model-Patter dovrebbe chiamarsi col nome della classe di modello a cui si riferisce (nel nostro caso Prodotto) seguito dal suffisso Pattern, per cui ProdottoPattern

derivazione la nostra classe deve derivare dalla classe org.regola.model.ModelPattern fornita con Regola kit

serilizzazione la nostra classe deve essere progettata per la serializzazione e quindi necessariamente implementare l'interfaccia di marker java.io.Serializable

La classe ProdottoPattern così realizzata è predisposta per individuare particolari sottoinsiemi di instanze Prodotto che presentano particolari valori della proprietà id, in particolare tutte le istanze di Prodotto che hanno id uguale alla proprietà chiave di ProdottoPattern. Il legame tra chiave ed id è realizzato mediante l'annotazione @Equals (alla riga 6) che, posta sul getter della proprietà chiave di ProdottoPattern, unisce quest'ultima alla proprietà dell'oggetto di modello specificata dentro l'annotazione stessa (in questo caso id).
Bisogna chiarire da subito che non è la classe ModelPattern ad individuare un sottoinsieme ma ogni sua istanza per cui, tornando al nostro esempio, bisogna istanziare un oggetto del tipo ProdottoPattern.

```
ProdottoPattern pattern = new ProdottoPattern();
2
pattern.setChiave(234532); //ora pattern rappresenta un sottoinsieme
```

Dopo l'assegnazione pattern (l'oggetto) e non ProdottoPattern (la classe) è in grado di individuare un sottoinsieme di oggetti Prodotto aventi la proprietà id uguale a 234532 (essendo id una chiave primaria il sottoinsieme conterrà al più un elemento). Da notare come l'oggetto pattern non contenga in sé il sottoinsieme ma solo la descrizione sintetica di questo; per ottenere il sottoinsieme bisogna rivolgersi alla classe ProdottoDao invocando il metodo find.

```
1 ProdottoPattern pattern = new ProdottoPattern();
3 pattern.setChiave(234532);
5 List<Prodotto> prodotti = prodottoDao.find(pattern);
```

Ora che abbiamo visto concretamente come creare ed utilizzare una classe ModelPattern possiamo scendere nel dettaglio ricordando che Model Pattern deve individuare con esattezza il sottoinsieme, in particolare deve occuparsi di ordinamento, paginazione e proiezione. Nei prossimi paragrafi quindi scopriremo come specificare:

1. la selezione da effettuare
2. i criteri di ordinarmento
3. la paginazione da effettuare
4. le proprietà da comprendere nella proiezione
5. fornire rappresentazioni sintetiche del sottoinsieme indicato

8.4.2 La selezione

La selezione avviene impostando alcuni criteri di filtraggio sul ModelPattern. Ogni criterio è caratterizzato da:

1. la proprietà del modello a cui si applica
2. il tipo di criterio (ad esempio ugualianza, appartenenza, ecc.)
3. il valore da utilizzare per il confronto

Riprendendo l'esempio del paragrafo precedente troviamo un unico criterio di filtraggio, di tipo ugualianza, che confronta la proprietà di modello id con il valore della proprietà chiave di ProdottoPattern.

```
class ProdottoPattern extends org.regola.model.ModelPattern
    implements Serializable {

    Integer  chiave;

    @Equals("id")
    public Integer getChiave() {
        return chiave;
    }

    ...
}
```

La proprietà di modello può trovarsi direttamente sulla classe radice, ovvero quella a cui riferisce ModelPattern (nell'esempio Prodotto) oppure su classi a questa collegate tramite associazioni del tipo uno-a-uno, molti-a-uno od uno-a-molti. In generale si navigano le associazioni utilizzando come separatore il punto tranne che per relazioni uno-a-molti (collezioni) dove si utilizza il simbolo [] postfisso. Nel dettaglio:

radice	il nome della proprietà	id nome
uno-a-uno	il punto	indirizzo.via categoria.descrizione
molti-a-uno	il punto	cliente.nome fattura.progressivo
uno-a-molti	[] postfisso ed il punto	elmenti[].nome elementi[].dettagli[].progressivo

Il tipo del criterio è impostato scegliendo l'annotazione tra quelle presenti in Regola kit. Attualmente è possibile scegliere tra le annotazioni seguenti:

Equals	ugualianza
NotEquals	disugualianza
Like	il like
GreatherThan	maggiore
LessThan	minore
In	appartenenza

Il valore di ogni criterio una proprietà di ModelPattern individuata annotandone il getter. Il tipo di questa proprietà deve corrispondere a quello del modello tranne per il criterio In per cui è necessario specificare un'array. L'esempio seguente fornisce una rappresentazione piuttosto completa di quando esposto:

```
public class CustomerPattern extends ModelPattern
   implements Serializable {

  private Integer id;

  @Equals("id")
  public .Integer getId()
  {
    return id;
  }

  private String firstName;

  @Like(value = "firstName", caseSensitive = true)
  public String getFirstName()
  {
    return firstName;
  }

  private String[] lastNames;

  @In("lastName")
  public String[] getLastNames() {
    return lastNames;
  }

  private Integer invoiceId;

  @Equals("invoices[].id")
  public Integer getInvoiceId() {
    return invoiceId;
  }

}
```

8.4.3 Ordinamento

Sugli oggetti selezionati è possibile impostare criteri di ordinamento in base alle proprietà della radice del modello o sugli oggetti ad essa associati. L'implementazione di Model Pattern presente in Regola kit utilizza per

specificare gli ordinamenti (così come le proiezioni) la classe ModelProperty, una specie di descrittore di una generica proprietà. Si instazia così:

```
ModelProperty mp = new ModelProperty("id","customer.column.",Order.asc);
```

ModelProperty contiene il nome della proprietà (in base a cui ordinare), un prefisso da utilizzare per individuare in modo univoco la proprietà all'interno di tutta l'applicazione ed, infine, la direzione dell'ordinamento (acendente o discendente). Nell'esempio si esprime un ordinamento ascentente sulla proprietà id, individuata nell'applicazione come customer.column.id. Per speficiare l'ordinamento basta popolare la lista sortedProperties di ModelPattern con tante istanze di ModelProperty in base all'ordinamento da realizzare. Ad esempio:

```
1 ModelProperty id = new ModelProperty("id","customer.column.",Order.asc);
  ModelProperty street = new
  ModelProperty("address.street","customer.column.",Order.desc);
3
  modelPattern.getSortedProperties.clear();
5 modelPattern.getSortedProperties.add(id);
  modelPattern.getSortedProperties.add(street);
```

In questo caso si impone un ordinamento crescente per id e descrescente per la proprietà strret dell'oggetto associato address.

8.4.4 Paginazione

Sugli oggetti così selezionati ed ordinati è possibile effettuare un'olteriore selezione dividendolo in blocchi contigui dette pagine contenenti al più n elementi. ModelPattern consente di impostare la dimensione delle pagine così come impostare la pagina da selezionare. Ad esempio:

```
modelPattern.setPageSize(20);
2 modelPattern.setCurrentPage(0);
```

Qui si suddivide l'insieme ordinato in pagina con al più di 20 oggetti, il primo blocco comprende gli oggetti da $[0, 19]$, il secondo da $[20, 39]$ e così via. Inoltre si seleziona la prima pagina (la numerazione delle pagine parte da 0), ovvero gli oggetti da $[0, 19]$.

8.4.5 Proiezione

Fino a qui abbiamo visto come effettuare la selezione, ordinarla e limitarla ad un certo numero di elementi. É inoltre possibile limitare non solo il numero ma anche le proprietà del singolo oggetto del sottoinisieme. Ad esempio può essere conveniente limitarsi a considerare solo la proprietà id e name piuttosto che il complesso di tutte le proprietà della radice del modello e/o degli oggetti associati. Questo genere di limitazione si chiama proiezione. Il meccanismo

per specificare le proiezione dentro Regola kit si basa sempre sugli oggetti ModelProperty, basta popolare la lista visibleProperties di ModelPattern con le proprietà che si intende proiettare.

```
ModelProperty id = new ModelProperty("id","customer.column.",Order.asc);
ModelProperty street = new
ModelProperty("address.street","customer.column.",Order.desc);

modelPattern.getVisibleProperties.clear();
modelPattern.getVisibleProperties.add(id);
modelPattern.getVisibleProperties.add(street);
```

Nell'esempio si effettua una proiezione comprendente esclusivamente le proprietà id e strett dell'oggetto associato street.

8.4.6 Rappresentazione

TODO:

Capitolo 9

Servizio

9.1 Scopo

TODO: A cosa serve il livello di Servizio? (spr)

9.2 GenericManager

TODO: Come crearlo. Si ricorda che esiste un generatore per questo.

9.3 Transazioni

TODO: come sono demarcate e come aggiungere politiche diverse per aprire e chiudere transazioni

9.4 Politiche di detach

Gli EJB, fin dal loro esordio nel 1998, hanno nascosto le classi di modello al livello di presentazione ed utilizzato in loro vece delle classi apposite, i Data Transfer Object. Questi DTO avevano il compito di raccogliere una visione appiattita del modello che però contenesse tutte e sole le informazioni necessarie al livello di presentazione. Erano oggetti di passaggio, tutti dati e nessuna logica che nascondevano la complessità del modello agli strati superiori, realizzando di fatto un disaccoppiamento forte tra questi livelli. Nonostante questo pregio i DTO hanno contribuito molto alla fama di pesantezza degli EJB in quanto, anche in progetti piccoli, risultava evidente che il lavoro per creare, popolare e sincronizzare i DTO con la classi di modello era

di gran lunga superiore ai benefici. La domanda sorta spontanea a molti era: perché non passare ai livelli superiori direttamente le classi di modello? Molti progetti hanno dimostrato che la strada è praticabile con successo e formalizzato i vari modi per farlo. In questo paragrafo presenteremo due pattern per utilizzare direttamente le classi di modello nel livello di presentazione conosciuti rispettivamente come pojo façade e modello esposto.

9.4.1 Pojo façade

Questo pattern per utilizzare le classi di modello mantiene l'architettura complessiva degli EJB interponendo tra la presentazione ed il modello un apposito livello (façade) il cui compito è ancora quello di fornire servizi per la presentazione. La differenza principale con gli EJB però risiede in due aspetti:

1. Il livello façade è costituito da semplici pojo
2. Invece che DTO si passano oggetti di modello, opportunamente trattati

Del primo punto bisogna sottolineare che il livello façade, come negli EJB, è ancora responsabile di aprire e chiudere le transazioni così come della sicurezza (autenticazione ed autorizzazione). La differenza è che non è necessario un container JEE per ottenere questi servizi ma sono realizzati tramite un container invertito (come Spring) e la programmazione orientata agli aspetti (AOP). Con il vantaggio di poter effettuare tutti i test fuori dal container. Il secondo punto invece richiede alcuni approfondimenti che discuterò di seguito.

incapsulamento del modello

Il modello, secondo la definizione più accettata di Model Driven Development (MDD), è una realtà attiva in grado di utilizzare risorse esterne, ad esempio può presentare dei metodi per salvarsi su database. Se il livello di presentazione chiamasse direttamente questi metodi verrebbe meno la funzione centralizzatrice dei pojo façade ed il sistema solleverebbe diverse eccezioni legate all'assenza di transazioni attive sul livello di presentazione. Per ovviare a questo problema esistono alcune tecniche:

convenzione si stabilisce che nessun sviluppatore chiami mai questi metodi e si limiti ad ignorarli. Per gruppi numerosi potrebbe essere necessario rafforzare la convenzione magari marcando i metodi da ignorare con delle annotazioni ed utilizzare un compilatore ad aspetti che riporti come errore ogni chiamata a questi metodi effettuata nel livello di presentazione.

visibilità dei metodi ovvero marcare i metodi come private o protected. Spesso però il modello è sparso in vari package Java per cui questa possibilità non è praticabile.

utilizzare interfacce il livello di presentazione è scritto non in termini delle classi del modello ma in termini di sotto interfacce che ne espongano solo alcuni metodi nascondendone altri. Può essere realizzato in due modi:

realizzare le interfacce direttamente sul modello è una strada tecnicamente complessa perché richiede di scrivere metodi covarianti, di affrontare il problema delle collezioni immutabili ed inoltre non consente comunque di esporre quei metodi che richiedevano computazioni realizzabili solo nel livello di modello (perché ad esempio richiedono l'accesso al database)

creare degli adapter hanno lo svantaggio principale di essere terribilmente complicati e finiscono spesso per somigliare ai DTO (in termini dei soli svantaggi).

oggetti staccati (detached)

Gli elementi del modello possono effettuare implicitamente delle richieste al motore di persistenza in modo invisibile per i client. Il caso più esemplare è quello delle collezioni che gli ORM gestiscono a volte in modalità di caricamento differito (lazy o late loading). Ovvero gli elementi delle collezione sono recuperati dal database solo al momento del primo accesso ad un elemento, in modo trasparente. Il problema è che nel livello di presentazione sessioni ORM e transazioni non sono disponibili con conseguente errore a run time. Esistono in generale due tipi di soluzioni:

Eliminare i caricamenti differiti ovvero effettuare delle configurazioni per gli ORM (dette mappaggi) che carichino subito tutte le collezioni leggendole dal database assieme all'oggetto che le contiene. Bisogna però ricordare che questa strada non è percorribile quando la mole dei dati è notevole o lo schema del database non lo consenta; circostanze, queste, molte frequenti nella pratica.

Fare scattare i caricamenti differiti ovvero prima di passare al livello di presentazione gli oggetti del modello fare scattare tutti i caricamenti differiti. Spesso gli ORM hanno metodi appositi per fare questo (ad esempio Hibernate ha il metodo Hibernate.initialize()).

Vantaggi di Pojo Façade

Confrontando Pojo Façade, in particolare con gli EJB Façade, emergono diversi punti di forza che riassumo brevemente:

sviluppo più facile e (quindi) veloce per la possibilità di testare fuori dal container e l'assenza di DTO che, come visto, possono essere molto onerosi in termini di tempo.

possono eliminare la necessità di un container visto che non è più necessario per transazioni e sicurezza potrebbe non essere necessario utilizzare un container completo JEE nel progetto.

demarcazione flessibile delle transazioni la configurazione tramite AOP delle transazione può avere un livello di flessibilità inaudito nel mondo EJB.

Confrontando invece Pojo Façade con il pattern del modello esposto (presentato nel prossimo paragrafo) si possono isolare i seguenti vantaggi:

livello di presentazione facilitato perché non è richiesta una tecnica particolare di gestione delle transazioni sul livello di presentazione, come vedremo

vista coerente dei dati del database tutti i dati sono esposti dai pojo façade e sono quindi raccolti all'interno di un'unica transazione. Questo può non essere vero nel pattern del modello esposto.

Svantaggi di Pojo Façade

Gli svantaggi principali rispetto agli EJB façade sono:

mancanza di standard lo standard JEE non prevede Spring (per ora e purtroppo) quindi i problemi possono essere di vario genere. In verità Spring è largamente riconosciuto nella comunità di sviluppatori tanto che moltissime librerie o sono espressamente dedicata a Spring oppure sono facilmente integrabili. Inoltre molte configurazioni standard sono riconosciute da Spring (ad esempio le annotazioni per i web service o per la persistenza). Comunque a titolo di esempio i problemi legati alla mancanza di standard possono essere:

ottenere il pojo façade il client potrebbe non sapere come ottenere un'istanza del façade, ad esempio un generatore che espone una classe come end point di un servizio web potrebbe non sapere come ottenere un'istanza del pojo façade.

sicurezza la dichiarazione dei vincoli di sicurezza avviene in modo fuori standard e quindi non portabile.

transazioni iniziate su client remoti attualmente non è supportato da framework come Spring: i pojo façade non possono partecipare a transazioni avviate esternamente.

Con riferimento al pattern del modello esposto i principali svantaggi sono:

problemi con gli oggetti staccati come visto nei paragrafi precedenti può richiedere una codifica minuziosa e difficile accertare che tutti gli oggetti siano effettivamente staccati dal motore di persistenza. Inoltre gli errori si presentano solo a runtime, cosa che rende più difficile la loro individuazione.

incapsulamento del modello come abbiamo visto può risultare molto complesso nascondere alcuni metodi ai client.

9.4.2 Modello esposto

Un altra tecnica per evitare l'impiego dei DTO non si limita ad utilizzare le classi di modello per la presentazione ma imbocca la strada più radicale di rinunciare ad un livello di façade fornendo direttamente accesso ad un modello non più staccato (detached) ma attivo e transazionale. Il modello esposto è noto anche col nome di Open Session In View o anche Open Persistence Manager In View.

Quando è possibile utilizzare il modello esposto

Per poter utilizzare questo modello sono necessarie due condizioni inderogabili per il livello di presentazione che deve infatti:

- poter gestire la sessione dell'ORM
- accedere da locale al modello (sono escluse connessioni remote)

In assenza di queste due condizioni è necessario ripiegare su soluzioni alternative come pojo façade.

Come funziona

Per consentire al modello di funzionare nel livello di presentazione è necessario occuparsi di due aspetti fondamentali:

gestire la sessione La sessione utilizzata dall'ORM deve rimanere aperta per tutta la durata di una richiesta HTTP o addirittura per diverse richieste. Per fare questo è una soluzione consolidata ricorrere ad un filtro HTTP che si occupi prima di mantenere aperta la sessione ed

infine di chiuderla anche in presenza di eccezioni. Spring presenta già dei filtri per svolgere questa funzione, tra cui OpenSessionInViewFilter. I vantaggi dell'impiego di un filtro sono:

sicurezza il filtro viene sempre invocato all'inizio ed alla fine della richiesta

riusabilità lo stesso filtro può essere riusato in diverse applicazioni

ortogonalità è possibile aggiungere e rimuovere il filtro senza modificare il codice del livello di presentazione.

gestire le transazioni La gestione delle transazioni potrebbe avvenire a livello di presentazione (ovvero essere delegata al filtro della sessione) oppure al livello del modello (ovvero gestita con AOP). Nel primo caso la transazione viene aperta all'inizio della richiesta dal filtro della sessione e chiusa al termine. Nel secondo caso (AOP) è possibile aprire e chiudere le transazioni in base ad uno schema molto più flessibile, come ad esempio attorno alle classi di modello che si occupano dei servizi. Entrambi gli approcci presentano vantaggi e svantaggi:

9.4.3 Conversazioni

Nelle applicazioni web una singola operazione di business può essere completata solo attraverso diverse richieste HTTP; ad esempio mediante la compilazione da parte dell'utente di alcune pagine in sequenza e solo al completamento dell'ultima l'operazione si considera o conclusa o annullata. In termini di interazione utente queste operazioni si chiamano conversazioni (o transazioni lunghe o anche transazioni utente) e sono gestite in modo completamente diverso nel pattern dei pojo façade piuttosto che nel modello esposto. Nei pojo façade la sessione dell'ORM inizia e si conclude all'interno della singola richiesta HTTP, per cui all'interno di una conversazione (che si compone di diverse richiesta HTTP) vengono usate sessioni diverse. Gli oggetti di modello prima di passare al livello di presentazione sono staccati (detached) dal motore di persistenza e poi riattaccati (reattached) nella richiesta HTTP successiva. Questa modalità operativa prevede quindi transazioni sul database limitate all'interno delle richieste HTTP e dedica molta attenzione alla fase di attach e detach degli oggetti dall'ORM. Nel pattern del modello esposto invece si utilizza la stessa sessione dell'ORM per tutta la durata della conversazione. Gli oggetti di modello passati al livello di presentazione non sono mai staccati (detached) ma rimangono sempre nel contesto di persistenza dell'ORM. Le transazioni sul database sono invece aperte e chiuse all'interno delle richieste HTTP per evitare dei lock sulle righe delle tabelle ed il conseguente degrado per tutta l'applicazione. Per assicurare la

coerenza dei dati tra le varie transazioni è necessario abilitare un meccanismo per individuare eventuali variazione sui dati trattati, ad esempio una politica di lock ottimistico.

Capitolo 10

Presentazione Web

10.1 Scopo

TODO: Di cosa si occupa questo livello?

10.2 Tecnologie

TODO: Panoramica brevissima su JSF, Spring WebFlow e Spring MVC.

10.3 Pagina di lista

TODO: Partendo da un esempio si provvede a spiegare quali sono i file coinvolti e come devono essere configurati. Si ricorda che esiste un generatore per questo.

10.4 Pagina di form

TODO: Partendo da un esempio si provvede a spiegare quali sono i file coinvolti e come devono essere configurati. Si ricorda che esiste un generatore per questo.

10.5 Componenti aggiuntivi

Per sopperire alla mancanza di alcuni componenti nel tool kit di IceFaces (ad esempio di una tabella in grado di visualizzare una sotto tabella annidata)

Regola kit propone alcuni componenti da considerarsi sostituti temporanei, ovvero da utilizzare solo fino a quando non saranno disponibili delle versioni ufficiali con le medesime caratteristiche fornite IceFaces. Includere i componenti di Regola kit all'interno di una pagina template di facelets richiede di specificare tra i namespace utilizzati anche http://www.regola-kit.org/jsf, come nell'esempio;

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  ...
  xmlns:regola="http://www.regola-kit.org/jsf">
...
```

10.5.1 expandibleTable

Questo componente espande HtmlDataTable di IceFaces facendogli disegnare, a richiesta, una riga aggiuntiva subito dopo ogni riga disegnata. Il contenuto di questa riga aggiuntiva è descritto nel facet innerRow come nell'esempio seguente:

```
<regola:expandibleTable id="data" var="primo"
value="#{dataController.model}">

  <f:facet name="innerRow">
        <ice:outputText value="Riga aggiuntiva" />
  </f:facet name="innerRow">

  <ice:column>...</ice:column>
  <ice:column>...</ice:column>
      ...

</regola:expandibleTable>
```

Qualsiasi componente descritto dentro il facet innerRow viene renderizzato nella riga aggiuntiva, per cui può essere utilizzata per disegnare una sotto tabella relativa alla riga precedente, ovvero una tabella nidificata. Ad esempio supponiamo di avere un semplice modello di due classi: Primo che contiene una lista di Secondo.

```
public class Primo {

  int id;
  String nome;
  String cognome;

  List<Secondo> figli = new ArrayList<Secondo>();

  //accessor
}

public class Secondo {

  String uno;
  String due;
```

```
17   //accessor
}
```

Il componente expandibleTable non richiede come modello una lista oggetti del tipo Primo ma una lista della classe di utilità ExpandibleItem che wrappano la classe originale ad aggiungono una proprietà booleana per indicare se la riga aggiuntiva della tabella deve essere disegnato o meno. Per effettuare questo wrapping si può utilizzare il metodo di utilità wrapList.

```
ArrayList<Primo> primi = new ArrayList<Primo>();
2
Primo primo;
4
primo = new Primo(1,"Adam", "Smith");
6 primo.figli.add(new Secondo("1.1","1.2"));
primo.figli.add(new Secondo("1.3","1.4"));
8 primo.figli.add(new Secondo("1.5","1.6"));
primi.add(primo);
10
primo = new Primo(2,"Karl", "Marx");
12 primo.figli.add(new Secondo("2.1","2.2"));
primo.figli.add(new Secondo("2.3","2.4"));
14 primo.figli.add(new Secondo("2.5","2.6"));
primi.add(primo);
16
...
18
List model = ExpandibleItem.wrapList(primi);
```

Il modello così trattato deve essere fornito da un managed bean ad esempio dataController:

```
1 public class DataController {

3   List model;

5   public DataController()
    {
7             ...
      model = ExpandibleItem.wrapList(primi);
9   }

11  //accessor

13 }
```

L'ultimo tassello del puzzle è il template faceltet da utilizzare che potrebbe essere simile a questo:

```
1 <ice:form>
    <regola:expandibleTable id="data" var="primo"
    value="#{dataController.model}">
3
   <f:facet name="innerRow">
5       <ice:dataTable var="figlio" value="#{primo.target.figli}">
     <ice:column>
7       <f:facet name="header">
          <ice:outputText value="UNO" />
```

```
9       </f:facet>
            <ice:outputText value="#{figlio.uno}" />
11        </ice:column>

13    <ice:column>...</ice:column>
     </ice:dataTable>
15     </f:facet>

17      <ice:column>
    <f:facet name="header">
19  <ice:outputText value="" />
    </f:facet>
21  <ice:selectBooleanCheckbox value="#{primo.expanded}"
            partialSubmit="true" />
23  </ice:column>

25  <ice:column>
       <f:facet name="header">
27        <ice:outputText value="ID" />
       </f:facet>
29     <ice:outputText value="#{primo.target.id}" />
    </ice:column>
31

33  <ice:column>...</ice:column>

35  </regola:expandibleTable>
</ice:form>
```

Da notare che le espressioni utilizzate per il binding devono tener conto che l'oggetto in lista è del tipo ExpandibleItem e non del tipo Primo, per cui per accedere alle proprietà di quest'ultimo è necessario passare attraverso la proprietà ExpandibleItem.target. Infine si consideri come la proprietà ExpandibleItem.expanded determini la visibilità della riga aggiuntiva.

Capitolo 11

Application mashup

11.1 Servizi Web SOAP

Un modo molto comune per consentire ad applicazioni esterne di accedere al livello di servizio delle applicazioni scritte con Regola kit è quello di esportare uno o più bean situati in quel livello tramite servizio web (web service). Per farlo è necessario che la classe sia progettata in modo da consentire la costruzione della busta SOAP o, in generale, in modo da consentire la conversione dei parametri scambiati in una qualche forma di XML; ad esempio è buona norma utilizzare tipo serializzabili, rendere transitorie le associazioni che non si intende trasferire nel servizio, gestire le ricorsioni nel grafo di oggetti, ecc. Regola kit utilizza CXF per gestire in toto i servizi web, per cui si rimanda alla documentazione ufficiale per le tante e utili opzioni disponibili. Di seguito presenteremo un esempio piuttosto comune di basato su JAX-WS realizzato a partire da codice Java esistente nel livello di presentazione. La prima cosa è marcare l'interfaccia del servizio con l'annotazione WebService :

```
@WebService
public interface HelloWorldManager {
    String sayHi(@WebParam(name="text") String text);
}
```

Anche la classe che realizza questa interfaccia deve essere annotata per precisare, ad esempio, il nome del servizio stesso:

```
@WebService(endpointInterface = "demo.service.HelloWorld",
            serviceName = "HelloWorld")
public class HelloWorldManagerImpl implements HelloWorldManager {

    public String sayHi(String text) {
        return "Hello " + text;
    }

```

```
}
```

A questo punto rimane da configurare il bean in modo tale che costituisca un Service End Point. Nell'esempio di seguito viene modificato il file applicationContext-services.xml in modo da prevedere anche il protocollo di sicurezza WS-Security:

```
<jaxws:endpoint id="helloWorld"
  implementor="demo..service.impl.HelloWorldManagerImpl"
  address="/services/HelloWorld">
   <jaxws:features>
     <bean class="org.apache.cxf.feature.LoggingFeature" />
   </jaxws:features>
   <jaxws:inInterceptors>
   <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <constructor-arg>
    <map>
      <entry key="action" value="UsernameToken Timestamp" />
      <entry key="passwordType" value="PasswordText" />
      <entry key="passwordCallbackClass" value="ServerPasswordCallback" />
   </map>
      </constructor-arg>
    </bean>
   </jaxws:inInterceptors>
</jaxws:endpoint>
```

Da notare tra le features l'abilitazione del log e tra gli inIntercaptors la configurazione della classe WSS4JInInterceptor che predispone il servizio di sicurezza WS-Security, in questo caso basato su Timestamp e su semplice password in chiaro. Conclude il tutto la classe ServerPasswordCallback che si occupa di verificare le password in ingresso.

```
public class ServerPasswordCallback implements CallbackHandler {

  public void handle(Callback[] callbacks) throws IOException,
     UnsupportedCallbackException {

  WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
  if (pc.getIdentifer().equals("joe")) {
  if (!pc.getPassword().equals("password")) {
      throw new SecurityException("wrong password");
    }
  }
  }
}
```

CXF consente inoltre di configurare un client per il servizio in modo analogo ance se un po' prolisso:

```
<bean id="prenotazioni" class="demo.service.HelloWorldManager"
  factory-bean="clientFactory" factory-method="create" />

<bean id="clientFactory" class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
  <property name="serviceClass" value="it.kion.service.PrenotazioniManager"
  />
  <property name="address"
  value="http://localhost/soap/services/HelloWorld" />
  <property name="outInterceptors">
```

```
    <list>
9     <bean class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
       <constructor-arg>
11       <map>
     <entry key="action" value="UsernameToken Timestamp" />
13   <entry key="passwordType" value="PasswordText" />
     <entry key="user" value="joe" />
15       <entry key="passwordCallbackClass" value="ClientPasswordCallback" />
   </map>
17      </constructor-arg>
     </bean>
19   <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor" />
    </list>
21  </property>
   <property name="inInterceptors">
23   <list>
     <bean class="org.apache.cxf.interceptor.LoggingInInterceptor" />
25   </list>
   </property>
27 </bean>
```

Qui è possibile vedere la configurazione per la sicurezza (speculare a quella lato server) , la configurazione del log e la classe ClientPasswordCallback responsabile di presentare al server le password.

```
1 public class ClientPasswordCallback implements CallbackHandler {

3   public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
5       WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
   // set the password for our message.
7  pc.setPassword("password");
    }
9 }
```

CXF è una libreria davvero completa che consente di realizzare servizi web in modalità diversa (ad esempio partendo dal WSDL piuttosto che dal codice Java, oppure utilizzando SOAP o servizi di tipo REST, ecc.). Si invita quindi a consultare la documentazione per avere un quadro generale delle funzioni disponibili.

■ Per utilizzare CXF all'intenro di JBOSS è necessario impostare il parametro di avvio della virtual machine javax.xml.soap.MessageFactory al valore com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl. ■

11.2 Servizi Web REST

Un servizio di tipo REST può essere realizzato sempre annotando una classe di tipo manager (nel livello quindi di servizio) nel modo seguente:

```
1 @Path("/nome-servizio/")
  @ProduceMime("text/xml")
3 public class RestManager {

5   @POST @Path("/foo/{id}/{nome}")
```

```
    public Dto postFoo(@PathParam("id") String id, @PathParam("nome") String
    nome, Dto parametro) {

      return new Dto(id + ":" + nome + ":" + parametro.prova);
    }

    @PUT @Path("/foo/{id}/{nome}")
    public Dto putFoo(@PathParam("id") String id, @PathParam("nome") String
    nome, Dto parametro) {

      return new Dto(id + ":" + nome + ":" + parametro.prova);
    }

    @GET @Path("/foo/{id}/{nome}")
    public String getFoo(@PathParam("id") int id, @PathParam("nome") String
    nome) {

      return "GET di:" + id + ":" + nome;
    }

    @DELETE @Path("/foo/{id}/{nome}")
    public String deleteFoo(@PathParam("id") int id, @PathParam("nome") String
     nome) {

      return "DELETE di:" + id + ":" + nome;
    }

}
```

Le cose importanti da sottolineare sono:

1. l'annotazione @Path che specifica la url alla quale risponde il servizio ed le singole operazioni, nell'esempio /nome-servizio/foo.
2. I parametri di tipo primitivi (stringhe, diciamo) sono passati all'operazione aggiungendoli alla url. Sempre nell'annotazione @Path è possibile indicare i nomi dei parametri (@Path(/foo/id/nome))che poi saranno effettivamente passati al metodo Java con l'annotazione @PathParam (ad esempio @PathParam(id)).
3. il metodo http al quale l'operazione deve rispondere ad esempio @POST, @GET , @PUT e @DELETE

La registrazione del servizio in Spring richiede semplicemente qualcosa di simile ad questo:

```
  <jaxrs:server id="testService" address="/">
    <jaxrs:serviceBeans>
        <ref bean="testServiceImpl" />
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <bean id="testServiceImpl" class="it.kion.service.RestManager" />
```

Per invocare i servizi di tipo REST Regola kit mette a disposizione RestClient, una piccola classe che dovrebbe rendere agevole l'invocazione di servizi. Ad esempio: per un invocare un servizio col metodo http GET si può procedere come segue:

```
RestClient client = new RestClient();
String result = client.get(url, param1, param2, ..., paramN);
```

I parametri sono utilizzati per costruire la url che diventa qualcosa di simile ad (url/param1/param2/.../paramN). Eánche possibile passare una singola entità nel body della richiesta HTTP, tipicamente una frammento di xml che rappresenta un'istanza di un oggetto (creato con la classe di utilità di Regola kit JAXBMarshaller):

```
Dto dto = new Dto();
String dtoXml = toXml("org.regola.ws", "dto", dto);
String result = client.post(url, dtoXml, 1, "salve!");
```

Anche il valore di ritorno dei servizi spesso sono dei documenti xml che rappresentano un oggetto; per ottenere l'oggetto di partenza si può ricorrere sempre alla classe JAXBMarshaller:

```
String result = client.put(url, dtoXml, 1, "salve!");
Dto dto = fromXml("org.regola.ws", result);
```

Nota bene: prima di effettuare le conversioni oggetto/xml è necessario utilizzare il compilatore di JAXB per annotare le classi coinvolte e produrre un oggetto del tipo ObjectFactory (si veda la documentazione di JAXB). Ad esempio, partendo da uno schema xml che descrive il documento relativo ad una certa classe, diciamo Dto, bisogna invocare il compilatore di JAXB come segue:

```
xjc schema1.xsd -p it.il.tuo.package -d src/main/java
```

A questo punto tutte le classi coinvolte saranno create nel package it.il.tuo.package unitamente alla classe ObjectFactory.

Infine se per qualche bizzarra circostanza si disponesse solo della classe e non dello schema relativo, sarà possibile usare il metodo di utilità JAXBMarshaller.generateSchema() per ottenere uno schema di base.

11.3 Portlet

Le pagine di delle applicazioni Regola kit possono essere fruite normalmente attraverso il browser (come abbiamo descritto nei capitoli precedenti) e, nel contempo, esportate come Portlet; questo doppio canale di fruizione offre il vantaggio di poter esportare la vostra applicazione all'interno di portale, di testare la vostra Portlet semplicemente dal browser e, infine, di poter utilizzare lo stesso stack operativo di Regola kit (dao, manager, Model Pattern, ecc.) per realizzare Portlet.

Per trasformare una pagina web in una Portlet è necessario adottare alcune accortezza all'interno del modello facelet (il file con estensione .xhtml)

e specificare alcune configurazioni. Per quando riguardo il primo punto bisogna limitarsi ad inserire il tag ice:portlet dopo il tag f:view e prima di tutta la gerarchia di componenti. Ecco un semplice esempio di pagina abilitata ad essere un modello di Portlet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ice="http://www.icesoft.com/icefaces/component">
<body>
  <f:view>
    <ice:portlet>
      Portlet di esempio realizzata con Regola kit
    </ice:portlet>
  </f:view>
</body>
</html>
```

Per quanto riguarda le configurazioni bisogna intervenire prima di tutto sul file /WEB-INF/portlet.xml per elencare le Portlet esportate dall'applicazione e per ciascuna indicare il nome (nell'esempio MyPortlet) e la pagina web da utilizzare come modello (ad esempio /myportlet.html). Per le altre configurazioni si rimanda alla documentazione ufficiale relativa alle Portlet. Ecco un esempio di configurazione:

```
<?xml version="1.0"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
    <portlet>
        <portlet-name>MyPortlet</portlet-name>
        <display-name>Regola kit Portlet</display-name>
        <portlet-class>com.icesoft.faces.webapp.http.portlet.MainPortlet</portlet-class>
        <init-param>
            <name>com.icesoft.faces.VIEW</name>
            <value>/myportlet.html</value>
        </init-param>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
        </supports>
        <portlet-info>
            <title>Regola kit Portlet Example</title>
            <short-title>My Portlet</short-title>
            <keywords>regola-kit icefaces portlet</keywords>
        </portlet-info>
    </portlet>
</portlet-app>
```

A questo punto la vostra applicazione è in grado di essere consegnata dentro un portale aderente alle specifiche Portlet 1.0, ad esempio LifeRay o JeetSpeed alla cui documentazione rimandiamo per i dettagli del deploy. Di seguito però illustreremo come importare la nostra applicazione all'in-

terno di un container Pluto che consiste semplicemente nell'aggiungere all'interno del /WEB-INF/web.xml una servelt del tipo PortletServlet associata ad ogni Portlet esposta (nell'esempio MyPortlet) e mappata all'indirizzo /PlutoInvoker/NomePortlet. Ecco il frammento da aggiungere al file web.xml:

```
<servlet>
        <servlet-name>MyPortlet</servlet-name>
        <servlet-class>org.apache.pluto.core.PortletServlet</servlet-class>
        <init-param>
             <param-name>portlet-name</param-name>
             <param-value>MyPortlet</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
      <servlet-name>MyPortlet</servlet-name>
      <url-pattern>/PlutoInvoker/MyPortlet</url-pattern>
  </servlet-mapping>
```

Infine bisogna consegnare la nostra applicazione nella stessa istanza del container (ad esempio di Tomcat 5.5) in cui sta girando il container Pluto. Ricordiamo che le applicazioni Regola kit sono configurate di default per essere un container Pluto (oltre che un fornitore di Portlet) e quindi possono visualizzare Portlet di altre applicazioni (realizzate o meno con Regola kit). Vediamo come.

11.4 Mashup

Un'applicazione realizza un mashup quando le proprie pagine raccolgono all'interno frammenti di altre applicazioni. Regola kit realizza il mashup di Portlet e fornisce strumenti per agevolare la realizzazioni di Portlet come descritto nel paragrafo precedente. Le applicazioni Regola kit sono di default dei contenitori di Portlet basati su Pluto, quindi non è necessario realizzare nessuna configurazione per abilitare questa funzionalità tranne ricordarsi di abilitare, a livello di container, la funzionalità di accedere al altri contesti. In Tomcat 5.5 questo si realizza specificando nel descrittore di contesto qualcosa di simile a questo:

```
<Context crossContext="true" />
```

Ricordiamo che il contesto si può configurare dentro la cartella conf di Tomcat, oppure dentro conf/Catalina/localhost o addirittura dentro la web root nel file META-INF/context.xml.

I Portlet ad importare all'interno della vostra applicazione devono essere consegnati nello stesso application server dove gira il container. Attualmente è possibile utilizzare esclusivamente delle pagine jsp per effettuare il mashup

dei vari portlet. Ecco un esempio di pagina in cui viene inclusa una Portlet chiamata MyPortlet, fornita da un'applicazione che si trova nel contesto homes:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://portals.apache.org/pluto" prefix="pluto" %>
<html>
<head>
    <title>Prova</title>
    <style type="text/css" title="currentStyle" media="screen">
        @import "<c:out value="${pageContext.request.contextPath}"/>/pluto.css";
        @import "<c:out value="${pageContext.request.contextPath}"/>/portlet-spec-1.0.css";
    </style>
    <script type="text/javascript" src="<c:out value="${pageContext.request.contextPath}"/>/pluto.js"></script>
</head>
    <body>
        <h1>Questa pagina include la portlet con nome MyPortlet</h1>
        <pluto:portlet portletId="/homes.MyPortlet">
            <pluto:render/>
         </pluto:portlet>
   </body>
</html>
```

Capitolo 12

Sicurezza

12.1 Autenticazione

L'autenticazione mediante SSO di ogni applicazione richiede una serie di scambi tra l'applicazione stessa e Cas che sono rappresentati nel grafico seguente e descritti poco dopo:

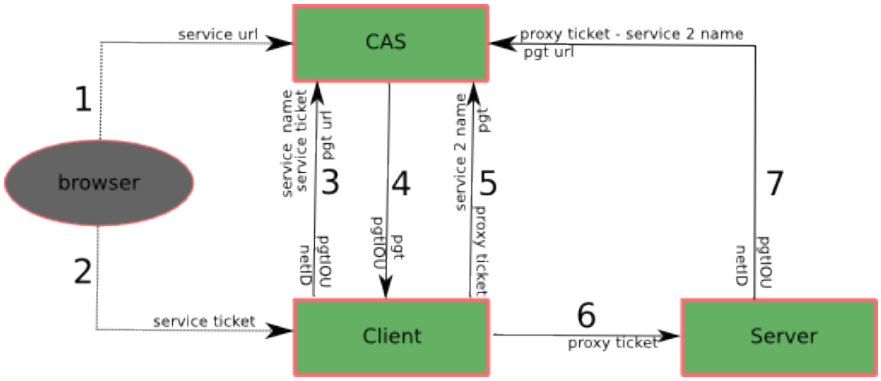

Figura 12.1: Processo di autenticazione

1. Quando un browser tenta di accedere per la prima volta ad un'applicazione (d'ora in poi Client) realizzata con Regola kit e protetta con cas il browser viene reindirizzato all'applicazione Cas (d'ora in poi CAS) passandole come parametro una stringa che rappresenta il servizio richiesto. Su CAS attraverso un form HTML l'utente inserisce username e password. A questo punto, se l'autenticazione è andata a buon fine, CAS conosce il netID dell'utente autenticato, ad esempio nicola.santi@unibo.it

2. CAS ridirige il browser all'applicazione Client passandole come parametro il service ticket, un numero che servirà poi in seguito.
3. Client richiama CAS passandogli il service ticket appena ricevuto ed una URL chiamata pgt url che CAS userà in seguito. A questa chiamata CAS risponde con il netID dell'utente (nicola.santi@unibo.it) ed un altra stringa, il pgtIOU che servirà in seguito.
4. CAS richiama il client alla url che aveva ricevuto nella richiesta precedente (la pgt url) ed invia a Client di nuovo lo stesso pgtIOU di prima perché Client verifichi che siano uguali ed inoltre invia un pgt da usare in seguito nell'autenticazione di tipo proxy.
   A questo punto l'autenticazione di un utente che voleva accedere a Client tramite browser è terminata. Rimane da spiegare l'autenticazione di tipo proxy, ovvero se Client decide di invocare ad esempio un servizio web di un'altra applicazione, diciamo Server senza doversi riautenticare.
5. Client invia il pgt a CAS che gli risponde con il proxy ticket, una stringa da passare a Server.
6. Client richiama Server e gli passa il proxy ticket.
7. Server contatta CAS passandogli il proxy ticket e la pgt url. A questa chiamata CAS risponde con il netID dell'utente (nicola.santi@unibo.it) ed un altra stringa, il pgtIOU che servirà in seguito. Questo passaggio è del tutto simile a quello numero 3. Seguirà una passaggio simile a 4 in cui CAS chiamerà Server passandogli pgt e pgtIOU.

Le url coinvolte nei passaggi precedenti sono configurabili all'interno dell'applicazione Client modificando alcune proprietà nel file src/main/webapp/WEB-INF/cas.properties. La tabella seguente riporta le url coinvolte per ogni passo unitamente alla proprietà da configurarare ed il suo valore di default:

step	URL	parametro
1	login url	cas.loginUrl=/login
2	service url	cas.main=/personale/j_acegi_cas_security_check
3,7	validation url	cas.validationUrl=/proxyValidate
4	pgt url	cas.pgtUrl=/casProxy
5	proxy url	cas.proxyUrl=/proxy
6	la url specifica del servizio	

12.2 Configurazione di Cas

Affinché la propria applicazione (Client o Server) possa usufruire del SSO deve registrare presso Cas uno (o più servizi). I servizi sono essenzialmente

identificati da una stringa che deve corrispondere (anche solo parzialmente come vedremo) con la service url richiamata da Cas al punto 2. Per non restare nel vago immaginiamo di dover registrare un servizio per un'applicazione che risponde alla url https://tirocini.unibo.it e la cui service url sia https://tirocini.unibo.it/j_acegi_cas_security_check. Il nome del servizio può coincidere con questa url oppure con una suo sottoparte come ad esempio https://tirocini.unibo.it/* (si prega di notare l'asterisco).

Si rammenta che nel caso di autorizzazione di tipo proxy è necessario utilizzare due servizi distinti, uno per l'autenticare l'applicazione Client da passare nel passo 1 ed quello da utilizzare per l'applicazione Server per il passo 5; da notare che il secondo servizio deve coincidere (anche solo parzialmente) con la url di Server mentre il primo con la url di Client.

12.3 Configurazione di Client

Nonostante le possibilità di Cas siano particolarmente ampie, la configurazione di default realizzata da Regola kit dovrebbe soddisfare le esigenze più comuni di un'applicazione web. In questo caso la configurazione si limita a modificare il nome del servizio e l'host dove risponde Cas all'interno del file src/main/webapp/WEB-INF/cas.properties.

```
# servizi registrati per questa applicazione (da riportare sul server CAS)
cas.service.proxied=/services/
cas.main=/personale/j_acegi_cas_security_check

# url di test
cas.hostUrl=https://localhost:8443/cas
cas.ourAppUrl=https://localhost:8443/${artifactId}

# url di produzione
prod.cas.host=https://localhost:8443/cas
prod.cas.ourAppUrl=https://localhost:8443/${artifactId}
```

Nell'esempio abbiamo configurato due servizi, uno per l'autenticazione di tipo proxy ovvero quando la nostra applicazione funziona come Server alla url /services e l'altra per quando l'applicazione nostra funziona come Client. In questo caso conviene usare come nome del servizio la nostra service url, come nell'esempio. Le altre proprietà specificano le url della nostra applicazione e del server Cas sia in produzione sia in test.

La prossima configurazione da modificare si trova del file src/main/webapp/WEB-INF/security-cas.xml e prevede l'impostazione delle url da proteggere e per ciascuna di esse quale gruppo a titolo per visualizzarle.

```
  <bean id="filterChainProxy"
  class="org.acegisecurity.util.FilterChainProxy">
    <property name="filterInvocationDefinitionSource">
      <value>
```

```
        CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
        PATTERN_TYPE_APACHE_ANT
        /images/**=#NONE#
        /scripts/**=#NONE#
        /styles/**=#NONE#
        /portal.jsp=httpSessionContextIntegrationFilter,logoutFilterMain,casProcessingFilter,
        /personale/**=httpSessionContextIntegrationFilter,logoutFilterMain,casProcessingFilter
      </value>
      <!--
        Put channelProcessingFilter before
        securityContextHolderAwareRequestFilter to turn on SSL switching
      -->
    </property>
  </bean>

  <!-- specifica ruoli che possono accedere alle diverse url -->
  <bean id="filterInvocationInterceptor"
    class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="casAuthenticationManager" />
    <property name="accessDecisionManager" ref="accessDecisionManager" />
    <property name="objectDefinitionSource">
      <value>
        PATTERN_TYPE_APACHE_ANT
        /portal.jsp=admin,user
        /esterni/richiesta*.*=ROLE_ANONYMOUS,admin,user
        /esterni/loginMigrazione.*=ROLE_ANONYMOUS,admin,user
        /**/*.htm*=admin,user
        /**/*.jsp*=admin,user
      </value>
    </property>
  </bean>
```

Questo è quanto occorre generalmente per consentire alla propria applicazione di funzionare sia come Client che come Server. Nel resto di questo capitolo mostreremo le altre possibili variazioni che comprendono la modifica del file src/main/webapp/WEB-INF/security-cas.xml ma richiedono una conoscenza più approfondita di Cas e Acegi Securty.

12.4 Presentazione

Il sistema di sicurezza è costruito attorno ad una catena di oggetti chiamati filtri che vengono eseguiti in cascata e cooperano per assolvere i diversi compiti nei quali la funzione della sicurezza si concretizza. Esistono molti filtri standard ma è comunque possibile aggiungerne di personalizzati ad esempio per aggiungere un sistema di autenticazione proprietario o non ancora implementato. L'elenco che segue indica i filtri standard nell'ordine con cui sono eseguiti e per ciascuno il nome della classe che lo implementa ed un alias per riferirsi ad esso:

CHANNEL_FILTER	ChannelProcessingFilter
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter
SESSION_CONTEXT_INTEGRATION_FILTER	HttpSessionContextIntegrationFilter
LOGOUT_FILTER	LogoutFilter
X509_FILTER X509	PreAuthenticatedProcessigFilter
PRE_AUTH_FILTER	Subclass of AstractPreAuthenticatedProc
CAS_PROCESSING_FILTER	CasProcessingFilter
AUTHENTICATION_PROCESSING_FILTER	AuthenticationProcessingFilter
BASIC_PROCESSING_FILTER	BasicProcessingFilter
SERVLET_API_SUPPORT_FILTER	classname
REMEMBER_ME_FILTER	RememberMeProcessingFilter
ANONYMOUS_FILTER	AnonymousProcessingFilter
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter
NTLM_FILTER	NtlmProcessingFilter
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor
SWITCH_USER_FILTER	SwitchUserProcessingFilter

Prima di calarci nel dettaglio dei filtri più importanti conviene concentrarci sulla configurazione più importante del sistema, ovvero quella che specifica quali di questi filtri utilizzare per una certa url e che si concretizza nella definizione del bean filterChainProxy:

```
<bean id="filterChainProxy"
      class="org.acegisecurity.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
     <value>
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
    PATTERN_TYPE_APACHE_ANT
    /images/**=#NONE#
    /scripts/**=#NONE#
    /styles/**=#NONE#
    /**=httpSessionContextIntegrationFilter,logoutFilter,authenticationProcessingFilter,securityContex
    </value>
  </property>
</bean>
```

L'aspetto è piuttosto intimidatorio sulle prime ma in effetti si limita a specificare le catene di filtri da adottare per le url indicate, nell'esempio dispone di non utilizzare nessun filtro (#NONE#) per le url /images/**, /scripts/**, e /style/**. Per tutte le altre url (/**) invece si utilizza la catena dei sette filtri specificati. Si può notare che le url sono indicate utilizzando la notazione di ANT essendo stato passato il parametro PATTERN_TYPE_APACHE_ANT, alternativamente si potevano utilizzare le espressioni regolari. Come si diceva ogni filtro assolve un compito specifico e deve essere configurato per modificarne comportamento; ad esempio l'ultimo dei sette filtri specificati nella catena, filterInvocationInterceptor, si occupa di stabilire quali utenti (o quali gruppi di utenti) possano accedere alle pagine

protette. Si può configurare come segue:

```
<bean id="filterInvocationInterceptor"
  class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager"

                  ref="authenticationManager" />
  <property name="accessDecisionManager"
        ref="accessDecisionManager" />
  <property name="objectDefinitionSource">
    <value>
    PATTERN_TYPE_APACHE_ANT
    /services/*=ROLE_ANONYMOUS,admin,user
    /login.*=ROLE_ANONYMOUS,admin,user
    /**/*.html*=admin,user
    </value>
  </property>
</bean>
```

Iniziamo dalla proprietà dal nome oscuro objectDefinitionSource che specifica una serie di url (sempre nel formato ANT) e per ciascuno quale utente o ruolo ha il permesso di accesso; ad esempio le url del tipo /services/* possono essere accedute da tutti (utenti anonimi cioè quelli con il ruolo ROLE_ANONYMOUS, l'utente admin e l'utente user). Stessa cosa per la pagina di login mentre per tutte le altre pagine l'accesso è consentito solo agli utenti user ed admin. Da notare come queste regole siano applicate in cascata, così come quelle specificate nella configurazione della catena dei filtri. Come spesso accade la configurazione di un bean ne richiede altri, nel caso di filterInvocationInterceptor si utilizzano anche i bean authenticationManager e accessDecisionManager. Trattiamo prima quest'ultimo che ha il compito di stabilire se un utente possa o meno accedere; di solito si utilizza una configurazione standard per in base al ruolo si può accedere o meno. La configurazione è la seguente.

```
<bean id="accessDecisionManager"
  class="org.acegisecurity.vote.AffirmativeBased">
  <property name="allowIfAllAbstainDecisions" value="false" />
  <property name="decisionVoters">
    <list>
      <bean class="org.acegisecurity.vote.RoleVoter">
        <property name="rolePrefix" value="" />
      </bean>
    </list>
  </property>
</bean>
```

L'altro bean utilizzato per configurare filterInvocationInterceptor è authenticationManager, si tratta di un bean importantissimo che è utilizzato anche da altri filtri (ad esempio authenticationProcessingFilter, il terzo filtro della catena di sette configurata sopra). Il bean authenticationManager si occupa infatti di fornire i componenti che effettuano l'autenticazione dell' utente corrente (ad esempio ma non necessariamente utilizzando username

e password) e fornire l'oggetto che rappresenta quell'utente ed i ruoli ricoperti all'interno del sistema di sicurezza. Ora, dal momento che i metodi per autenticare un utente potrebbero essere diversi, ad esempio prima su una database poi in caso di fallimento su un altro poi in caso di fallimento tramite LDAP e così via authenticationManager contiene la lista di questi servizi di autenticazione chiamati authentication provider:

```
<bean id="authenticationManager"
  class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider" />
      <ref local="anonymousAuthenticationProvider" />
      <ref local="rememberMeAuthenticationProvider" />
    </list>
  </property>
</bean>
```

Come si può immaginare esistono già pronti per l'uso diversi authentication provider come l' anonymousAuthenticationProvider che provvede le credenziali per l'utente anonimo, la cui configurazione si limita a dare un nome (anonymous) a questo particolare utente.

```
<bean id="anonymousAuthenticationProvider"
    class="org.acegisecurity.providers.anonymous.AnonymousAuthenticationProvider">
  <property name="key" value="anonymous" />
</bean>
```

Invece l' authentication provider rememberMeAuthenticationProvider si occupa di autenticare un utente in base ad un cookie precedentemente salvato sul browser, la cui configurazione (riutilizzando l'uthenticationManager) non è riportata di seguito per evitare di perdere il filo del discorso. Ci occupiamo invece del daoAuthenticationProvider perché ricopre un ruolo di rilievo nel sistema di sicurezza. Infatti la classe DaoAuthenticationProvider è in grado di utilizzare tutti i meccanismi di autenticazioni basati su username e password forniti col sistema semplicemente delegando l'individuazione dell'utente e del ruolo ad un bean chiamato userDetailsService.

```
<bean id="daoAuthenticationProvider"
  class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="inMemoryDaoImpl"/>
  <property name="saltSource" ref="saltSource"/>
  <property name="passwordEncoder" ref="passwordEncoder"/>
</bean>
```

Nell'esempio lo userDetailsService utitlizzato è inMemoryDaoImpl che si limita a leggere utenti e password direttamente dal file di configurazione.

12.5 Configurazione semplificata

Nel paragrafo precedente abbiamo specificato la catena dei filtri da applicare per ogni url e poi indicato per ogni url quale utente o gruppo potesse accedere. Infine abbiamo fornito un semplice sistema di autenticazione basato su un file di configurazione di utenti, gruppi e password. In molti hanno pensato che, per quanto flessibile, il sistema di configurazione sopra descritto risulti davvero complicato e finalmente nella versione 2.0 (oltre al cambio di nome da Acegi a Spring Security) è stato introdotto un sistema di configurazione basato su configurazioni di default ed implementato utilizzando un namespace specifico. Il risulto è che tutta la configurazione descritta precedentemente può essere realizzato come segue:

```
<http auto-config='true'>

<intercept-url pattern="/**" access="#NONE#" />
<intercept-url pattern="/**" access="#NONE#" />
<intercept-url pattern="/**" access="#NONE#" />
<intercept-url pattern="/services/*" access="ROLE_ANONYMOUS,admin,user" />
<intercept-url pattern="/**/*.html*" access="admin,user" />
</http>

<authentication-provider>
   <user-service>
    <user name="jimi" password="jimispassword" authorities="ROLE_USER,
    ROLE_ADMIN" />
    <user name="bob" password="bobspassword" authorities="ROLE_USER" />
   </user-service>
</authentication-provider>
```

La cosa importante da segnalare è che la nuova configurazione non sostituisce la precedente ma la realizza in modo automatico; ad esempio il tag <http> ed <intercept-url> definiscono implicitamente filterInvocationInterceptor specificando una catena di filtri di default mentre <authentication-provider> crea un daoAuthenticationProvider e <user-service> un userDetailsService del tipo InMemoryDaoImpl. Questi authentication provider sono poi associati ad un bean authenticationManager proprio come quello specificato precedentemente.

Si rimanda poi alla documentazione ufficiale per esempi in cui si mescolano le due modalità di configurazione in modo da ridurre al minimo il codice necessario a mettere in sicurezza la vostra applicazione.

Capitolo 13

Plitvice Security

Questo documento è ancora in fase di redazione così come la libreria a cui si riferisce.

13.1 Autorizzazioni

Si presentano le strutture dai: - Ruolo - Contesto

13.2 Ruolo

La classe Ruolo raccoglie principalmente una collezione di diritti di cui parleremo ampiamente tra poco. Ogni ruolo è identificato da un intero unico per tutte le applicazioni e presenta una descrizione per chiarine la finalità; quest'ultima può essere utilizzate nei livelli di presentazione. Ogni ruolo ha validità per un'applicazione, ovvero contiene diritti da utilizzare solo con una specifica applicazione. Le applicazioni sono individuate anch'esse da un intero e sono persistite in un'apposita tabella di database. Si diceva che ogni ruolo contiene una collezione di diritti, ovvero di classi del tipo Diritto; gli elementi comuni ad ogni diritto sono un identificativo unico (un intero) ed un Ambito che rappresenta un ulteriore partizionamento oltre a quello già menzionato dell'applicazione, specificata nel ruolo. Ad esempio esiste l'ambito dei tirocini che raccoglie tutti i diritti relativi alla grande area della gestione dei tirocini. Inoltre tutti i diritti fanno riferimento ad un'entità, ovvero un elemento del modello come indicato del Domain Driven Development. L'elenco delle entità e degli è persistito su database, in due tabelle, e a ciascuno è assegnato un identificativo univoco. Infine i diritti possono essere di due tipi: di workflow o applicativi.

13.2.1 Diritti di workflow

I diritti relativi a workflow si riferiscono ad un flusso e specificano le autorizzazioni, ovvero se l'identità assunta dall'utente corrente possa o meno segnalare una transizione oppure ottenere l'elenco di tutti i documenti che si trovino in un certo stato. Di conseguenza ogni diritto di workflow specifica il nome del flusso da utilizzare. I diritti di workflow si ripartiscono, per quanto detto, in diritti da applicare agli stati e diritti da applicare alle transizioni. Questi ultimi, se presenti, consentono all'identità corrente di segnalare una transizione. Il default, ovvero in caso di assenza di uno specifico diritto di questo tipo, la transazione non può essere segnalata da nessuno. I diritti di workflow che si applicano agli stato consentono di elencare tutti i documenti che si trovino in quello stato. In assenza di un particolare diritto chiunque può richiedere l'elenco completo dei documenti per quel certo stato. Da notare come questo tipo di diritti ammette un meccanismo di eccezione che consente anche ai non aventi diritto di elencare tutti i documenti in un certo stato; si rimanda alla proprietà vincoloLettura di AutorizzazioneUtente e l'interfaccia Visibilità per dettagli in merito.

13.2.2 Diritti applicativi

I diritti applicativi sono delle mere etichette, delle stringhe di caratteri che ricoprono un preciso significato solo per una certa applicazione. Ad esempio il permesso individuato dalla stringa OT_INS assume un significato solo per l'applicazione dei tirocini (ovvero quella specificata nel ruolo che contiene il diritto) che conosce quindi come comportarsi; in particolare se il permesso è presente consente l'inserimento di nuove offerte di tirocini.

13.3 Contesti

I contesti costituiscono un sistema per limitare la visibilità delle entità; ogni applicazione presenta un modello di contesto specifico del problema che devono affrontare. In generale comunque ogni contesto è individuato in modo univoco (per l'applicazione che lo usa) con un intero.

13.3.1 Tirocini

Qui si illustrano Regole e dei Destinatari.

13.3.2 Plitvice

Qui si illustrano i contesti di Plitvice.

13.4 Insieme delle identità

Ad ogni utente autenticato potrebbero essere associabile a diversi ruoli o diversi contesti, ad esempio l'utente con netID nicola.santi@unibo.it potrebbe scegliere tra il contesto di amministratore che consente di vedere ogni entità ed il contesto di operatore della facoltà di Economia che limita le entità visibili a quelle afferenti a tale facoltà. Prima di operare con un'applicazione può essere richiesto all'utente autenticato di scegliere quale identità usare; nell'esempio se entrare come amministratore o come operatore. Questa scelta è definita, usando un po' di insiemistica, scelta di un identità mentre tutte le possibilità tra cui può scegliere è l'insieme di tutte le identità. Come vedremo nel paragrafo successivo esiste un servizio applicativo che consente di ottenere l'insieme delle identità così come strumenti per ottenere una singola identità. Quest'ultima è rappresentata da un oggetto del tipo AutorizzazioneUtente:

```
1
public class AutorizzazioniUtente implements Serializable {
3
  protected String nome;
5
  protected Long[] idRuoli;
7
  protected Long[] idContesti;
9
  protected Long idAmbito;
11
  protected Long idEsternoSelezionato;
13
  private Long idRichiestaRegistrazione;
15
  transient protected Visibilita visibilita;
17
  protected String tipo;
19
  private Integer vincoloLettura;
21
}
```

Questa classe è pensata per funzionare come DTO, ovvero come un oggetto da trasferire anche attraverso la rete per specificare l'identità correntemente assunta dall'utente autenticato. Si veda il modulo plitvice workflow per un esempio di funzionamento di questa classe. Tra le proprietà spicca nome, che contiene il netID dell'utente. Necessarie le proprietà idRuoli, idContesti ed idAmbito che contengono rispettivamente i ruoli, i contesti e l'ambito scelti tra quelli presenti nell'insieme delle identità per l'utente autenticato. Per

una discussione delle proprietà visibilità, tipo e vincoloLettura si rimmanda al paragrafo Visibilità.

13.5 Il processo di autorizzazione

Definire l'interfaccia da utilizzare

13.6 Restrizione di visibilità

La classe AutorizzazioneUtente rappresenta, come si è visto, l'identità correntemente scelta dall'utente autenticato. Si tratta di un DTO che il client deve inviare magari attraverso la rete ad un servizio, ad esempio un manager che gestisca un workflow, per consentire a quest'ultimo di applicare correttamente le autorizzazioni.

```
1
public class AutorizzazioniUtente implements Serializable {
3
  ...
5
  transient protected Visibilita visibilita;
7
  protected String tipo;
9
  private Integer vincoloLettura;
11
}
```

Le tre proprietà elencate consentono al servizio (non al client) di implementare un meccanismo molto comodo per limitare la visibilità delle entità del modello mediante l'impiego dell'interfaccia Visibilita.

```
public interface Visibilita extends Serializable {
2
  void puoGestire(Serializable document);
4
}
```

Il client non dovrà impostare nessuna visibilità (la proprietà rimane a null) ma si deve limitare ad impostare il tipo che non è altro se non una stringa contenente un valore simbolico con un significato specifico per il client. Sarà invece il servizio che riceve l'AutorizzazioneUtente che dovrà provvedere a progettare ead istanziare un'implementazione di Visibilita sulla base della proprietà tipo impostato dal client. Esiste un meccanismo abbastanza comodo per istanziare dei tipi visibilità creando delle classi del tipo VisibilitaFactory:

```
1 public interface VisibilitaFactory {
```

```
3   public Visibilita create(AutorizzazioniUtente identita) throws
    TypeUnkonownException;

5 }
```

Tutte le factory create devono poi riunite in una catena del tipo VisibilitaFactoryChain mediante la seguente configurazione di Spring:

```
1     <bean id="visibilitaFactoryChain"
      class="it.kion.plitvice.autorizzazioni.visibilita.VisibilitaFactoryChain"
      >
        <property name="factories">
3         <list>
            <bean class="EsternoVisibiltaFactory"  />
5         </list>
        </property>
7     </bean>
```

La visibilitaFactoryChain così ottenuta è richiesta come prorpietà da tutti i servizi che aderiscono allo standard per il controllo delle autorizzazioni di Plitvice kit (ad esempio WorkflowManager).

Visto come il server provveda, una volta ricevuto il DTO, a creare una visibilità ed assegnarla allo stesso DTO, può essere utile capire come utilizzare una gerarchia di classi basate su Visibilita. Ogni servizio o gruppo di servizi può estendere l'interfaccia di base con dei metodi comuni per restringere le selezioni, ad esempio con dei metodi che prendano come parametro di input un ModelPattern. Ad esempio, nel caso di utenti esterni che debbano visualizzare solo le richieste della propria azienda sarebbe possibile creare una classe simile a questa:

```
1 public class AbstractVisibilita implements Visibilita {

3   public void limitaRichieste(RichiestaPattern p) {
      //non faccio nulla
5   }
        ...
7 }

9 public class VisibilitaEsterni extends AbstractVisibilita {

11  private Long idRichiesta;

13  @Override
    public void limitaRichieste(RichiestaPattern p) {
15    p.setId(idRichiestaRegistrazione);
    }
17  ...
  }
```

Dal momento che AutorizzazioniUtente è passato ad ogni servizio e contiene all'interno la visibilità dell'utente corrente è possibile utilizzare in modo conveniente il polimorfismo come nell'esempio seguente:

```
1 public void faiQualcosa(..., RichiestaPattern p, AutorizzazioniUtente i)
  {
```

```
3     AbstractVisibilita v = (AbstractVisibilita) i.getVisibilita();
      v.limitaRichieste(p);

5     ...
}
```

In questo modo il servizio può ignorare quale sia la classe concreta che implementa la visibilità e limitarsi a chiamare il metodo limitaRichieste() che non farà nulla tranne nel caso di un utente esterno.

13.7 Condivisione di scelte

Dove si spiega il meccanismo per condividere le scelte di un'utente (tra cui Identità) tra un'applicazione e l'altra.

Capitolo 14

Plitvice Workflow

14.1 Terminologia e convenzioni

Un workflow si compone di una serie di stati collegati l'uno a l'altro tramite passaggi detti transizioni. Il modo più utile per rappresentare un workflow è quello di utilizzare un grafo come quello seguente:

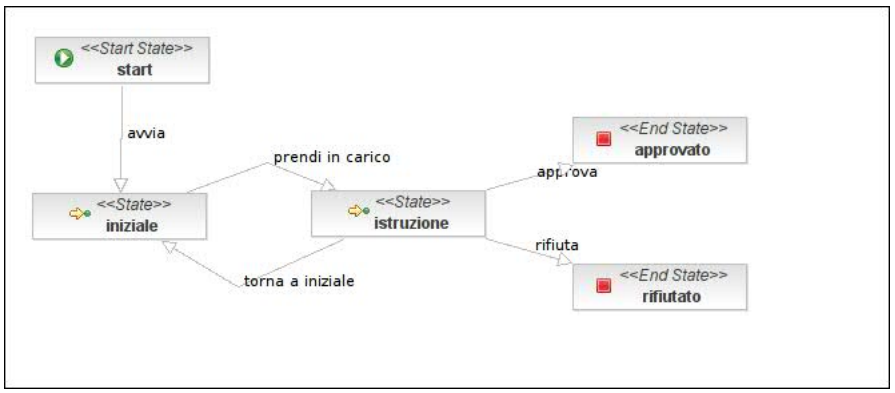

Figura 14.1: Grafico di un workflow

Nel flusso di esempio sono presenti cinque stati: *start*, *iniziale*, *istruzione*, *approvato* e *rifiutato*. Nella definizione di un nuovo flusso conviene rammentarsi di aderire ad alcune convenzioni stabilite da Plitvice kit: ad esempio conviene di utilizzare come nomi degli stati *a*) solo caratteri minuscoli e possibilemente *b*) preferire gli aggettivi ai sostantivi. Ogni workflow deve necessariamente *c*) iniziare con lo stato *start* che deve prevedere *d*) un'unica transizione denominata avvia. Tornando all'esempio le altre transizioni sono *prendi in carico*, *torna a iniziale*, *approva* e *rifiuta*; da notare la convenzione

di utilizzare *e*) dei verbi in modo imperativo per definire le transizioni. Infine alcuni stati sono detti terminali perché una volta raggiunti non è più possibile lasciarli (nell'esempio *approvato* e *rifiutato*).

14.2 Tabella di marcia

Le applicazioni Regola kit possono essere configurate per la gestione di workflow semplicemente aggiungendo una dipendenza al modulo *plitvice-workflow*. Predisporre invece un proprio gestore di flussi richiede diverse operazioni:

- definire un Workflow
- creare un WorkflowRepository
- definire un Ruolo con i relativi Diritti
- progettare una classe di Visibilità
- creare un AuthenticationManager
- progettare il documento
- progettare la classe WorkflowManager
- progettate la classe WorkflowActions

In effetti sembrano parecchie ma in realtà sono piuttosto veloci da approntare perché Plitvice kit fornisce tutte le classi di basi necessarie ad un'applicazione reale: ad esempio sono già previsti i supporti per i WorkflowRepository su database o in memoria tramite mock. Nel corso della presentazione verranno toccati tutti i punti dell'elenco e si provvederà a realizzare un esempio concreto per la gestione di un Workflow denominato *accordo* in grado di gestire un documento di tipo Accordo.

14.3 Workflow

Plitvice kit utilizza JBoos jBPM internamente per la gestione dei Workflow. La definizione di un Workflow avviene quindi mediante un file XML che raccoglie gli stati e le transizioni. Per chi utilizza Eclipse è disponibile un comodo plug in visuale che può essere di un qualche aiuto nella progettazione del workflow. Affinché la definizione sia fruibile da Plitvice kit è necessario che si attenga alle seguenti convenzioni:

- il file xml contenente la definizione deve essere salvato nella posizione src/main/resources/workflows/nomeflusso/processdefinition.xml
- il nodo iniziale di ogni flusso deve chiamarsi *start*

- dal nodo iniziale deve uscire solo una transizione denominata *avvia*
- ogni transizione che prevede una modifica di database deve essere associata ad un'azione gestita dalla classe PersistenceAction come nel frammento di definizione seguente:

```
<state name="iniziale">
    <transition to="istruzione" name="prendi_in_carico">
      <action
       class="it.kion.plitvice.workflow.service.impl.PersistenceAction"></action>
    </transition>
 </state>
```

- i nomi degli stati devono contenere solo caratteri minuscoli e possibilmente essere espressi medianti aggettivi (tipo iniziale, approvato, ecc.)
- i nome delle transizioni devono contenere solo caratteri minuscoli e possibilmente essere espressi mediante verbi nel modo imperativo (tipo accetta, rifiuta, ecc.)

14.4 WorkflowRepository

Un WorkflowRepository è un'entità di modello che consente di accedere a tutti i Workflow disponibili per l'applicazione in modo che i WorkflowManager possano, conoscendone solo il nome, accedere a tutte le proprietà di un flusso, compresa la definizione. Generalmente il repository si concretizza semplicemente in una tabella di database che elenchi i diversi workflow e ne raccolga gli attributi. Plitvice kit prevede una classe per gestire questa tabella, WorkflowRepositoryHibernate. Alternativamente, spesso impiegati per la fase di test, si può creare un repository in memoria popolato mediante configurazione di Spring (si veda ad esempio WorkflowRepositoryMock).

14.5 Autorizzazioni

L'autorizzazione su un workflow consiste nello stabilire se l'identità correntemente assunta dell'utente autenticato possa o meno segnalare una certa transizione oppure abbia i permessi per elencare i documenti in un certo stato. Lo schema delle autorizzazioni è quello definito nel modulo plitvice security che prevede il concetto di Ruolo a cui fa capo una collezione di diritti (Diritto); questi ultimi possono essere di natura applicativa oppure relativi ai workflow. Infine questi ultimi si distinguono per occuparsi degli stati oppure delle transizioni.

Gli elementi comuni ad ogni diritto sono un identificativo unico (un intero) ed un Ambito che rappresenta un partizionamento su base applicativa;

ad esempio esiste l'ambito dei tirocini che raccoglie tutti i diritti relativi alla grande area della gestione dei tirocini. Inoltre tutti i diritti fanno riferimento ad un'Entità, ovvero un elemento del modello con un ciclo di vita ampio secondo i dettami del Domain Driven Development. L'elenco delle Entità è persistito su database e a ciascuna è assegnato un identificativo univoco. I diritti relativo a workflow contengono invece sempre l'indicazione del flusso.[1]

I diritti di workflow di tipo transazione prevedono la possibilità di segnalare una specifica transazione. Il default, ovvero in caso di assenza di uno specifico diritto, la transazione non può essere segnalata da nessuno. I diritti di workflow di tipo stato consentono di elencare tutti i documenti che si trovino in quello stato. In assenza di un particolare diritto chiunque può richiedere l'elenco completo dei documenti per quel certo stato. Esiste comunque un meccanismo che consente anche ad identità che non avrebbero i diritti di accedere all'elenco dei documenti; si rimanda alla proprietà vincoloLettura di AutorizzazioneUtente e l'interfaccia Visibilità per dettagli in merito.

Le autorizzazioni sono abilitate o disabilitate tramite un aspetto di Spring; per attivarle basta aggiungere le seguenti righe in un file di configurazione di bean:

```
<!-- Enable @AspectJ support -->
<aop:aspectj-autoproxy proxy-target-class="false" />

<bean class="it.kion.plitvice.autorizzazioni.WorkflowManagerAuthentication"
>
   <property name="authentication" ref="authenticationManager" />
</bean>
```

14.6 Il documento

Un workflow documentale è destinato alla gestione di un documento, ad esempio potrebbe rappresentare i passi necessari e le persone (o gli uffici) coinvolti nel processo ufficiale di produzione di una pratica. In questo contesto il termine documento si riferisce ad una classe Java, in particolare un'entità di modello applicativo per la quale è eventualmente stato definito un Model Pattern di Regola kit. Nel corso di questo capitolo, come si è detto, presenteremo un esempio concreto il cui documento è l'entità Accordo:

```
package it.kion.plitvice.workflow.model;

```

[1] Attualmente dato un workflow discende una ed una sola entità per cui basterebbe l'indicazione del flusso per risalire inequivocabilmente all'entità; comunque al momento in cui questo manuale è redatto, sia l'id dell'entità che il nome del flusso devono specificate in modo congruente.

```
import it.kion.plitvice.workflow.service.impl.AccordoWorkflowActions;
import it.kion.plitvice.workflow.service.impl.AccordoWorkflowActions.Stati;

import java.io.Serializable;

public class Accordo implements Serializable {

  private Stati stato = Stati.start;
  private Long id = null;

  public void setStato(Stati stato) {
    this.stato = stato;
  }

  public Stati getStato() {
    return stato;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public Long getId() {
    return id;
  }

}
```

Durante la fase di progettazione del documento deve essere tenuta in considerazione la relazione forte che lega il documento al workflow che lo gestisce; in qualsiasi momento del ciclo di vita del documento deve esistere un modo (un algoritmo) che partendo da un'istanza del documento sappia determinare il workflow di riferimento unitamente allo stato in cui il workflow si trova. In altri termini lo stato del workflow deve essere desumibile attraverso il documento stesso, o perché lo contiene direttamente o perché contiene gli elementi per recuperare da una database lo stato del flusso. Nel nostro semplice esempio lo stato è del flusso è semplicemente una proprietà del documento.

Il WorkflowManager consentirà di effettuare delle selezioni dalla popolazione di tutti i documenti gestiti tramite flussi: sarà in particolare possibile ottenere l'elenco di tutti i documenti che si trovino in un certo stato oppure gestiti da un certo workflow. Per effettuare questo genere d'interrogazioni è stato esteso il meccanismo di ModelPattern con due annotazioni, @Stati e @Workflow.

```

package it.kion.plitvice.workflow.dao;

import it.kion.plitvice.workflow.model.Workflow;
import it.kion.plitvice.workflow.pattern.Stati;

import org.regola.model.ModelPattern;

public class AccordoPattern extends ModelPattern {
```

```
11    String[] inStati = new String[] {};
      private Workflow flusso;
13
      @Stati()
15    public String[] getInStati() {
        return inStati;
17    }

19    public void setInStati(String[] inStati) {
        this.inStati = inStati;
21    }

23    @it.kion.plitvice.workflow.pattern.Workflow
      public Workflow getFlusso() {
25      return flusso;
      }
27
      public void setFlusso(Workflow flusso) {
29      this.flusso = flusso;
      }
31
}
```

Tanto per avere un'idea fin da subito del funzionamento del ModelPattern nel contesto dei workflow può risultare utile l'esempio seguente che richiede al gestore del flusso tutti i documenti che si trovino nello stato *iniziale* mediante una chiamata al metodo elencoProcessiInStati.

```
AccordoPattern pattern = new AccordoPattern();
2 pattern.setInStati(new String[] { "iniziale" });

4 // recupero il documento gestito dal flusso
List<Accordo> accordi =
6  accordoManager.elencoProcessiInStati(pattern, autNicola);
```

Come si vede prima si imposta AccordoPattern con lo stato d'interesse (iniziale) e poi lo si passa al metodo elencoProcessiInStati

14.7 WorkflowManager: funzionamento

Nel livello di servizio, assieme agli altri Business Delegate, i Service Facade ed in generale tutte le classi che consentono di fruire del modello, trovano collocazione i WorkflowManager; si tratta di Manager nel senso indicato da Regola kit che offrono un'interfaccia comune per la gestione dei flussi di lavoro.

```
public interface WorkflowManager
2 <D extends Serializable, ID extends Serializable>
extends GenericManager<D,ID> {
4
  List<D> elencoProcessiInStati(ModelPattern pattern, AutorizzazioniUtente
  utente);
6 int countProcessiInStati(ModelPattern pattern, AutorizzazioniUtente
  utente);
```

```
8   String segnala(D processo, String transizione,AutorizzazioniUtente
    utenti) throws ProcessoNonValidoException, TransizioneNonValidaException;

10  String avviaFlusso(Workflow flusso,Object
    documentoIniziale,AutorizzazioniUtente utenti);
    String avviaFlusso(Object documentoIniziale,AutorizzazioniUtente utenti);
12
    List<String> transizioniDisponibili (D processo, AutorizzazioniUtente
    utente);
14
    public List<String> elencoStati(D processo);
16

18  Workflow[] elencoFlussiGestiti();

20 }
```

Ogni WorkflowManager si occupa di un solo tipo di documento D (un entità serializzabile e persistibile su database) la cui chiave d'identificazione è del tipo ID. Ricordo che il tipo D (il documento) può essere associato ad uno o più workflow tutti gestiti dallo stesso WorkflowManager; per sapere quali sono i flussi di lavoro gestiti da un WorkflowManager bisogna invocare il metodo elencoFlussiGestiti(). Non bisogna dimenticare che se la classe D del documento può essere gestita da diversi flussi la singola istanza è gestita solo da uno ed un solo flusso; ovvero se un'istanza inizia il suo percorso all'interno di un workflow non può abbandonarlo più. Da questo deriva che data istanza di D è possibile determinare in modo univoco il flusso che la gestisce ed in questo modo è utilizzata all'interno degli altri metodi di WorkflowManager. Un'altra considerazione generale riguarda il sistema di autorizzazione che verifica le credenziali dell'utente collegato prima di consentire l'elencazione di documenti in un certo stato oppure la segnalazione di una transizione; molti metodi di WorkflowManager richiedono il passaggio di un DTO chiamato AutorizzazioniUtente che consente di individuare con esattezza l'identità selezionata dall'utente corrente ed applicare i criteri di autorizzazione. Per maggiori dettagli si rimanda al modulo plitvice security.

Il primo metodo da invocare per avviare un nuovo servizio è avviaFlusso() al quale si può eventualmente passare un oggetto preliminare che contenga le informazioni necessarie per creare il documento da inserire nel flusso. Naturalmente documento e oggetto preliminare possono coincidere se lo si ritiene conveniente. Come si è visto data un'istanza di documento è possibile risalire al flusso; nel momento che precede la creazione di un nuovo flusso però quell'istanza non esiste per cui è necessario specificare al metodo avviaFlusso() quale workflow utilizzare tra quelli gestiti da uno specifico WorkflowManager. Se si omette questa indicazione sarà utilizzato il primo workflow tra quelli gestiti.

Una volta avviato un flusso è possibile elencare tutti i documenti che si trovino in un certo stato tramite il metodo elencoProcessiInStati() passandogli come parametro un ModelPattern come indicato nel paragrafo precedente.

Avuto un documento in un certo stato è possibile sapere quali siano le transizioni che gli sono consentite dall'identità correntemente assunta tramite il metodo transizioniDisponibili().

Infine per segnalare una transizione (cioè cambiare stato) si richiami il metodo segnala().

14.8 WorkflowManager: progettazione

La creazione di un servizio di workflow si limita alla creazione di un'interfaccia che estenda WorkflowManager e nel darne un'implementazione di massima.

```
2  public interface AccordoWorkflowManager
    extends WorkflowManager<Accordo, Long> {
4  }

6  public class AccordoWorkflowManagerImpl
   extends WorkflowManagerImpl<Accordo, Long>
8  implements AccordoWorkflowManager
   {
10
     public AccordoWorkflowManagerImpl(GenericDao<Accordo, Long> dao) {
12     super(dao);
       setFlussiGestiti(new String[] { "accordo" });
14   }

16   @Override
     protected void addProcessVariable(ProcessInstance instance) {
18     // TODO Auto-generated method stub
     }
20
     @Override
22   public Workflow flowForDocument(Accordo documento) {
       return repository.findByName(flussiGestiti[0]);
24   }

26 }
```

I metodi di cui occuparsi nell'implementazione sono il costruttore in cui specificare i workflow gestiti da questo WorkflowManager, nell'esempio solo uno di nome *accordo* che per quando indicato nelle convenzioni relativi alle definizioni dei workflow deve trovarsi nella posizione src/main/resources/workflows/accordo/processdefinition.xml. Il metodo flowForDocument() consente di individuare a quale flusso sia associato l'istanza del documento; nel nostro esempio ogni istanza è gestita da un solo workflow. Infine addProcessVariable() consente di inserire delle variabili da utilizzare all'interno delle

action durante le segnalazioni. Nell'esempio non è impiegato questo metodo. Il file di definizione di Spring per le classi appena create deve assomigliare al seguente frammento:

```
<bean id="accordoActions"
class="it.kion.plitvice.workflow.service.impl.AccordoWorkflowActions" >
  <property name="dao" ref="accordoDao" />
</bean>

<bean id="accordoManager"
class="it.kion.plitvice.workflow.service.impl.AccordoWorkflowManagerImpl" >
   <constructor-arg>
      <ref bean="accordoDao"/>
    </constructor-arg>
        <property name="actions" ref="accordoActions" />
    <property name="repository" ref="workflowRepositoryMock" />
    <property name="visibilitaFactoryChain" ref="visibilitaFactoryChain" />
</bean>
```

Come si può osservare le dipendenze di ogni WorkflowManager sono la classe di actions che si occupa come si vedrà di definire le logiche di persistenza, un elemento di modello chiamato repository che si occupa di fornire l'accesso a tutte le difinizioni dei workflow disponibili per l'applicazione ed infine una visibilitaFactoryChain che è intimamente legata con i criteri di applicazione delle autorizzazioni per cui si rimanda alla documentazione del modulo plitvice security per i dettagli.

14.9 WorkflowActions

La persistenza del workflow (e del documento a cui si riferisce) può essere completamente personalizzata tramite la progettazione di una classe che estenda WorkflowActions; i metodi di questa classe saranno richiamati (internamente da WorkflowManager) al momento di attraversare una transizione o quando è richiesto un elenco di documenti. Molto probabilmente la vostra classe di action avrà un dao di Regola kit tra le dipendenze e lo userà nei modi consueti finendo per assomigliare ad una versione complessa di quella presentata qui di seguito:

```
public class AccordoWorkflowActions
   extends WorkflowActions<Accordo> {

  public enum Stati {
    start, iniziale, istruzione, approvato, rifiutato
  };

  private AccordoDao dao;

  public List<Accordo> elencoProcessiInStati(Workflow flusso, String []
  stati, ModelPattern p, AutorizzazioniUtente utente)
  {
    log.info("elencoProcessiInStati=");
    return getDao().find(p);
```

```
14  }

16  public int countProcessiInStati(Workflow flusso, String [] stati,
    ModelPattern p, AutorizzazioniUtente utente)
    {
18    log.info("countProcessiInStati=");
      return getDao().count(p);
20  }

22    @Override
    @StatoCorrente()
24  public String getStatoCorrente(Accordo documento)
    {
26    return documento.getStato().toString();
    }
28
    @Override
30  public String avviaFlusso(Object documentoIniziale, AutorizzazioniUtente
    utente) {
      log.info("avviaFlusso");
32
      Accordo accordo = new Accordo();
34
      accordo.setId(1l);
36    accordo.setStato(Stati.iniziale);

38    dao.save(accordo);

40    return Stati.iniziale.toString();
    }
42
    @Transizione("prendi_in_carico")
44  public void segnalaPrendiInCarico(Accordo documento, String transizione,
    AutorizzazioniUtente utente) {
      documento.setStato(Stati.istruzione);
46    dao.save(documento);
    }
48
    @Transizione("torna_a_iniziale")
50  public void segnalaTornaAIniziale(Accordo documento, String transizione,
    AutorizzazioniUtente utente) {
      documento.setStato(Stati.iniziale);
52    dao.save(documento);
    }
54
    @Transizione("approva")
56  public void segnalaApprova(Accordo documento, String transizione,
    AutorizzazioniUtente utente) {
      documento.setStato(Stati.approvato);
58    dao.save(documento);
    }
60
    @Transizione("rifiuta")
62  public void segnalaRifiuta(Accordo documento, String transizione,
    AutorizzazioniUtente utente) {
      documento.setStato(Stati.rifiutato);
64    dao.save(documento);
    }
66
    public void setDao(AccordoDao dao) {
68    this.dao = dao;
    }
```

```
70
  public AccordoDao getDao() {
72   return dao;
  }
74
}
```

Nel progettare la classe action bisogna implementare dei metodi la cui firma è nota (ProcessiInStati, countProcessiInStati, getStatoCorrente() e avviaFlusso()) ed aggiungere anche dei metodi (dal nome libero e la firma predeterminata) da associare ad ogni transizione gestita. ElencoProcessiInStati e countProcessiInStati forniscono la base materiale per l'estrazione dal database dei documenti nello stato richiesto. Il metodo getStatoCorrente() che consente di determinare, dato un documento, il suo stato attuale. Infine avviaFlusso() provvede a creare un flusso nuovo dato un oggetto preliminare. Inoltre devono essere creati tanti metodi quante sono le transizioni che abbisognano di persistenza da marcare con l'annotazione @Transizione ed il nome della transazione. Il sistema provvederà a chiamare automaticamente questi metodi al momento della segnalazione, subito dopo che lo stato logico del flusso è stato aggiornato.

Bibliografia

[i18] Resource Bundles. http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/.

[mav] Maven2. http://maven.apache.org/.

[spr] Spring Framework. http://www.springframework.com.

Indice analitico