Web Application Engineering
Object Relational Mapping

cristian lucchesi
IIT-CNR

Pescara, 15-16 Maggio 2007
Alei – Ud'A

ORM

[OR Mapping - Object-Relational mapping is the process of the transformation of the data between the class objects and databases. Applications can depend on an OR-M like tool that greatly simplifies this work instead of manually coding the transformation process.]

Wikipedia

Introduction to the Java Persistence API

- Java Persistence API (JPA) fornisce POJO (Plain Old Java Object) standard e object relational mapping (OR mapping) per dati persistenti.
- la persistenza riguarda:
  - il salvataggio dei dati
  - la loro consultazione
  - la loro gestione (aggiornamento, cancellazione)
- i dati adesso possono essere gestiti tramite JPA a partire da EJB 3.0 come risultato della JSR 220

Persistence Entities

- con Persistent Data normalmente ci si riferisce a dati permanenti in una applicazione
- lo stato dei dati è reso permanente salvandolo in uno storage come un Database, un filesystem, una flash memory, ...
- in JPA ci si riferisce ai dati persistenti come **Entity**
- per esempio *il titolo*, *l'abstract*, *l'articolo*, *la data* di una entry di un blog sono dati tipicamente permanenti e possono essere raggruppati in una entity **BlogEntry**

Pojo

```
public class BlogEntry {
   private String id;
   private String title;
   private String excerpt;
   private String body;
   private Date date;
   ...
// Getters e Setters vanno qui }
```

- in termini JPA una entità è un oggetto persistente che può essere salvato e consultato da uno storage
- in termini tecnici, questa entity corrisponde ad una classe Java
- a classe corrisponde ad un oggetto BlogEntry con attributi: it, titolo, excerpt, ...

Entity

```
@Entity
public class BlogEntry {

  private String id;
  private String title;
  private String excerpt;
  private String body;
  private Date date;
  ...

}
```

- per rendere persistente la classe java utilizzando JPA è necessario qualificarla come una entity utilizzando l'annotazione **@Entity**
- l'annotazione dice al motore della persistenza che gli oggetti creati da quella classe sono supportati dalla JPA

@Table

```
@Entity
@Table(name="blog_entries")
public class BlogEntry {

  ...

}
```

- **@Table** è utilizzato a livello della classe
- permette di specificare i nomi delle tabella, del catalogo, dello schema relativi al mapping
- se l'annotazione @Table non è presente il default (in Hibernate) è il nome non qualificato della classe

@Column

```
@Column(name="id",
        nullable=false)
private String id;

@Column(name="title",
        nullable=false,
        length=70)
private String title;

@Column(name="excerpt",
        nullable=false,
        length=200)
private String excerpt;

...
```

- **@Column** è utilizzato a livello di proprietà o relativo getter
- permette di specificare gli attributi della colonna su cui la proprietà è mappata
- se l'annotazione @Column non è presente nome della colonna (in Hibernate) è uguale al nome della proprietà
- altri attributi impostabili sulla colonna sono: unique, insertable, updatable, precision, scale

@Id

```
@Id private
String id;
```

- una entity deve essere sempre identificabile univocamente
- si può utilizzare un tipo primitivo od un oggetto apposito come chiave univoca
- il method equals verrà chiamato dalla JPA per confrontare l'uguaglianza di due chiavi
- l'annotazione @Id su un campo (o sul suo setter) marca un campo come chiave

BlogEntry Entity

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class BlogEntry {
  @Id @Column(name="id", nullable=false)
  private String id;

  @Column(name="title", nullable=false, length=70)
  private String title;

  @Column(name="excerpt", nullable=false, length=200)
  private String excerpt;

  @Column(name="body", nullable=false, length=1400)
  private String body;

  @Column(name="date", nullable=false)
  private Date date;

  //Getter, setter e altri metodi di seguito...
```

EntityManager

- questa classe segue il pattern **Manager** per gestire le entità
- gestire una o più entità si riferisce all'azione di mantenere un insieme di oggetti Java sotto il controllo dell'EntityManager
- fino a quando le entità non hanno un'associazione con l'EntityManager esse sono solamente normali oggetti java anche se marcati con l'annotazione @Entity

EntityManager - J2SE

- l'EntityManager fornisce una API per rendere persistenti le entità, rimuoverle, aggiornarle, interrogarle e cancellarle
- Nelle applicazioni J2SE, un riferimento all'EntityManager può essere ottenuto tramite un Factory

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("PersistentUnitName");

EntityManager eManager =
    entityManagerFactory.createEntityManager();
```

- L'EntityManagerFactory può essere configurato con l'aiuto della Persistent Unit di cui parleremo più avanti.

EntityManager - J2EE

```
import javax.persistence.EntityManager;

...

@EntityManager
private EntityManager entityManager;
```

- nelle applicazioni J2EE il container inietterà direttamente un riferimento all'entityManager utilizzando la dependency injection

EntityManager.persist()

```
...

BlogEntry blogEntry = new BlogEntry();
blogEntry.setId("waeSeminar")
blogEntry.setTitle("Seminario di WAE")

// Update delle varie proprietà
entityManager.persist(blogEntry);

...
```

- gli oggetti annotati come entità sono regolari oggetti java fino a quando non vengono resi persistenti dall'EntityManager
- il metodo *persist(entityObject)* rende persistente l'entità nel database (esegue la INSERT SQL)
- quando la persist viene invocato viene controllato che non ci siano oggetti con lo stesso id nel database, altrimenti viene sollevata una run-time exception: **EntityExistsException**

EntityManager.find()

```
BlogEntry entry =
    eManager.find(BlogEntry.class,
                  "waeSeminar");

If (entry != null){
    // entry object may or may not be null.
    // Process the object.
}
```

- il metodo find è utilizzabile per interrogare gli entity object
- il metodo **find()** accetta due paremetri
  - l'oggetto Entity
  - il valore della chiave primaria
- se l'oggetto richiesto non può essere trovato, l'entityManager restituisce null

- l'oggetto restituito dall'EntityManager diventa così direttamente utilizzabile

EntityManager.getReference()

```
BlogEntry entry =
    eManager.getReference(BlogEntry.class,
                          "waeSeminar");

// entry object may not contain the actual
// state values for title, excerpt,

// body and date, the states may be loaded
// during the first access.

String title = entry.getTitle();

// The persistence engine may fetch the title
// value for the entry here
// at this particular point of time. ...
```

- accetta gli stessi parametri del metodo find
- se l'oggetto non è trovato questo metodo restituisce un'eccezione **EntityNotFFoundException**
- l'istanza è prelevata in modo lazely (non si prelavano i valori delle proprietà dal database)
- lo stato dell'entità (title, excerpt, body...) sono prelevati la prima volta che si accede all'oggetto

EntityManager.remove()

```
BlogEntry entry =
    eManager.getReference(BlogEntry.class,
                                "waeSeminar");

entityManager.remove(entry)
```

- per cancellare un oggetto dal database è possibile utilizzato la chiamata **EntityManager.remove(entityObject)**
- l'entityObject passato deve essere gestito dall'entityManager altrimenti la rimozione fallisce
- l'operazione di cancellazione dal db può avvenire successivamente (dopo la flush())
- dopo la chiamata, l'entity diventerà "detached" dal entityManager e non più da lui gestita

Flushing and Refreshing

```
BlogEntry entry = ....
    entry.setExcerpt("Seminario ...");

// Aggiorna lo stato dello oggetto ....
entityManager.flush();

// Sincronizza il database con i valori
// presi dall'oggetto "entry"
BlogEntry entry = ...
    entry.setBody("Il web...");

// Lo stato dell'oggetto "entry" è aggiornato ...
entityManager.refresh();

// l'entity object viene aggiornato
// con i valori presi dal db
// Tutti i cambiamenti fatti sono persi
```

- il metodo entityManager.flush() sincronizza tutti i cambiamenti delle entity sul db
- il metodo entityManager.refresh() al contrario preleva le informazioni dal database e aggiorna le entity, eventuali nuovi valori impostati sull'oggetto vengono persi.

Query API

- uno degli svantaggi dei metodi find() e getReference(), è che si possono effettuare interrogazioni solo per chiave primaria
- inoltre il nome della classe deve essere noto a priori
- le Query API della JPA sono molto più potenti e molti più criteri possono essere specificati a runtime
- l'EntityManager viene utilizzato come factory per ottenere un riferimento ad un oggetto **Query**
- il linguaggio per le stringhe utilizzate è chiamato JPQL, è molto simile all'SQL ma più object-oriented, robusto e flessibile

Static Query

```
@NamedQuery(
   name="BlogEntry.findAll“,
   query="SELECT be FROM BlogEntry be")
@Entity class BlogEntry {
   ....
}
```

- una static query (o named query) è una query definita staticamente prima dell'entity class
- gli viene assegnato un nome in modo che sia rintracciabile dagli altri componenti che vogliono utilizzarla

```
Query findAllQuery =
   eManager.createNamedQuery(
   "BlogEntry.findAll“);

/ / Execute the query.
```

- una "named query" così definita può essere utilizzata successivamente

Dynamic Queries

```
String queryString = ...
// Obtained during run-time.

Query dynaQuery =
   eManager.createQuery(queryString);
```

- dynamic queries sono quelle in cui la query string viene fornita a runtime
- per crearle si utilizza *entityManager. createQuery(queryString)*
- sono meno efficienti delle named query perché sia il parsing che la validazione della queryString che la trasformazione da JPQL a SQL sono fatte a runtime

Single result

```
Query query =
   eManager.createQuery(
     "SELECT be FROM BlogEntry be " +
     "WHERE be.title = " +
     "'Web Application Engineering'");

BlogEntry entry =
   query.getSingleResult();
```

- la query string definita dentro la createQuery viene trasformata in sql
- nella clausola FROM invece del nome della tabella di usa il nome dell'entity
- **"WHERE be.title = ..."** indica che l'attributo title deve avere il valore specificato
- la **getSingleResult()** esegue la query e restituisce una singola riga di risultato (un solo oggetto)
- se non sono presenti entity che corrispondono alla query viene sollevata una eccezione **EntityNotFoundException**
- se ci sono più righe che corrispondono viene sollevata una **NotUniqueResultException**

Multiple Results

- **query.getResultList()** esegue la query e restituisce una lista di istanze di entity (oppure una lista vuota)

```
Query query =
   eManager.createQuery(
      "SELECT be FROM BlogEntry");

List<BlogEntry> entries =
   (List<BlogEntry>) query.getResultList();
```

- il type-cast (List<BlogEntry>) è necessario perché viene restituita una lista non parametrizzata di Object
- se solamente una BlogEntry corrisponde ai criteri della query viene restituita una lista di dimensione 1
- *getResultList()* può eseguire solo operazioni di **SELECT**, se si utilizza statement di UPDATE o DELETE viene sollevata una *IllegalStateException* a run-time

Lavorare con i parametri

- per riutilizzare ed eseguire le query effecientemente con differenti insiemi di valori in JPA è possibile utilizzare il supporto ai parametri:
  - **posizionali**
  - **nominali**

```
String selectQuery =
    "SELECT be FROM BlogEntry " +
   "WHERE be.title = ?1 and be.excerpt = ?2";

Query selectQuery =
    eManager.createQuery(selectQuery);
...
selectQuery.setParameter(1, title);
selectQuery.setParameter(2, excerpt);
```

- i posizionali sono utilizzati per sostituire il valore in funzione del indice della posizione del parametro e sono marcati con ?index
- similmente i nominali sono utilizzati per sostituire il valore ad uno specifico termine marcato con :name

- durante l'esecuzione della query ?1 e ?2 saranno rimpiazzati dai valori specificati dalle stringhe title e exceprt

Parametri con nome

```
String query =
   "SELECT be FROM BlogEntry " +
   "WHERE be.title = :title " +
   " AND be.date = :fromDate“;

Query selectQuery =
   eManager.createQuery(query);
...
selectQuery.setParameter("title", title);
selectQuery.setParameter("fromDate", myDate);
```

- al parametro può essere dato un nome significativo prefissato da :
- il metodo *createQuery()* si occupa anche di trasformare le date nel corretto formato per l'SQL

Named parameter e @NamedQuery

```
@NamedQuery(
   name ="BlogEntry.findByTitle",
   query = "SELECT be FROM BlogEntry " +
           " WHERE be.title = :title")
...
//e successivamente nel codice...
Query namedQuery =
   e.createNamedQuery("BlogEntry.findByTitle");

namedQuery.setParameter("title", titleValue);
```

- i parametri posizionali e quelli con nome possono essere utilizzati anche nelle static query

Paginare i risultati

```
private static int maxRecords = 25;
...
List<BlogEntry> getEntries(int startPosition) {
    String queryString =
       "SELECT be FROM BlogEntry";

    Query query =
       eManager.createQuery(queryString);

    query.setMaxResults(maxRecords);
    query.setFirstResult(startPosition);

   return eManager.getResultList(queryString);
}
```

- nel caso una query restituisca molti risultati non è una buona idea mostrarle tutti insieme all'utente, potrebbe rallentare molto la visualizzazione
- mostrare i risultati suddivisi in pagine (come in Google) risolve il problema
- l'applicazione si può occupare di chiamare il metodo **getEntries()** passando la posizione iniziale (*startPosition*) scelta dall'utente

Configurare la persistenza

```
// Nel caso di J2SE
String puName ="MyPersistentUnit";

EntityManagerFactory factory =
Persistence.createEntityManagerFactory(puName);

EntityManager entityManager =
    factory.createEntityManager();
```

```
// Nel caso J2EE
@PersistenceContext(
    unitName="MyPersistentUnit")
private EntityManager entityManager;
```

- ogni applicazione che vuole utilizzare la JPA deve specificare un file **persistence.xml** nella directory **META-INF**
- il file persistence.xml configura **l'EntityManagerFactory** e di conseguenza

  l'EntityManager

Persistence.xml

- Esempio di file persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
  <persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="entityManager">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/blogDatasource</jta-data-source>
      <properties>
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        <property name="jboss.entity.manager.factory.jndi.name"
                  value="java:/blogEntityManagerFactory"/>
        <property name="hibernate.jdbc.charSet" value="utf-8"/>
      </properties>
  </persistence-unit>
</persistence>
```


Java DataSource

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
   <local-tx-datasource>
      <jndi-name>
         blogDatasource
      </jndi-name>

      <connection-url>
         jdbc:hsqldb:.
      </connection-url>

      <driver-class>
        org.hsqldb.jdbcDriver
      </driver-class>

      <user-name>sa</user-name>

      <password></password>

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

- i file datasource permettono di configurare l'accesso ad una sorgente di dati, tipicamente un database
- sono reperibili tramite un nome specificato con l'elemento **jndi-name**

Molteplicità nelle relazioni

- ci sono quattro tipi di molteplicità
- One-to-one: ogni istanza dell'entity è correlata ad una singola istanza di un'altra entity
- One-to-many: una singola istanza di un'entity può essere legata a molte istanze di un'altra entity
- Many-to-one: molte istanze di un'entity sono correlate ad una singola istanza di un'altra entity
- Many-to-many: le istanze di un'entity possono essre correlate a molte istanze di un'altra entity

@ManyToOne: esempio

```
@Entity
@Table(name = "bookings",
       schema = "public")
public class Booking {
  ...
  private Customer customer;
  ...
  @ManyToOne
  @JoinColumn(name = "customer_id",
              nullable = false)
  @NotNull
  public Customer getCustomer() {
    return this.customers;
  }
  ....
}
```

- molte istanze di un oggetto **Booking** possono essere associate ad un oggetto **Customer** (UML)

Relazioni inverse

```
@Entity
@Table(name = "customers",
        schema = "public")
public class Customer {
  ...
  private List<Booking> bookings;
  ...
  @OneToMany(mappedBy = "customer")
  public List<Booking> getBookings() {
    return this.bookings;
  }
  ...
}
```

- un **Customer** può avere associati molti **Booking**
- l'attributo *mappedBy* dell'annotazione *@OneToMany* specifica qualle attributo della classe **Booking** è utilizzato per definire la relazione

aiuto!

- qualcuno mi aiuta con tutte queste annotazioni???
- Hibernate tool può generare le entity per noi a partire da un database....

Riferimenti

- Introduction to Java Persistence API(JPA):
  http://www.javabeat.net/javabeat/ejb3/articles/2007/04/introduction_to_java_persistence_api_jpa_ejb_3_0_1
- The Java EE 5 Tutorial:
  http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

grazie per l'attenzione

cristian.lucchesi@iit.cnr.it