Enterprise JavaBeans 3.0
Entity Beans

Luca Cavallaro
cavallaro@elet.polimi.it
credit to: Matteo Miraz
miraz@elet.polimi.it

20/11/2007

Outline Entity Beans

- Intro
- Entity structure
- EntityManager
- *EJB Query Language*

Entity beans

- *An instance of an “entity bean” object is a component that:*
  - *Represents domain elements (relational data) as objects*
  - *Has a long life cycle: as the tuple in the DB*

Entity & POJO

- A DB entity is associated to a POJO
  - Plain Old Java Object
  - Entity instances are POJO till an EntityManager creates a DB entry

- An entity instance can be “attached” or “detached”, depending on whether it is manged or not by an EntityManager

Persistence.xml

```
<persistence>
  <persistence-unit name="titan">
    <jta-data-source>java:/OracleDS
    </jta-data-source>
    <properties>
      <property name="org.hibernate.hbm2ddl">
       update
      </property>
    </properties>
  </persistence-unit>
</persistence>
```

Persistency unit

- EM manage a fixed set of entity beans (Persistence unit), saving data on a DB
- File *META-INF/persistence.xml*
  - For each persistence unit :
    - Name *<persistence-unit name="iwbank">*
    - Database *<jta-data-source>java://DefaultDS</…>*
    - Transaction type
      <transaction-type>JTA/RESOURCE_LOCAL</…>
    - Entity classes *<class>com.test.EntityOne</class>*
    - Flags
      <exclude-unlisted-classes/>
    - Vendor-specific properties
      <properties>…</properties>

How we can get an Entity Manager

- EntityManagerFactory
  - EntityManager createEntityManager()
  - EntityManager createEntityManager(Map properties)
  - void close()
  - boolean isOpen()
- How we can get an EntityManagerFactory
  - Java SE: use javax.persistence.Persistence EntityManagerFactory createEMFactory(String name);
  - Java EE: use @PersistenceUnit(unitName="name")
- In a session bean you can get PC using annotation @PersistenceContext(unitName="name")

Transactions

- In Java EE are managed by the container
- In Java SE you need manual managment
  - *EntityManager.getTransaction()* returns a transaction, on which you can call *begin*, *commit*, *abort* and *isAlive*

```
Cabin c = new Cabin(); c.set…
EntityTransaction t = em.getTransaction();
t.begin();
em.persist(c);
t.commit();
```

Entity Manager: persist

- ***After creating an entity and setting its properties it is possible to "persist" it***
  - A database entry is created(when it is created it depends on the transaction scope)
  - Auto generated keys are created
  - Entity turns MANAGED

Entity Manager: find an entity

- Primary key search
  - entity class and primary key have to be specified
  - find e getReference methods. If no entity found:
    - find: returns null
    - getReference: throws EntityNotFoundException
  - Returned entity is MANAGED
- Using a query
  - Named o unnamed
  - SQL o EJB QL

Managing DETACHED entities

- Entities, as Java classes, can be managed by a client, but they are detached by a persistence manager
- Il client can modify parameters and return the entity to server
- DB state can be updated using merge

    Cabin managed = em.merge(unmanaged);

Entity deletion

- To remove an entity from the DB you can use remove:

  em.remove(cabin);

  – Object becomes DETACHED, no more present in the DB

Flush

- Container decides when flushing effects on DB
  - All changes are executed before a related query
- Synchronization can be forced using flush()
- Flush can be forced when transaction commits using:
  setFlushMode(FlushModeType.COMMIT)
  - Watch out for cosistency!

Entity Beans Structure

Entity beans

- Usually model named concepts
- Describe both state and behavior
- Model persistent data base entries
- POJO with annotations
- Operation on entity beans are written on the DB
  - You have to interact with an EntityManager

A first entity bean

Represent a customer, using name, surname.

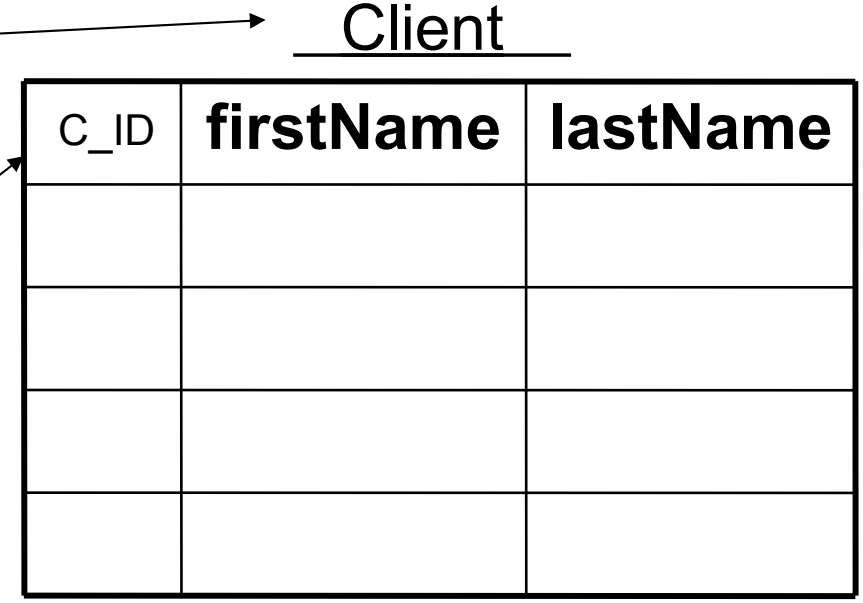
```
package com.titan.domain;

import javax.persistence

@Entity
public class Customer implements java.io.Serializable {

   @Id
    private long id;

    private String firs

    private String lastName;

    /* getters e setter */
}
```

- This class represents data that can be stored in a db
- Has a “name” used in EJB QL queries (default: short name)

- Primary key
- Can be put also on getters and setters
- Watch out for placement!

Mapping with DB

```
package com.titan.domain;

@Entity
@Table(name="Client")
public class Customer {

  @Id
  @Column(name="C_ID")
  private long id;

  private String firstName;

  private String lastName;

  /* getters e setter */
}
```



Mapping with DB

@Table:

- name: String
- catalog: String
- schema: String

@Column:

- name: String
- unique: Boolean (false)
- nullable: Boolean (true)
- insertable: Boolean (true)
- updatable: Boolean (true)
- columnDefinition: String
- table: String

Chiave primaria semplice

```
@Entity
@SequenceGenerator(name="CUSTOMER_SEQUENCE",
   sequenceName="CUST_SEQ")
public class Customer {

   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
                   generator="CUSTOMER_SEQUENCE")

   private long id;

   private String firstName;

   private String lastName;
   /* getters and setter */
}
```

Oracle sequence

States that we want the DB to generate a key

We want a sequence

Embeddable objects

```
package com.titan.domain;

import javax.persistence.*;

@Embeddable
public class Address implements java.io.Serializable {
    private String street;
    private String city;
    private String state;

   /* getters and setters */

 }
```

Embedded objects

```
package com.titan.domain;

import javax.persistence.*;

@Entity
public class Customer implements java.io.Serializable {

  @Id
   private long id;

   private String firstName;
   private String lastName;

  @Embedded
   private Address address;


   /* getters e setter */
}
```

Multiple primary keys

- Not trivial with EJB3.0

- Java Persistence allows two different ways:
  - `@javax.persistence.IdClass`
  - `@javax.persistence.EmbeddedId`

@IdClass

- How to:
  - Create a Java class representing multiple key
  - Must be a Serializable class
  - Must have default constructor
  - Should define `Equals hashCode`

```
public class CustomerPK implements Serializable {
  private String lastName, firstName;
  /* getter, setter, default constructor, equals hashCode */
}
```

Annotate l'entity:

```
@Entity
@IdClass(CustomerPK.class)
public class Customer implements Serializable {
 @Id private String lastName;
 @Id private String firstName;
 private int age;

  /* getters and setters */
}
```

@EmbeddedId

- Define an embeddable class to define a primary key:
  - Must be a Serializable class
  - Must have default constructor
  - Should define `Equals hashCode`

```
@Embeddable
public class CustomerPK implements Serializable {
  private String lastName, firstName;
  /* getter, setter, costruttore di default, equals e hashCode */
}
```

Annotare l'entity:

```
@Entity
public class Customer implements Serializable {
 @EmbeddedId private CustomerPK pk;

 private int age;

  /* getters setters */
}
```

Relations between entities

- Entities can be linked by relations
  - EG represenst ships, with their cabins
- Kinds of relations:
  - one to one (uni and bi-directional)
  - one to many (uni and bi-directional)
  - Many to one (unidirectional)
  - Many to many (uni and bi-directional)

One to one unidirectional relations

Eg: a client and his address

- We want to get a client's address...
- but not an address's clients

```
@Entity
public class Customer {
  …
@OneToOne
@JoinColumn(name=”ADDRESS_
    ID”)
  private Address addr;
  …
}
```

```
@Entity
public class Address {
 …
}
```

One to one bidirectional relations

Eg: a customer and his CC

```
@Entity
public class Customer {
  …
@OneToOne
@JoinColumn(name="ADDRESS_
    ID")
  private CreditCard card;
  …
}
```

```
@Entity
public class CreditCard {
 …
 @OneToOne(mappedBy="card")
  private Customer cust;
  …
}
```

Warning!!! a customer is the owner of this relation!

One to many unidirectional relations

Eg: a client and his phone number

```
@Entity
public class Customer {
  …
@OneToMany
@JoinColumn(name=”CUSTOME
    R_ID”)
  private Collection<Telephone> p;
  …
}
```

```
@Entity
public class Telephone {
 …
}
```

many to one unidirectional relations

Eg: a cruise and a ship

```
@Entity
public class Cruise {
  …
@ManyToOne
  private Ship ship;
  …
}
```

```
@Entity
public class Ship {

 …
}
```

One to many bidirectional relations

Eg: a reservation and a cruise

```
@Entity
public class Reservation {
  …
  @ManyToOne
  private Cruise c;
  …
}
```

```
@Entity
public class Cruise {
 …
 @OneToMany(mappedBy="c")
  private Collection<Reservation>
     r;
  …
}
```

Many to many unidirectional relations

Eg: a reservation and a cabin

```
@Entity
public class Reservation {
  …
@ManyToMany
@JoinTable(name =
    “CABIN_RESERVATION”,
     joinColumns{@JoinColumn
        (name=”RESERVATION_ID
        ”)}
     inverseJoinColumns{@Joincol
        umn(name=”CABIN_ID”)}
  private Collection<Cabin> cabin;
  …
}
```

```
@Entity
public class Cabin {
  …
}
```

Many to many bidirectional relations

- Eg: a customer and his reservations

```
@Entity
public class Customer {
  …
  @ManyToMany
@JoinTable(name =
    “CABIN_RESERVATION”,
     joinColumns{@JoinColumn
        (name=”RESERVATION_ID”)
        }
     inverseJoinColumns{@Joincolu
        mn(name=”CABIN_ID”)}
  private Collection<Reservation> r;
  …
}
```

```
@Entity
public class Reservation {
  …
 @ManyToMany(mappedBy=“r”)
  private Collection<Customer> c;
  …
}
```

Relations ans Collections

- You can use
  - Collections
  - Sets
  - Lists
    - They have different properties, so they represents different relation details. EG: @OrderBy("*lastName ASC*") for Lists
  - Maps
    - Primary key is a field of the mapping object, field comes from the mapped object
    - es.

      @OneToMany

      @MapKey(name="number")

      Private Map<String, Phone> phoneNumbers;

Cascading

- When an entity manager performs an operation on an entity, sometimes we want to cascade it on related objects
- We can customize this behavior with cascade attributes es. @OneToMany(cascade=CascadeType.ALL)
- Types:
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - ALL

Hyerarchies

- Java allows hyerarchies, RDBMS don't

  .. We have a problem!
- EJB 3.0 provides three ways to map hyerarchies:
  - Single table
  - A table for each concrete class
  - A table for each subclass

Single table

- A single table to store all attributes in the hierarchy
  - We store a coloumn with each tuple element type
  - We use the following annotations:
    - @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
    - @DiscriminatorColumn(name="DTYPE", discriminatorType=DiscriminatorType.STRING)
    - @DiscriminatorValue("Type_A")
  - Plus: simple and efficient
  - Minus: we can't have NOT NULL coloumns

A table for each concrete class

- We create a table for each conrete class, with all attributes of the superclasses
- We use the annotation:
  @Inheritance(strategy=
  InheritanceType.TABLE_PER_CLASS)
- Plus:
  - We ca state constraints on coloumns
  - Easier to map on an existing DB
- Minus:
  - Schema is not normalized
  - Tough for entity manager
  - Requires SQL UNION

A table for each subclass

- We create a table for each subclass and we store properties as coloumns
  - Normalized!
- We use the annotation:
  @Inheritance(strategy=
  InheritanceType.JOINED)
- Plus
  - We can state constraints on coloumns
  - We have a normalized DB
  - No SQL UNION
- Minus:
  - Not as efficient as single coloumn

References

- Burke & Monson-Haefel. *Enterprise JavaBeans 3.0.* O Reilly, fifth edition 2006.
- Ball et. al. *The Java EE 5 tutorial*. Sun Microsystems 2006. http://java.sun.com/javaee/5/docs/tutorial/doc/