HIberanete

Ch7 : Assosisatons

TO do : One example each of different assosiation

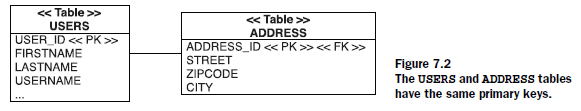
One to One relationship

1. Single values assossiation are best represented with a <component> mapping. This is usually the simplest way to represent one-to-one relationships, because the lifecycle is almost always dependent in such a case, it’s either an aggregation or a composition in UML.
2. Example : between User and Address (the user has a billingAddress, homeAddress, and shippingAddress)

***Shared primary key associations***

Rows in two tables related by a primary key association share the same primary key values.

The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the objects are saved.



public class User {

...

private Address shippingAddress;

// Getters and setters

}

<one-to-one name="shippingAddress"

class="Address"

cascade="save-update"/>

1. Cascade – save-update make sense as when user will be saved address will be saved too
2. How can Hibernate possibly know that the record in the ADDRESS table needs to get the same primary key value as the USERS row?

public class Address {

...

private User user;

// Getters and setters

}

<one-to-one name="user"

class="User"

constrained="true"/>

1. constrained="true", adds a foreign key constraint linking the primary key of the ADDRESS table to the primary key of the USERS table.
2. You can now use the special foreign identifier generator for Address objects. (not clear if this is mandatory to use)

<class name="Address" table="ADDRESS">

<id name="id" column="ADDRESS\_ID">

<generator class="foreign">

<param name="property">user</param>

</generator>

</id>

...

<one-to-one name="user"

class="User"

constrained="true"/>

</class>

When an Address is saved,

the primary key value is taken from the user property. The user property is a reference

to a User object; hence, the primary key value that is inserted is the same

as the primary key value of that instance

Annotation

@Entity

@Table(name = "ADDRESS")

public class Address {

@Id @GeneratedValue(generator = "myForeignGenerator")

@org.hibernate.annotations.GenericGenerator(

name = "myForeignGenerator",

strategy = "foreign",

parameters = @Parameter(name = "property", value = "user")

)

@Column(name = "ADDRESS\_ID")

private Long id;

...

private User user;

}

Class User {

@OneToOne

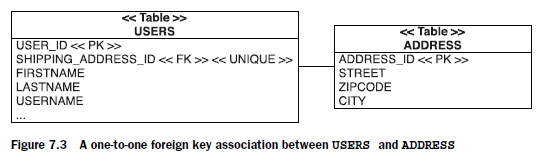
@PrimaryKeyJoinColumn

private Address shippingAddress;

}

***One-to-one foreign key associations***

Let’s change the mapping from a User to an Address. Instead of the shared primary key, you now add a SHIPPING\_ADDRESS\_ID column in the USERS table:



<class name="User" table="USERS">

<many-to-one name="shippingAddress"

class="Address"

column="SHIPPING\_ADDRESS\_ID"

cascade="save-update"

unique="true"/>

</class>

1. Instead of sharing a primary key, two rows can have a foreign key relationship. One table has a foreign key column that references the primary key of the associated table

Why to use many to one mapping :

1. The mapping element in XML for this association is <many-to-one>—not <oneto-one>, as you might have expected. The reason is simple You don’t care what’s on the target side of the association, so you can treat it like a *to-one* association without the *many* part. All you want is to express “This entity has a property that is a reference to an instance of another entity” and use a foreign key field to represent that relationship.
2. By making the SHIPPING\_ADDRESS\_ID column unique, you declare that a particular address

can be referenced by at most one user, as a shipping address

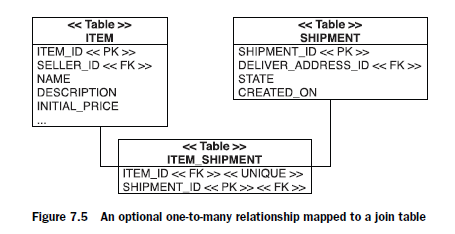
Bidirectional on User side (this is optional , and need to be specified if we want to make association bidirectional )

<one-to-one name="user"

class="User"

property-ref="shippingAddress"/>

In future : **One to one optional with Join**



public class Shipment {

...

private Item auction;

...

// Getter/setter methods

}

<class name="Shipment" table="SHIPMENT">

<id name="id" column="SHIPMENT\_ID">...</id>

...

<join table="ITEM\_SHIPMENT" optional="true">

<key column="SHIPMENT\_ID"/>

<many-to-one name="auction"

column="ITEM\_ID"

not-null="true"

unique="true"/>

</join>

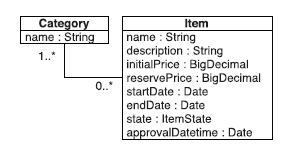
</class>

In future : **many to one optional with Join**

Not e: there is no inverse="true" for a many-to-one mapping .

***7.2.2 Many-to-many associations***

The association between Category and Item is a many-to-many association



if the Category has a set of Items, you can create this mapping:

<set name="items"

table="CATEGORY\_ITEM"

cascade="save-update">

<key column="CATEGORY\_ID"/>

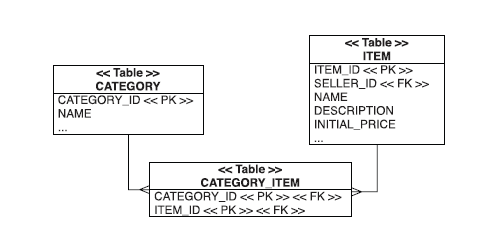
<many-to-many class="Item" column="ITEM\_ID"/>

</set> misleading ?

The join table (or *link table*, as some developers call it) has two columns: the foreign

keys of the CATEGORY and ITEM tables.

Category\_ITEM class is known as association class.



|  |  |
| --- | --- |
| <idbag name="items" table="CATEGORY\_ITEM”  cascade="save-update">  **<collection-id type="long" column="CATEGORY\_ITEM\_ID">**  **<generator class="sequence"/>**  **</collection-id>**  <key column="CATEGORY\_ID"/>  <many-to-many class="Item" column="ITEM\_ID"/>  </idbag> | <list name="items"  table="CATEGORY\_ITEM"  cascade="save-update">  <key column="CATEGORY\_ID"/>  **<list-index column="DISPLAY\_POSITION"/>**  <many-to-many class="Item" column="ITEM\_ID"/>  </list> |
| the primary key is a surrogate key column,  CATEGORY\_ITEM\_ID. Duplicate links are therefore allowed; | The primary key of the link table is a composite of the CATEGORY\_ID and  DISPLAY\_POSITION columns; |

In JPA

@ManyToMany

@JoinTable(

name = "CATEGORY\_ITEM",

joinColumns = {@JoinColumn(name = "CATEGORY\_ID")},

inverseJoinColumns = {@JoinColumn(name = "ITEM\_ID")}

)

private Set<Item> items = new HashSet<Item>();

***A bidirectional many-to-many association***

1. one side in a bidirectional association has to be mapped as inverse because you have named the foreign key column(s) twice.
2. this setting tells Hibernate to ignore changes made to the categories collection and that the other end of the association, the items collection, is the representation that should be synchronized with the database if you link instances in Java code.

|  |  |
| --- | --- |
| <class name="Category" table="CATEGORY">  ...  <set name="items"  table="CATEGORY\_ITEM"  cascade="save-update">  <key column="CATEGORY\_ID"/>  <many-to-many class="Item" column="ITEM\_ID"/>  </set> | <class name="Item" table="ITEM">  ...  <set name="categories"  table="CATEGORY\_ITEM”  **inverse="true"**  cascade="save-update">  <key column="ITEM\_ID"/>  <many-to-many class="Category" column="CATEGORY\_ID"/>  </set>  </class> |
| @ManyToMany  @JoinTable(  name = "CATEGORY\_ITEM",  joinColumns = {@JoinColumn(name = "CATEGORY\_ID")},  inverseJoinColumns = {@JoinColumn(name = "ITEM\_ID")}  )  private Set<Item> items = new HashSet<Item>(); | @ManyToMany(mappedBy = "items")  private Set<Category> categories = new HashSet<Category>();  As you can see, you don’t have to repeat the join-table declaration on the inverse  side. |

Good to know

q What types of collections may be used for bidirectional many-to-many associations?

1. A many-to-many association can’t be mapped with indexed collections on both sides.
2. It’s reasonable to map, for example, a <list> for the noninverse side of the association and a <bag> on the inverse side.
3. Hibernate won’t initializ or maintain the index column if the collection is inverse.
4. the cascading options all, delete, and delete-orphans aren’t meaningful for many-to-many associations.

Many to many when link table has more properties will be covered in future.

***Polymorphic many-to-one associations***

A *polymorphic association* is an association that may refer instances of a subclass of

the class that was explicitly specified in the mapping metadata

You don’t have to do anything special to enable polymorphic associations in

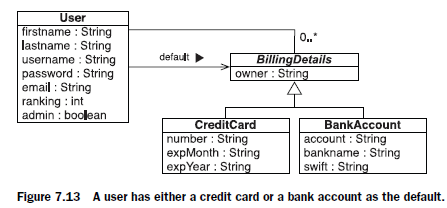
Hibernate; specify the name of any mapped persistent class in your association

mapping (or let Hibernate discover it using reflection), and then, if that class

declares any <union-subclass>, <subclass>, or <joined-subclass> elements,

the association is naturally polymorphic

Example



<many-to-one name="defaultBillingDetails"

class="BillingDetails"

column="DEFAULT\_BILLING\_DETAILS\_ID"/>

whats the issue with below

User user = (User) session.get(User.class, userid);

BillingDetails bd = user.getDefaultBillingDetails();

System.out.println( bd instanceof CreditCard ); // Prints "false"

CreditCard cc = (CreditCard) bd; // ClassCastException!

In this code, the typecast fails because bd is a proxy instance. When a method is invoked on the proxy, the call is delegated to an instance of CreditCard that is fetched lazily (it’s an instance of a runtime generated subclass, so instanceof also fails). Until this initialization occurs, Hibernate doesn’t know what the subtype of this would require a database hit, which you try to avoid

with lazy loading in the first place. To perform a proxy-safe typecast, use load():

User user = (User) session.get(User.class, userId);

BillingDetails bd = user.getDefaultBillingDetails();

// Narrow the proxy to the subclass, doesn't hit the database

CreditCard cc =

(CreditCard) session.load( CreditCard.class, bd.getId() );

expiryDate = cc.getExpiryDate();

After the call to load(), bd and cc refer to two different proxy instances, which

both delegate to the same underlying CreditCard instance. However, the second

proxy has a different interface, and you can call methods (like getExpiryDate())

that apply only to this interface.

Note that you can avoid these issues by avoiding lazy fetching, as in the following

code, using an eager fetch query:

User user = (User)session.createCriteria(User.class)

.add(Restrictions.eq("id", uid) )

.setFetchMode("defaultBillingDetails", FetchMode.JOIN)

.uniqueResult();

// The users defaultBillingDetails have been fetched eagerly

CreditCard cc = (CreditCard) user.getDefaultBillingDetails();

expiryDate = cc.getExpiryDate();

CH 8

***Mapping a natural key***

Issues with natural key

natural primary keys can be a bad idea. Natural keys often make it difficult to refactor the data model when business requirements change. They may even, in extreme cases, impact performance

Q if username is primary key in usertable

If you encountered a USERS table in a legacy schema, it’s likely that USERNAME is

the actual primary key. In this case, you have no surrogate identifier that is automatically

generated. Instead, you enable the assigned identifier generator strategy

to indicate to Hibernate that the identifier is a natural key assigned by the

application before the object is saved:

<class name="User" table="USERS">

<id name="username" column="USERNAME" length="16">

<generator class="assigned"/>

</id>

...

</class>

***Mapping a composite natural key***

Suppose that the primary key of the USERS table consists of a USERNAME and

DEPARTMENT\_NR. You can add a property named departmentNr to the User class

and create the following mapping:

<class name="User" table="USERS">

<composite-id>

<key-property name="username"

column="USERNAME"/>

<key-property name="departmentNr"

column="DEPARTMENT\_NR"/>

</composite-id>

...

</class>

The code to save a new User looks like this:

User user = new User();

// Assign a primary key value

user.setUsername("johndoe");

user.setDepartmentNr(42);

// Set property values

user.setFirstname("John");

user.setLastname("Doe");

session.saveOrUpdate(user);

session.flush();

suppose that the DEPARTMENT\_NR is a foreign key referencing the DEPARTMENT

table, and that you wish to represent this association in the Java domain

model as a many-to-one association.

***Foreign keys in composite primary keys***

We recommend that you map a foreign key column that is also part of a composite

primary key with a regular <many-to-one> element, and disable any Hibernate

inserts or updates of this column with insert="false" update="false", as

follows:

<class name="User" table="USER">

<composite-id name="userId" class="UserId">

<key-property name="username"

column="USERNAME"/>

<key-property name="departmentId"

column="DEPARTMENT\_ID"/>

</composite-id>

<many-to-one name="department"

class="Department"

column="DEPARTMENT\_ID"

insert="false" update="false"/>

...

</class>

Hibernate now ignores the department property when updating or inserting a

User, but you can of course read it with johndoe.getDepartment(). The relationship

between a User and Department is now managed through the departmentId

property of the UserId composite key class:

UserId id = new UserId("johndoe", department.getId() );

User user = new User();

// Assign a primary key value

user.setUserId(id);

// Set property values

user.setFirstname("John");

user.setLastname("Doe");

user.setDepartment(department);

session.saveOrUpdate(user);

session.flush();

Only the identifier value of the department has any effect on the persistent state;

the setDepartment(department) call is done for consistency: Otherwise, you’d

have to refresh the object from the database to get the department set after the

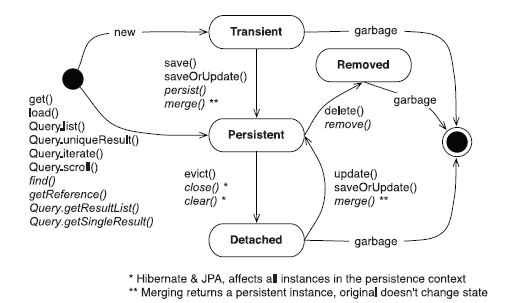
flush.

*Arbitrary join conditions with formulas and rest of 8.1 is of practical usage.*

*It can be looked in details in later ...*

Ch 9

**Hiber obj State**

****

***Transient objects***

Objects instantiated using the new operator aren’t immediately persistent. Their state is *transient*

**Persistance Object**

A *persistent* instance is an entity instance with a *database identity*

Objects persistent by calling the save() method of the persistence manager

Alternatively, a persistent instance might be an instance retrieved from the database by execution of a query, by an identifier lookup, or by navigating the object graph starting from another persistent instance. In other words, persistent instances are always associated witha Session and are *transactional*

Remove object :

An obj is in *removed* state if it has been scheduled for deletion at the end of a unit of work, but it’s still managed by the persistence context until the unit of work completes.

**Detached Object :**

In the case of Hibernate, however, these instances lose their association with the persistence manager when you close() the Session. We refer to these objects as *detached*, indicating that their state is no longer guaranteed to be synchronized with database state

GTK

1. ORM software must have a strategy for detecting which persistent objects have been modified by the application in the transaction.
2. We call this ***automatic dirty checking***(an object with modifications that haven’t yet been propagated to the database is considered *dirty)*. Again, this state isn’t visible to the application. We call this feature *transparent transaction-level write-behind*, meaning that Hibernate propagates state changes to the database as late as possible but hides this detail from the application
3. If you only want to update modified columns, you can enable dynamicSQL generation by setting **dynamic-update="true**" in a class mapping

***The persistence context cache***

.

Is it same as 1st level cache

1. A persistence context is a cache of persistent entity instances. This means it remembers all persistent entity instances you’ve handled in a particular unit of work.
2. Automatic dirty checking is one of the benefits of this caching. Another benefit is *repeatable read* for entities and the performance advantage of a unit of work-scoped cache.

For eg. In case of load call it 1st check in persistent cache, and goes to database only if the object does not exist in cache thus saves time and performance.

9.2 Object identity and equality

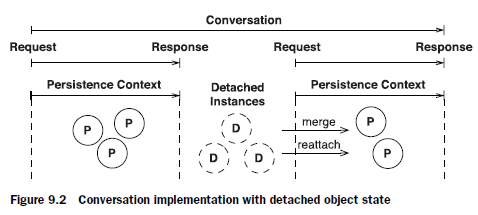
***Introducing conversations***

Two strategies are available to implement a conversation in a Hibernate or Java

Persistence application: with detached objects or by extending a persistence context.

Objects are held in detached state

during user think-time, and any modification of these objects is made persistent manually through reattachment or merging.



***The scope of object identity***

***GTK***

1. Java identity, a==b, and database identity, a.getId().equals( b.getId() )
2. conditions under which Java identity is equivalent to database identity is referred as the *scope of object identity*
3. A primitive persistence layer with *no identity scope = no gaureenty that one row will be represented by one object*
4. A persistence layer using *persistence context-scoped identity = IN a* single persistence context, only one object instance represents a particular database row.
5. Hib uses the *persistence context-scoped identity,*
6. *Issue with process scope is , object needs to be synchronized in multithreaded application.*

Session session1 = sessionFactory.openSession();

Transaction tx1 = session1.beginTransaction();

// Load Item with identifier value "1234"

Object a = session1.get(Item.class, new Long(1234) );

Object b = session1.get(Item.class, new Long(1234) );

( a==b ) // True, persistent a and b are identical

tx1.commit();

session1.close();

// References a and b are now to an object in detached state

Session session2 = sessionFactory.openSession();

Transaction tx2 = session2.beginTransaction();

Object c = session2.get(Item.class, new Long(1234) );

( a==c ) // False, detached a and persistent c are not identical

tx2.commit();

session2.close();

Hibernate doesn’t guarantee Java identity, so a and c aren’t identical. Of course, a test for database identity, a.getId().equals( c.getId()

Set allObjects = new HashSet();

allObjects.add(a);

allObjects.add(b);

allObjects.add(c);

it should be ideally one , but size in the map will be 2. So that’s why it is imp to understand equals and hascode.

1. If you are not comparing objects in detached state, you may not have to override equals() and hashCode().The identity scope guarantee provided by Hibernate is sufficient if you never compare detached instances. that is, if you never put detached instances into the same Set.
2. But if that is not the case : then override equals and hashcode with **Business key**
3. Don’t use Hib id column to equals because , its only got generated when we persist the obj.
4. Don’t use the all the properties, use only unique key properties, for example don’t use user password, as it can e changed between two sessions.
5. Business key includes natural natural/ primary key.

9.3 Hib Interface

1) Imp interfaces. In the case of Hibernate are Session, Query, Criteria, and Transaction.

2) In Java Persistence interfaces main interface is the EntityManager, same as the Hibernate Session. Others are Query and EntityTransaction .

***Making an object persistent***

Serializable itemId = session.save(item);

tx.commit();

session.close();

***Retrieving a persistent object***

Item item = (Item) session.load(Item.class, new Long(1234));

// Item item = (Item) session.get(Item.class, new Long(1234));

**Get Vs Load**

1. The one difference between get() and load() is how they indicate that the instance could not be found.
2. Get returns null and The load() method throws an ObjectNotFound Exception.
3. More important, the load() method may return a *proxy*, a placeholder, without hitting the database. A consequence of this is that you may get an ObjectNotFoundException later, as soon as you try to access the returned placeholder and force its initialization.
4. only returns an initialized object instance if it’s already managed by the current persistence context.
5. get()method on the other hand never returns a proxy, it always hits the database.

Why proxy is useful :

1. Use proxy to get instanace and assigne a refeance to other objects.
2. For example : you need the item only for a single purpose: to set an association with a Comment: aComment.setForAuction(item).

***Modifying a persistent object***

Item item = (Item) session.get(Item.class, new Long(1234));

item.setDescription("This Playstation is as good as new!");

tx.commit();

Note : No update is called and still object got saved .Its called Automatic dirty checking.

***Making a persistent object transient***

session.delete(item);

The SQL DELETE is executed only when the Session’s persistence context is synchronized with the database at the end of the unit of work.

After the Session is closed, the item object is considered an ordinary transient instance. The transient instance is destroyed by the garbage collector if it’s no longer referenced by any other object. Both the in-memory object instance and the persistent database row will have been removed.

*Do I have to load an object to delete it?* Yes, an object has to be loaded into the persistence context; an instance has to be in persistent state to be removed (note that a proxy is good enough).

You may have Hibernate interceptors enabled, and the object must be passed through these interceptors to complete its lifecycle.

***Making a detached object transient***

session.delete(item);

1. This means you don’t have to reattach (with update() or lock()) a detached

instance to delete it from the database

1. In this case, the call to delete() does two things: It reattaches the object to the Session and then schedules the object for deletion, executed on tx.commit().

***Reattaching a modified detached instance***

item.setDescription(...); // Loaded in previous Session

Session sessionTwo = sessionFactory.openSession();

Transaction tx = sessionTwo.beginTransaction();

sessionTwo.update(item);

item.setEndDate(...);

tx.commit();

sessionTwo.close();

1. It doesn’t matter if the item object is modified before or after it’s passed to update(). The important thing here is that the call to update() is reattaching the detached instance to the new Session.
2. Hibernate always treats the object as dirty and schedules an SQL UPDATE., which will be executed

during flush.

1. One way to avoid this UDPATE statement is to configure the class mapping of Item with the select-before-update="true" attribute. Hibernate then determines whether the object is dirty by executing a SELECT statement and comparing the object’s current state to the current database state.

***Reattaching an unmodified detached instance***

A call to lock() associates the object with the Session and its persistence context

without forcing an update,

sessionTwo.lock(item, LockMode.NONE);

item.setDescription(...);

item.setEndDate(...);

1. it *does* matter whether changes are made before or after the object has been reattached. Changes made before the call to lock() aren’t propagated to the database, you use it only if you’re sure the detached instance hasn’t been modified.

**Merging and NonUniqueObjectException**

. The error message of the exception is *A persistent instance with the same database identifier is already associated with the Session!*

*Examples :*

item.getId(); // The database identity is "1234"

item.setDescription(...);

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Item item2 = (Item) session.get(Item.class, new Long(1234));

session.update(item); **// Throws exception!**

tx.commit();

session.close();

before reattachment, another instance that represents the same database row has already been loaded into the persistence context of that Session. Obviously, the reattachment through update() clashes with this already persistent instance, and a NonUniqueObjectException is thrown

You can let Hibernate merge item and item2 automatically:

item.getId() // The database identity is "1234"

item.setDescription(...);

Session session= sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Item item2 = (Item) session.get(Item.class, new Long(1234));

**Item item3 = (Item) session.merge(item);**

(item == item2) // False

(item == item3) // False

(item2 == item3) // True

return item3;

tx.commit();

session.close();

With merge first, Hibernate checks whether a persistent instance in the persistence context has the same database identifier as the detached instance you’re merging.

If there is an equal persistent instance in the persistence context, Hibernate *copies* the state of the detached instance onto the persistent instance . In other words, the new description that has been set on the detached item is also set on the persistent item2.

***Controlling the persistence context cache***

1. Every object in persistent state is known to the persistence context, and a duplicate, a *snapshot* of each persistent instance, is held in the cache. This snapshot is used internally for dirty checking, to detect any modifications you made.
2. You may get OutOfMemory- Exception. This is typically the case when you load thousands of objects in a Session but never intend to modify them.
3. Keep the size of your persistence context to the necessary minimum.
4. call session.evict(object) to detach a persistent instance manually

from the persistence context cache.

1. You can call session.clear() to detach *all* persistent instances from the persistence context

**Flush**

***1)*** The synchronization of a persistence context with the database is called *flushing*. Hibernate flushes occur at the following times:

* When a Transaction on the Hibernate API is committed
* Before a query is executed
* When the application calls session.flush() explicitly

***2)*** Hibernate Session implements *write-behind*. Changes to persistent objectsmade in the scope of a persistence context aren’t immediately propagated to the database.

3) This allow hib to coalesce many changes into a minimal number of database requests, helping minimize the impact of network latency.

4) Another advantage of executing DML as late as possible, toward the end of the transaction, is shorter lock durations inside the database.

5) Committing a transaction with the JDBC API doesn’t trigger a flush. Hibernate doesn’t flush before every query. If changes are held in memory that would affect the results of the query, Hibernate synchronizes first by default.

Controlling flush

Hibernate FlushMOde can be changed via a call to session.setFlushMode(). The default flush mode is FlushMode.AUTO

If you chose FlushMode.COMMIT, the persistence context isn’t flushed before query execution (it’s flushed only when you call Transaction.commit() or Session.flush() manually).

This setting may expose you to stale data: Modifications you make to managed objects

only in memory may conflict with the results of the query.

By selecting Flush-Mode.MANUAL, you may specify that only explicit calls to flush() result in synchronization of managed state with the database.

***9.4 The Java Persistence API***

JPA provides a *subset* of functionality of the *superset* of Hibernate native APIs.

1)The equivalent of the Hibernate SessionFactory is the JPA EntityManagerFactory:

EntityManagerFactory emf =

Persistence.createEntityManagerFactory("caveatemptorDatabase");

EntityManager em = emf.createEntityManager();

EntityTransaction tx = em.getTransaction();

tx.begin();

***Making an entity instance persistent***

*Same as hib save ()*exceptthat persist() doesn’t return the database identifier value of the entity instance

EntityManager em = emf.createEntityManager();

EntityTransaction tx = em.getTransaction();

tx.begin();

em.persist(item);

tx.commit();

em.close();

there is a aso persist method in Hib, but that is a different, it dose not propagate/cascade changes

***Retrieving an entity instance***

1. Equlivalent to load\get
2. don’t need to cast the returned value of the find() operation; it’s a generic method

Item item = em.find(Item.class, new Long(1234));

Item item = em.getReference(Item.class, new Long(1234));

Referace is smae as load : as soon as you try to access any property of the item that isn’t the database

identifier property, an additional SELECT is executed to fully initialize the placeholder.

This also means you should expect an EntityNotFoundException at this point

***Modifying a persistent entity instance***

***Making a persistent entity instance transient***

Item item = em.find(Item.class, new Long(1234));

em.remove(item);

equivalent to delete in hib

note that you can’t call remove() on an entity instance in detached state, or an exception will be thrown, so either merge or get a reference of it through find\ getReference

***Flushing the persistence context***

The persistence context of an EntityManager is flushed whenever commit() on an EntityTransaction is called. But its allowed to flush at a different time for JPA implementation as in Hib.

***Manual detachment of entity instances***

em.clear();

it detaches all the objects but there is no equivalent of evict

***9.5 Using Java Persistence in EJB components***

***Injecting an EntityManager***

In an EJB 3.0 server, **a container-managed EntityManager** is available through *dependency injection*.

@Stateless

public class ManageAuctionBean implements ManageAuction {

@PersistenceContext

private EntityManager em;

// or setter injection:

@PersistenceContext

public void setEntityManager(EntityManager em) {

this.em = em;

}

How to use it

You can access the EntityManager in the findAuctionByName() method of the action. The container automatically *injects* an instance of an EntityManager into the em field of the bean,

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public Item findAuctionByName(String name) {

return (Item) em.createQuery()...

...

The injected EntityManager is maintained by the container. You don’t have to flush or close it, nor do you have to start and end a transaction, same as hibernate template.

***How to work with two databases***

Here is a variation that works with two databases—that is, two persistence units:

@Stateless

public class ManageAuctionBean implements ManageAuction {

@PersistenceContext(unitName = "auctionDB")

private EntityManager auctionEM;

@PersistenceContext(unitName = "auditDB")

private EntityManager auditEM;

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public void createAuction(String name, BigDecimal price) {

Item newItem = new Item(name, price);

auctionEM.persist(newItem);

auditEM.persist( new CreateAuctionEvent(newItem) );

...

}

}

***If the participate in same transaction if they are using diff Entity manager :***

Note that EntityManager instances from two different persistence units aren’t sharing the same persistence context. Naturally, both are independent caches of managed entity objects, but that doesn’t mean they can’t participate in the same system transaction.

***9.5.2 Looking up an EntityManager***

Instead of letting the container inject an EntityManager on your field or settermethod, you can look it up from JNDI when you need it:

@Stateless

@PersistenceContext(name = "em/auction", unitName = "auctionDB")

public class ManageAuctionBean implements ManageAuction {

@Resource

SessionContext ctx;

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public Item findAuctionByName(String name) {

EntityManager em = (EntityManager) ctx.lookup("em/auction");

return (Item) em.createQuery()...

}

}

1. First, you declare that you want the component environment of the bean populated with an EntityManager and that the name of the bound reference is supposed to be em/auction. The full name in JNDI is java:comp/env/em/auction—the java:comp/env/ part is the so called *bean-naming context*.
2. Everything in that subcontext of JNDI is bean-dependent. In other words, the EJB container reads this annotation and knows that it has to bind an EntityManager for this bean only, at runtime when the bean executes under the namespace in JNDI that is reserved for this bean.
3. You look up the EntityManager in your bean implementation with the help of the SessionContext. The benefit of this context is that it automatically prefixes the name you’re looking for with java:comp/env/; hence, it tries to find the reference in the bean’s naming context, and not the global JNDI namespace. The @Resource annotation instructs the EJB container to inject the SessionContext for you.
4. A persistence context is created by the container when the first method on the EntityManager is called, and it’s flushed and closed when the transaction ends— when the method returns.

An EJB container also allows you to access an EntityManagerFactory for a persistence

unit directly.

In a container, you can again utilize automatic dependency injection to get an

EntityManagerFactory:

@Stateless

public class ManageAuctionBean implements ManageAuction {

@PersistenceUnit(unitName = "auctionDB")

EntityManagerFactory auctionDB;

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public Item findAuctionByName(String name) {

EntityManager em = auctionDB.createEntityManager();

...

Item item = (Item) em.createQuery()...

...

em.flush();

em.close();

return

The EntityManager you created from the injected factory is again application-managed—the container won’t flush this persistence context, nor close it

If you close() the EntityManager, it doesn’t immediately close its persistence context, if this persistence context has been associated with a transaction. The persistence context is closed when the

transaction completes. However, any call of the closed EntityManager throws an

exception

***selective reassociation of detached instances*.** This means the application can efficiently reattach a *subgraph* of a graphof detached objects with the current (“second”) Hibernate Session. Once adetached object has been reattached to a new Hibernate persistence manager, itmay be considered a persistent instance, and its state will be synchronized with thedatabase at the end of the transaction .

Reattachment might result in the creation of new rows in the database when a reference is created from a detached instance to a new transient instance. For example, a new Bid might have been added to a detached Item while it was on the presentation tier. Hibernate can detect that the Bid is new and must be inserted in the database. For this to work, Hibernate must be able to distinguish between a “new” transient instance and an “old” detached instance. Transient instances (such as the Bid) might need to be saved; detached instances

Ch 10

Transaction

ACID property

A – atomicity – either all should succeed or none.

I – isolation – transactions allow multiple users to work concurrently with the same data without compromising the integrity and correctness of the data; a particular transaction should not be visible to other concurrently running transactions.

C – Consistency-DB is left in a clean and consistent state after the transactions completes. Your database integrity rules guarantee consistency.

D- Changes are not lost\persistence

*transaction demarcation*

To execute all your database operations inside a transaction, you have to mark the boundaries of that unit of work. You must start the transaction and at some point, commit the changes. If an error occurs, you have to roll back the transaction to leave the data in a consistent state.

How to handle multiple databases

In a case where we hadling multiple DB, you can’t achieve atomicity with JDBC alone. You need a *transaction manager* that can handle several resources in one *system transaction*.

Such transaction-processing systems expose the *Java Transaction API* (JTA) for interaction with the developer. The main API in JTA is the UserTransaction interface with methods to begin() and commit() a system transaction.

Can we use JDBC transcation in Hib –

Yes , but its discourage due to binding to JDBC interface

Transaction interfaces

java.sql.Connection—Plain JDBC transaction demarcation with set-

AutoCommit(false), commit(), and rollback().

org.hibernate.Transaction- Unified transaction demarcation in Hibernate

applications. It works in a nonmanaged plain JDBC environment and

also in an application server with JTA as the underlying system transaction

service.

javax.transaction.UserTransaction—Standardized interface for programmatic

transaction control in Java; part of JTA. This should be your primary

choice whenever you have a JTA-compatible transaction service and

want to control transactions programmatically.

javax.persistence.EntityTransaction—Standardized interface for programmatic

transaction control in Java SE applications that use Java Persistence.

CMT – Declarative transaction through EJB

Its the responsibility of EJB container to handle the tranctions.

***Programmatic transactions in Java SE***

1. The hibernate.transaction.factory\_class option defaults to org.hibernate.

transaction.JDBCTransactionFactory,

1. Hibernate obtains a JDBC connection for each Session you’re going to work with:

Session session = null;

Transaction tx = null;

try {

session = sessionFactory.openSession();

tx = session.beginTransaction();

concludeAuction(session);

tx.commit();

} catch (RuntimeException ex) {

tx.rollback();

} finally {

session.close();

}

1. A JDBC Connection from the connection pool is obtained only when the database transaction begins. The call to beginTransaction() translates into **setAutoCommit(false**) on the fresh JDBC Connection.
2. The Session is now bound to this database connection, and all SQL statements (in this case, all SQL required to conclude the auction) are sent on this connection.
3. After you commit the transaction (or roll it back), the database connection is released and unbound from the Session. Beginning a new transaction with the same Session obtains another connection from the pool.

***Handling exceptions***

1. Notic that in the above example we are catching a RunTimeException.
2. Until Hibernate 3.x, all exceptions thrown by Hibernate were *checked* exceptions, so every Hibernate API forced the developer to catch and handle exception
3. it soon became clear that this doesn’t make sense, because all exeption thrown by hib are fatel.
4. starting with Hibernate 3.x, all exceptions thrown by Hibernate are subtypes of the unchecked Runtime- Exception,
5. Any Hibernate operation, including flushing the persistence context, can throw a

RuntimeException. Even rolling back a transaction can throw an exception

catch (RuntimeException ex) {

try {

tx.rollback();

} catch (RuntimeException rbEx) {

log.error("Couldn’t roll back transaction", rbEx);

}

throw ex;

} finally {

session.close();

}

1. An optional method call in the example is setTimeout(), which takes the number of seconds a transaction is allowed to run, however it is not available in java SE.

***Programmatic transactions with JTA***

1. JTA; javax.transaction.UserTransaction is the primary interface to begin and end transactions.
2. The quality of JTA implementations is usually higher compared to simple JDBC connection pools

**Using JTA for multiple DBs**

A Java EE transaction manager can enlist multiple resources in a single transaction. If you work with several databases (or more than one resource), you probably want a ***two-phase commit*** protocol to guarantee atomicity of a transaction across resource boundaries. In such a scenario, Hibernate is configured with several SessionFactorys, one for each database, and their Sessions obtain managed database connections that all participate in the same system transaction.

**Using Hib transcation inside a JTA \JAVA EE Server**

you want to continue using the Hibernate Transaction API to keep your code runnable in Java SE *and* with managed Java EE services, without any code changes.

1. The hibernate.transaction.factory\_class option must be set to org.

hibernate.transaction.JTATransactionFactory

You have to set the

1. hibernate.transaction.manager\_lookup\_class option to configure

both: for example, to org.hibernate.transaction.JBossTransaction- ManagerLookup

1. Hibernate is no longer responsible for managing a JDBC connection pool;

it obtains managed database connections from the runtime container. These connections are exposed by the JTA provider through JNDI, Section 2.2.

1. the database connection-handling is slightly different.
2. Without JTA, Hibernate would hold on to a particular database connection from the beginning until the end of the transaction.
3. A connection is obtained and used for only a single SQL statement and then is immediately returned to the managed connection pool.
4. The application server guarantees that it will hand out the same connection during the same transaction, when it’s needed again for another SQL statement

**Hib in JTA env with JTA taking care of transaction**

|  |  |
| --- | --- |
| HIB Transaction | JTA Transaction |
| Session session = null;  Transaction tx = null;  try {  session = sessionFactory.openSession();  tx = session.beginTransaction();  tx.setTimeout(5);  concludeAuction(session);  tx.commit();  } catch (RuntimeException ex) {  try {  tx.rollback();  } catch (RuntimeException rbEx) {  log.error("Couldn't roll back transaction", rbEx);  }  throw ex; } finally {  session.close();  } | UserTransaction utx = (UserTransaction) new InitialContext()  .lookup("java:comp/UserTransaction");  Session session1 = null;  Session session2 = null;  try {  utx.begin();  session1 = auctionDatabase.openSession();  session2 = billingDatabase.openSession();  concludeAuction(session1);  billAuction(session2);  session1.flush();  session2.flush();  utx.commit();  } catch (RuntimeException ex) {  try { utx.rollback();}  catch (RuntimeException rbEx) {  log.error("Couldn't roll back transaction", rbEx);  }  throw ex;  } finally {  session1.close();  session2.close();  } |

1. a handle on a JTA UserTransaction must be obtained from the JNDI registry. Then, you begin and end a transaction
2. Even if you aren’t using the Transaction API, you should still configure

hibernate.transaction.factory\_class and hibernate.transaction. manager\_lookup\_class for JTA and your environment,

1. With default settings, it’s also your responsibility to flush() each Session manually to synchronize it with the database (to execute all SQL DML).while in case of Hib transc , hib does it for you.
2. you can enable the hibernate.transaction.flush\_before\_completion and/or the hibernate.transaction.auto\_close\_session configuration options and let Hibernate take care of this for you again—flushing and closing is then part of the internal synchronization procedure of the transaction manager and occurs before (and after, respectively) the JTA transaction ends.

***Container-managed transactions withHib***

1. hibernate.transaction.factory\_class option must be set to org. hibernate.transaction.CMTTransactionFactory
2. you define transaction boundaries with annotations.

@Stateless

public class ManageAuctionBean implements ManageAuction {

@**TransactionAttribute(TransactionAttributeType.REQUIRED)**

public void endAuction(Item item) {

Session session1 = auctionDatabase.openSession();

Session session2 = billingDatabase.openSession();

concludeAuction(session1, item);

billAuction(session2, item);

}

...

}

***\*** note that all EJB session beans default to CMT, so if you want to disable CMT

and call the JTA UserTransaction directly in any session bean method, annotate

the EJB class with @TransactionManagement(TransactionManagementType.

BEAN). You’re then working with *bean-managed transactions* (BMT).

***Transactions with Java Persistence***

*resource-local* transaction applies to all transactions that are controlled by the application (programmatic) and that aren’t participating in a global system transaction. They translate directly into the native transaction system of the resource you’re dealing with. Because you’re working with JDBC databases, this

means a resource-local transaction translates into a JDBC database transaction.

1. Resource-local transactions in JPA are controlled with the EntityTransaction API.
2. Exceptions thrown by JPA are subtypes of RuntimeException, any exception thrown by any method of the EntityManager interfaces triggers an automatic rollback of the current transaction

|  |  |
| --- | --- |
| Plain JAVA SE(Resource-local transactions) | In JAVA EE Server |
| EntityManager em = null;  EntityTransaction tx = null;  try {  em = emf.createEntityManager();  tx = em.getTransaction();  tx.begin();  concludeAuction(em);  tx.commit();  } catch (RuntimeException ex) {  try {  tx.rollback();  } catch (RuntimeException rbEx) {  log.error("Couldn't roll back transaction", rbEx);  }  throw ex;  } finally {  em.close();  } | UserTransaction utx = (UserTransaction) new InitialContext()  .lookup("java:comp/UserTransaction");  EntityManager em = null;  try {  utx.begin();  em = emf.createEntityManager();  concludeAuction(em);  utx.commit();  } catch (RuntimeException ex) {  try { utx.rollback();  } catch (RuntimeException rbEx) {  log.error("Couldn't roll back transaction", rbEx);  }  throw ex;  } finally {  em.close();  } |

***JTA transactions with Java Persistence***

If your Java Persistence code is deployed in an environment where JTA is available,

and you want to use JTA system transactions, you need to call the JTA UserTransaction

interface to control transaction boundaries programmatically:

The persistence context of the EntityManager is scoped to the JTA transaction.

All SQL statements flushed by this EntityManager are executed inside the JTA

transaction on a database connection that is enlisted with the transaction. The

persistence context is flushed and closed automatically when the JTA transaction

commits. You could use several EntityManagers to access several databases

in the same system transaction, just as you’d use several Sessions in a native

Hibernate application

***Java Persistence and CMT***

@Stateless

public class ManageAuctionBean implements ManageAuction {

@PersistenceContext(unitName = "auctionDB")

private EntityManager auctionEM;

@PersistenceContext(unitName = "billingDB")

private EntityManager billingEM;

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public void endAuction(Item item)

throws AuctionNotValidException {

concludeAuction(auctionEM, item);

billAuction(billingEM, item);

}

...

}

Each EntityManager has a persistence context that spans the

scope of the transaction and is flushed automatically when the transaction commits.

AuctionNotValidException. is a custom exception (application exception )you write, and does not trigger a roll back .Unlike runtime System exception which triagger a rollback

Its upto you, how u decide to handle application exception and you want roll back

You have two choices how you can roll back a transaction if an application exception is

thrown:

catch it and call the JTA UserTransaction manually, and set it to roll back.

Or you can add an @ApplicationException(rollback = true)

annotation to the class of AuctionNotValidException—the container will then

recognize that you wish an automatic rollback whenever an EJB method throws

this application exception

***Controlling concurrent access***

How databases implement concurrency control is of the utmost importance in

your Hibernate or Java Persistence application. Applications *inherit* the isolation

guarantees provided by the database management system. For example, Hibernate

never locks anything in memory.

Q How hibernate implement \prevent concurrent access

There are two stragies

1. Optimistic control
2. Pessamistic locking

***Optimistic concurrency control***

1. An optimistic approach always assumes that everything will be OK and that conflicting data modifications are rare. Optimistic concurrency control raises an error only at the end of a unit of work, when data is written
2. If you don’t enable optimistic concurrency control, and by default it isn’t enabled, your application runs with a *last commit wins* strategy, resulting in lost updates.
3. *Optimistic locking with versioning is enabled as soon as you add a <version> or a <timestamp> property to a persistent class mapping. There is no other switch*.

Other options when two users simultaneously modify a set of data

1. First commit wins and second got a error message
2. Merage confilict : Seconds user gets a error message and you allow him after displaying error message to merge conflict.

***Enabling versioning in Hibernate***

Hibernate provides automatic versioning.

public class Item {

...

private int version;

...

}

<class name="Item" table="ITEM">

<id .../>

<version name="version" access="field" column="OBJ\_VERSION"/>

...

</class>

1. Each entity instance can have a version, which can be a number or a timestamp. Hibernate increments an object’s version when it’s modified, compares versions automatically, and throws an exception if a conflict is detected
2. The <version> property mapping in XML must be placed immediately after the identifier property mapping. It can have a getter method but no setter method.
3. A timestamp can also be used for versioning but is less safe two timestamp can be same and in a global distributated application a time may not be synchronized in the all the nodes.

public class Item {

...

private Date lastUpdated;

...

}

<class name="Item" table="ITEM">

<id .../>

<timestamp name="lastUpdated"

access="field"

column="LAST\_UPDATED"/>

...

</class>

How does Hibernate use the version to detect a conflict?

1. Consider the item scenario – two uses have select\fetch the item object and trying to modify it now.
2. Both will get version as 1.
3. The user which will update the values 1st , hib will update the version num to 2

update ITEM set INITIAL\_PRICE='12.99', OBJ\_VERSION=2

where ITEM\_ID=123 and OBJ\_VERSION=1

1. When 2nd user will commit the changes, he will have a version num value 1, so no row will be updated and in this case hib throws a StaleObjectStateException.

If another concurrent unit of work updated and committed the same row, the OBJ\_VERSION column no longer contains the value 1, and the row isn’t updated.

Hibernate checks the row count for this statement as returned by the JDBC driver—which in this case is the number of rows updated, zero—and throws a StaleObjectStateException.

Opt Locking in relationships (GTK)

Think about the relationship between User and BillingDetails, a one-to-many entity association: If a Credit- Card is modified, the version of the related User isn’t incremented.

If you add or remove a CreditCard (or BankAccount) from the collection of billing details, the

version of the User is incremented.

If you want to disable automatic increment for a particular value-typed property or collection, map it with the optimistic-lock="false" attribute. The inverse attribute makes no difference here.

***Versioning without version numbers or timestamps***

If you don’t have version or timestamp columns, Hibernate can still perform automatic

versioning, but only for objects that are retrieved and modified in the same

persistence context (that is, the same Session).

You may enable this functionality by setting the optimistic-lock attribute on the class

mapping:

<class name="Item" table="ITEM" optimistic-lock="all">

<id .../>

...

</class>

The following SQL is now executed to flush a modification of an Item instance:

update ITEM set ITEM\_PRICE='12.99'

where ITEM\_ID=123

and ITEM\_PRICE='9.99'

and ITEM\_DESCRIPTION="An Item"

and ...

and SELLER\_ID=45

Hibernate lists all columns and their last known nonstale values in the WHERE clause of the SQL statement. If any concurrent transaction has modified any of these values, or even deleted the row, this statement again returns with zero updated rows. Hibernate then throws a StaleObjectStateException.

***Versioning with Java Persistence***

To enable automatic versioning for a particular

entity, you need to add a version property or field:

@Entity

public class Item {

...

@Version

@Column(name = "OBJ\_VERSION")

private int version;

...

}

**Pessimistic locking**

You may get un-repeatable read for scalar queries:

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Item i = (Item) session.get(Item.class, 123);

String description = (String)

session.createQuery("select i.description from Item i" +

" where i.id = :itemid")

.setParameter("itemid", i.getId() )

.uniqueResult();

tx.commit();

session.close();

1. This unit of work executes two reads
2. There is a small window in this unit of work in which a concurrently running transaction may commit an updated item description between the two reads

How it can be fixed :

Use lock() method on the Hibernate Session:

Item i = (Item) session.get(Item.class, 123);

session.lock(i, LockMode.UPGRADE);

String description = (String)

session.createQuery("select i.description from Item i" +

" where i.id = :itemid")

.setParameter("itemid", i.getId() )

.uniqueResult();

Or

Item i = (Item) session.get(Item.class, 123, LockMode.UPGRADE);

...

Using LockMode.UPGRADE results in a pessimistic lock held on the database for the row(s) that represent the Item instance. Now no concurrent transaction can obtain a lock on the same data—that is, no concurrent transaction can modify the data between your two reads.

Some hibernate Locks values :

LockMode.NONE—Don’t go to the database unless the object isn’t in any cache, default one used by get and load.

LockMode.READ—Bypass all caches, and perform a version check to verify that the object in memory is the same version that currently exists in the database

LockMode.UPDGRADE—Bypass all caches, do a version check (if applicable), and obtain a database-level pessimistic upgrade lock, if that is supported.

LockMode.FORCE—Force an increment of the objects version in the database, to indicate that it has been modified by the current transaction.

LockMode.WRITE—Obtained automatically when Hibernate has written to a row in the current transaction. (This is an internal mode; you may not specify it in your application.)

***10.3 Nontransactional data access***

IN Hib The call to beginTransaction() translates into setAutoCommit(false) on the fresh JDBC Connection.

The term *nontransactional* data access means there are no explicit transaction boundaries,

Look at the following code, which accesses the database without transaction boundaries.

What happen when this is executed

Session session = sessionFactory.openSession();

Long generatedId = session.save(item);

session.close();

The answer to that question is, “It depends!” The JDBC specification doesn’t say anything about pending transactions when close() is called on a connection.

The oracle commit the transaction, any other DB may roll back it.

Ch 11 ConverSation

***Propagating the Hibernate Session***

Code involving multiple updates

public void endAuction(Item item) {

// Reattach item

itemDAO.makePersistent(item);

// Set winning bid

Bid winningBid = itemDAO.getMaxBid( item.getId() );

item.setSuccessfulBid(winningBid);

item.setBuyer( winningBid.getBidder() );

// Charge seller

Payment payment = new Payment(item);

paymentDAO.makePersistent(payment);

// Notify seller and winner

...

}

The code in listing 11.1 doesn’t work. there is no transaction demarcation.

***Propagation through thread-local***

Session s = getSessionFactory().**getCurrentSession**();

If sessionFactory.getCurrentSession() is called for the first time in the current Java

thread, a new Session is opened and returned—you get a fresh persistence context.

All the data-access code that calls getCurrentSession() on the global shared

SessionFactory gets access to the same current Session—if it’s called in the

same thread.

The unit of work completes when the Transaction is committed (or

rolled back). Hibernate also flushes and closes the current Session and its persistence

context if you commit or roll back the transaction. The implication here is

that a call to getCurrentSession() after commit or rollback produces a new Session

and a fresh persistence context.

Internally, Hibernate binds the current Session to the currently running Java

thread.

You have to enable this binding in your Hibernate configuration by setting

the hibernate.current\_session\_context\_class property to thread.

public void endAuction(Item item) {

try {

// Begin unit of work

sf.getCurrentSession().beginTransaction();

// Reattach item

itemDAO.makePersistent(item);

// Set winning bid

Bid winningBid = itemDAO.getMaxBid( item.getId() );

item.setWinningBid(winningBid);

item.setBuyer( winningBid.getBidder() );

// Charge seller

Payment payment = new Payment(item);

paymentDAO.makePersistent(payment);

// Notify seller and winner

...

// End unit of work

sf.getCurrentSession().getTransaction().commit();

} catch (RuntimeException ex) {

try {

sf.getCurrentSession().getTransaction().rollback();

} catch (RuntimeException rbEx) {

log.error("Couldn't roll back transaction," rbEx);

}

In ItemDAO class

public class ItemDAO {

public Bid getMaxBid(Long itemId) {

Session s = getSessionFactory().getCurrentSession();

return (Bid) s.createQuery("...").uniqueResult();

}

...

}

Same in JPA \EJB

|  |  |
| --- | --- |
| JPA | EJB |
| public void endAuction(Item item) throws Exception {  try {  // Begin unit of work  utx.begin();  // Reattach item  itemDAO.makePersistent(item);  // Set winning bid  Bid winningBid = itemDAO.getMaxBid( item.getId() );  item.setWinningBid(winningBid);  item.setBuyer( winningBid.getBidder() );  // Charge seller  Payment payment = new Payment(item);  paymentDAO.makePersistent(payment);  // Notify seller and winner  ...  // End unit of work  utx.commit();  } catch (Exception ex) {  try { | @TransactionAttribute(TransactionAttributeType.REQUIRED)  public void endAuction(Item item) {  // Reattach item  itemDAO.makePersistent(item);  // Set winning bid  Bid winningBid = itemDAO.getMaxBid( item.getId() );  item.setWinningBid(winningBid);  item.setBuyer( winningBid.getBidder() );  // Charge seller  Payment payment = new Payment(item);  paymentDAO.makePersistent(payment);  // Notify seller and winner  ...  } |

the ItemDAO and PaymentDAO classes, which internally use getCurrentSession(), are

unchanged in both of them.

new persistence context begins when getCurrentSession() is

called for the first time in one of the DAO classes. The current Session is bound

automatically to the current JTA system transaction. When the transaction completes,

either through commit or rollback, the persistence context is flushed and

the internally bound current Session is closed.

The current Session is bound to the transaction that is started for the endAuction()

method, and it’s flushed and closed when this method returns. All code

that runs inside this method and calls sessionFactory.getCurrentSession()

gets the same persistence context.

If you compare this example with the first nonworking example, listing 11.1,

you’ll see that you had to add only some annotations to make it work. The

@TransactionAttribute is even optional—it defaults to REQUIRED. This is why

EJB 3.0 offers a *simplified programming model*.

***Conversations with Hibernate***

How can you make the conversation atomic?

The conversation spans several persistence contexts and several database transactions. But this isn’t the scope of a of work from the point of view of the application user; she considers the conversation

to be an atomic group of operations that either all fail or all succeed. In the current conversation this isn’t a problem, because you modify and persist data only in the last (second) step. Any conversation that only reads data and delays all reattachment of modified objects until the last step is automatically atomic and

can be aborted at any time.

If a conversation reattaches and commits modifications to the database in an intermediate step, it’s no longer atomic.

One solution is to not flush the persistence contexts on commit—that is, to set a FlushMode.MANUAL on a Session that isn’t supposed to persist modifications (of course, not for the last step of the conversation). Another option is to use *compensation* actions that *undo* any step that made permanent changes, and to call the appropriate compensation actions when the user aborts the conversation.

Detached object conversation – getAuction begins the conversation and endAucton ends it.

We save changes at the last point in endAuctions

public class ManageAuction {

public Item getAuction(Long itemId) ...

...

public void endAuction(Item item) {

Session s = sf.getCurrentSession();

s.beginTransaction();

// Reattach item

s.update(item);

// Set winning bid

// Charge seller

// Notify seller and winner

...

s.getTransaction().commit();

}

}

Rest does not seems important, so leaving

Optimizing fetching and caching

HQL is commonly used for object retrieval, not for updating, inserting, or

deleting data.

Query q = session.createQuery(

"from User as u where u.firstname = :fname"

);

q.setString("fname", "John");

List result = q.list();

After preparing query q, you bind a value to the named parameter :fname. The

result is returned as a List of User objects.

***Querying with a criteria***

Criteria criteria = session.createCriteria(User.class);

criteria.add( Restrictions.like("firstname", "John") );

List result = criteria.list();

***Querying by example***

Criteria criteria = session.createCriteria(User.class);

User exampleUser = new User();

exampleUser.setFirstname("John");

criteria.add( Example.create(exampleUser) );

criteria.add( Restrictions.isNotNull("homeAddress.city") );

List result = criteria.list();

A typical use case for query by example is a search screen that allows users to

specify a range of different property values to be matched by the returned result

set.

***The lazy default fetch plan***

Hibernate defaults to a *lazy* fetching strategy for all entities and collections. This

means that Hibernate by default loads only the objects you’re querying for

Item item = (Item) session.load(Item.class, new Long(123));

What is available in memory after the load() operation isn’t a

persistent item object. Even the SQL that loads an Item isn’t executed. Hibernate

created a *proxy* that looks like the real thing.

***Understanding proxies***

A proxy is a placeholder that triggers the loading of the real object when it’s accessed for the first time:

Item item = (Item) session.load(Item.class, new Long(123));

item.getId();

item.getDescription(); // Initialize the proxy

The third line in this example triggers the execution of the SQL that retrieves an

Item into memory. As long as you access only the database identifier property, no

initialization of the proxy is necessary.

(Note that this isn’t true if you map the

identifier property with direct field access; Hibernate then doesn’t even know that

the getId() method exists. If you call it, the proxy has to be initialized.)

A proxy is useful if you need the Item only to create a reference, for example:

Item item = (Item) session.load(Item.class, new Long(123));

User user = (User) session.load(User.class, new Long(1234));

Bid newBid = new Bid("99.99");

newBid.setItem(item);

newBid.setBidder(user);

session.save(newBid);

You first load two objects, an Item and a User. Hibernate doesn’t hit the database

to do this: It returns two proxies. This is all you need, because you only require

the Item and User to create a new Bid. The save(newBid) call executes an INSERT

statement to save the row in the BID table with the foreign key value of an Item

and a User—this is all the proxies can and have to provide. The previous code

snippet doesn’t execute any SELECT!

If you call get() instead of load() you trigger a database hit and no proxy is

returned. The get() operation always hits the database (if the instance isn’t

already in the persistence context and if no transparent second-level cache is

active) and returns null if the object can’t be found.

In JPA

Item item = em.find(Item.class, new Long(123));

Item itemRef = em.getReference(Item.class, new Long(1234));

Find = get

Getrefernce = load

Because Hibernate proxies are instances of runtime generated subclasses of

your entity classes, you can’t get the class of an object with the usual operators. This is where the helper method HibernateProxyHelper.getClassWithoutInitializingProxy(

o) is useful.

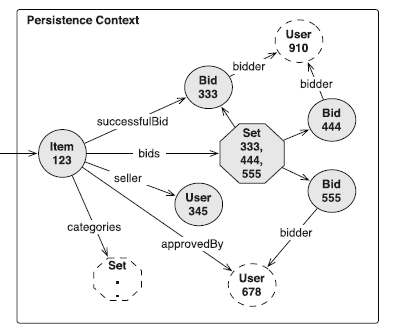
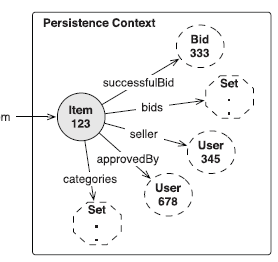
Proxy and association relation ship

**Proxy Vs No Proxy**

Let’s assume you have an Item instance into memory, either by getting it explicitly

or by calling one of its properties and forcing initialization of a proxy. Your

persistence context now contains a fully loaded object, as shown in figure below



Note : the figure on the right is for eagar fetching not for no proxy.

In lazy loading associated entity objects are *not* loaded right away; the proxies carry the identifier values only.

The identifier values are all foreign key columns in the item’s row.

Proxy and collection

Collections also aren’t loaded right away, but we use the term *collection wrapper* to

describe this kind of placeholder. Internally, Hibernate has a set of smart collections

that can initialize themselves on demand. Hibernate replaces your collections

with these; that is why you should use collection interfaces only in your

domain model.

A proxy is initialized if you call any method that is not the identifier getter

method, a collection is initialized if you start iterating through its elements or if

you call any of the collection-management operations, such as size() and contains().

Hibernate provides an additional setting that is mostly useful for large collections; they can be mapped as *extra* lazy. For example, consider the collection

of bids of an Item:

<class name="Item" table="ITEM">

...

<set name="bids"

lazy="extra"

inverse="true">

<key column="ITEM\_ID"/>

<one-to-many class="Bid"/>

</set>

</class>

The collection is no longer initialized if you call size(), contains(), or isEmpty()—the database is queried to retrieve the necessary information.

In JPA

A Hibernate extension annotation enables the same optimization:

@OneToMany

@org.hibernate.annotations.LazyCollection(

org.hibernate.annotations.LazyCollectionOption.EXTRA

)

private Set<Bid> bids = new HashSet<Bid>();

***Disabling proxy generation***

You can disable proxy generation for a particular entity class with the lazy="false" attribute in XML mapping metadata:

<class name="User"

table="USERS"

lazy="false">

...

</class>

The JPA standard doesn’t require an implementation with proxies; the word proxy doesn’t even appear in the specification.

Hibernate *is* a JPA provider that relies on proxies by default, so the switch that disables Hibernate proxies is available as a vendor extension:

@Entity

@Table(name = "USERS")

@org.hibernate.annotations.Proxy(lazy = false)

public class User { ... }

Disabling proxy generation for an entity has serious consequences

A load() of a User object can’t return a proxy

However, disabling proxies also has consequences for all associations that reference the entity. For example, the Item entity has a seller association to a User. Consider the following operations that retrieve an Item:

Item item = (Item) session.get(Item.class, new Long(123));

Item item = em.find(Item.class, new Long(123));

In addition to retrieving the Item instance, the get() operation now also loads the linked seller of the Item; no User proxy is returned for this association

***Eager loading of associations and collections***

Hibernate is lazy by default. All associated entities and collections aren’t initialized if you load an entity object.

When we need eager fetching

U want that that a particular entity association or collection should *always* be loaded.

hit.

More important, you want a guarantee that, for example, you can access the seller of an Item if the Item instance is in detached state.

U have to define this fetch plan, the part of your object network that you want to always load into memory.

Let’s assume that you always require the seller of an Item. In Hibernate XML

mapping metadata you’d map the association from Item to User as lazy="false":

<class name="Item" table="ITEM">

...

<many-to-one name="seller"

class="User"

column="SELLER\_ID"

update="false"

not-null="true"

lazy="false"/>

...

</class>

The same “always load” guarantee can be applied to collections—for example, all

bids of an Item:

<class name="Item" table="ITEM">

...

<many-to-one name="seller" lazy="false" .../>

<set name="bids"

lazy="false"

inverse="true">

<key column="ITEM\_ID"/>

<one-to-many class="Bid"/>

</set>

...

</class>

**LazyInitializationException**

Other lazy mapped associations and collections (the bidder of each Bid

instance, for example) are again uninitialized and are loaded as soon as you

access them. Imagine that you close the persistence context after loading an Item.

You can now navigate, in detached state, to the seller of the Item and iterate

through all the bids for that Item. If you navigate to the categories this Item is

assigned to, you get a LazyInitializationException!

With annotations, you switch the FetchType of an entity association or a collection

to get the same result:

@Entity

public class Item {

...

@ManyToOne(fetch = FetchType.EAGER)

private User seller;

@OneToMany(fetch = FetchType.EAGER)

private Set<Bid> bids = new HashSet<Bid>();

...

}

***Lazy loading with interception (don’t see imp or good)***

***13.2Selecting a fetch strategy***

By default, Hibernate fetches associated objects and collections lazily whenever

you access them

Item item = (Item) session.get(Item.class, new Long(123));

You didn’t configure any association or collection to be nonlazy, and that proxies

can be generated for all associations. Hence, this operation results in the following

SQL SELECT:

select item.\* from ITEM item where item.ITEM\_ID = ?

You can see that the SELECT queries only the ITEM table and retrieves a particular

row. All entity associations and collections aren’t retrieved. If you access any proxied

association or uninitialized collection, a second SELECT is executed to retrieve

the data on demand.

***Prefetching data in batches***

*n+1 selects problem*.

If every entity association and collection is fetched only on demand, many additional SQL SELECT statements may be necessary to complete a particular procedure.

For example, consider the following query that retrieves all Item objects and accesses the data of each items seller:

List allItems = session.createQuery("from Item").list();

processSeller( (Item)allItems.get(0) );

processSeller( (Item)allItems.get(1) );

processSeller( (Item)allItems.get(2) );

You see one SQL SELECT to retrieve all the Item

objects, and an additional SELECT for *every* seller of an Item as soon as you process

it. All associated User objects are proxies

select items...

select u.\* from USERS u where u.USER\_ID = ?

select u.\* from USERS u where u.USER\_ID = ?

select u.\* from USERS u where u.USER\_ID = ?

*batch fetching*

Hibernate offers some algorithms that can prefetch User objects. The first optimization

we now discuss is called *batch fetching*, and it works as follows: If one proxy

of a User must be initialized, go ahead and initialize several in the same SELECT.

<class name="User"

table="USERS"

batch-size="10">

...

</class>

You’re telling Hibernate to prefetch up to 10 uninitialized proxies in a single SQL SELECT, if one proxy must be initialized.

The resulting SQL for the earlier query and procedure may now look as follows:

select items...

select u.\* from USERS u where u.USER\_ID in (?, ?, ?)

You defined the batch size as “up to 10.” If more than 10 items are returned,

you see how the second query retrieves 10 sellers in one batch. If the application

hits another proxy that hasn’t been initialized, a batch of another 10 is retrieved—

and so on,

Batch fetching is also available for collections:

<class name="Item" table="ITEM">

...

<set name="bids"

inverse="true"

batch-size="10">

<key column="ITEM\_ID"/>

<one-to-many class="Bid"/>

</set>

</class>

If you now force the initialization of one bids collection, up to 10 more collections

of the same type, if they’re uninitialized in the current persistence context,

are loaded right away:

select items...

select b.\* from BID b where b.ITEM\_ID in (?, ?, ?)

***Prefetching collections with subselects***

Lets consider below :

List allItems = session.createQuery("from Item").list();

processBids( (Item)allItems.get(0) );

processBids( (Item)allItems.get(1) );

processBids( (Item)allItems.get(2) );

with batch fetching we have to set a blind guess in num of batch size.

you’d need to figure out an optimum batch size by trial. A much better optimization is *subselect fetching* for this collection mapping:

<class name="Item" table="ITEM">

...

<set name="bids"

inverse="true"

fetch="subselect">

<key column="ITEM\_ID"/>

<one-to-many class="Bid"/>

</set>

</class>

Hibernate now initializes *all* bids collections for all loaded Item objects, as soon

as you force the initialization of one bids collection. It does that by rerunning the

first initial query (slightly modified) in a subselect:

select i.\* from ITEM i

select b.\* from BID b

where b.ITEM\_ID in (select i.ITEM\_ID from ITEM i)

***Eager fetching with joins***

***Consider the situchantion***

“Every time I need an Item, I also need the seller of that Item.”

<class name="Item" table="ITEM">

...

<many-to-one name="seller"

class="User"

column="SELLER\_ID"

update="false"

fetch="join"/>

</class>

Item item = (Item) session.get(Item.class, new Long(123));

This operation triggers the following SQL SELECT:

select i.\*, u.\*

from ITEM i

left outer join USERS u on i.SELLER\_ID = u.USER\_ID

where i.ITEM\_ID = ?

the seller is no longer lazily loaded on demand, but immediately. Hence, a fetch="join" disables lazy loading

Eager join v lazy = false

If you only enable eager fetching

with lazy="false", you see an immediate second SELECT. With fetch="join",

you get the seller loaded in the same single SELECT

on a collection

<class name="Item" table="ITEM">

...

<set name="bids"

inverse="true"

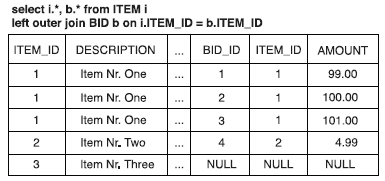
fetch="join">

<key column="ITEM\_ID"/>

<one-to-many class="Bid"/>

</set>

</class>



JPA

@ManyToOne(fetch = FetchType.EAGER)

private User seller;

@OneToMany(fetch = FetchType.EAGER)

private Set<Bid> bids = new HashSet<Bid>();

hibernate.max\_fetch\_depth to set the max depth of join

from 1 to 5, 0 for no join

***Optimizing fetching for secondary tables***

Can leave it

***Optimization guidelines,***

*What if we want fetching strategy not on global level but for current situation/query*

*We can not always use fetch = join in class level settings*

***The n+1 selects problem –*** is already covered and is same as batch fetching example.

Soluation

Betch fetching – will reduce the query to (n/10)+1

Subselect wil reduce it to two.

Join to One

However Fetching strategies in mapping metadata work on a global level

We don’t consider fetch="join" acommon optimization for collection mappings; you rarely need a fully initialized collection *all the time*. In addition to resulting in higher memory consumption,

every OUTER JOINed collection is a step toward a more serious Cartesian product

problem,

dynamic fetching strategy

List<Item> allItems =

session.createQuery("from Item i left join fetch i.bids")

.list();

List<Item> allItems =

session.createCriteria(Item.class)

.setFetchMode("bids", FetchMode.JOIN)

.list();

// Iterate through the collections...

Both queries result in a single SELECT that retrieves the bids for all Item instances

with an OUTER JOIN (as it would if you have mapped the collection with

join="fetch").

***The Cartesian product problem***

The opposite of the n+1 selects problem are SELECT statements that fetch *too much*

data. This Cartesian product problem always appears if you try to fetch several

“Parallel” collection

If we use fetch=”join” on global level then it can be result in lots of redundant results\object being created

transferred over the network.

So using fetch=”join” on global level is not a good optimization technique.

***Forcing proxy and collection initialization***

You can use the static method Hibernate.initialize() for manual initialization

of a proxy:

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Item item = (Item) session.get(Item.class, new Long(1234));

Hibernate.initialize( item.getSeller() );

tx.commit();

session.close();

processDetached( item.getSeller() );

Hibernate.initialize() may be passed a collection wrapper or a proxy.

Note that if you pass a collection wrapper to initialize(), it doesn’t initialize the target

entity objects that are referenced by this collection.

In the previous example, Hibernate.initalize( item.getBids() ) wouldn’t load all the Bid objects inside that collection. It initializes the collection with proxies of Bid objects!

For this case use\prefer a dynamic fetch with HQL or Criteria.f

***Optimization step by step***

Consider switching to lazy="false" (or FetchType.EAGER) on many-to-one,one-to-one, and (sometimes) collection mappings.

Keep in mind that more than one eagerly fetched collection per persistent

class creates a product. Evaluate whether your use case can benefit from

prefetching of collections, with batches or subselects. Use batch sizes

between 3 and 15.

***Caching fundamentals***

A major justification for our claim that applications using an object/relational

persistence layer are expected to outperform applications built using direct JDBC

is the potential for caching.

Caching is all about performance optimization, so naturally it isn’t part of the

Java Persistence or EJB 3.0 specification. Every vendor provides different solutions

for optimization, in particular any second-level caching.

***Caching strategies and scopes***

three main types of cache

*Transaction scope cache*

*Process scope cache*

*Cluster scope cache*

***The Hibernate cache architecture***

* The first-level cache is the persistence context cache.

This is a mandatory first-level cache that also guarantees the scope of object and database identity

* The second-level cache in Hibernate is pluggable and may be scoped to the process or cluster.

This is a cache of state (returned by value), not of actual persistent instances.

* Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature; it requires

two additional physical cache regions that hold the cached query results

and the timestamps when a table was last updated.

***The Hibernate second-level cache***

***CH 14***

***Creating and running queries***

Hibernate Query Language (HQL),

session.createQuery("from Category c where c.name like 'Laptop%'");

entityManager.createQuery(

"select c from Category c where c.name like 'Laptop%'");

Criteria API for *query by criteria* (QBC) and *query by example* (QBE):

session.createCriteria(Category.class)

.add( Restrictions.like("name", "Laptop%") );

Direct SQL with or without automatic mapping of resultsets to objects:

createSQLQuery() is used to create an SQL query using the native syntax of the

underlying database:

Query sqlQuery =

session.createSQLQuery(

"select {user.\*} from USERS {user}"

).addEntity("user", User.class);

Creating the query obj

Query hqlQuery = session.createQuery("from User");

Criteria crit = session.createCriteria(User.class);

***Paging the result***

Query query =

session.createQuery("from User u order by u.name asc");

query.setMaxResults(10);

***Considering parameter binding***

With named parameters, you can rewrite the query as

String queryString =

"from Item item where item.description like :search";

The colon followed by a parameter name indicates a named parameter. Then,

bind a value to the search parameter:

Query q = session.createQuery(queryString)

.setString("search", searchString);

In JPA

Query q = em.createQuery(queryString)

.setParameter("search", searchString)

.setParameter("minDate", mDate, TemporalType.DATE);

***Using Hibernate parameter binding***

Item item = new Item();

item.setSeller(seller);

item.setDescription(description);

String queryString = "from Item item"

+ " where item.seller = :seller and"

+ " item.description like :desccription";

session.createQuery(queryString).setProperties(item);

***Using positional parameters***

String queryString = "from Item item"

+ " where item.description like ?"

+ " and item.date > ?";

Query q = session.createQuery(queryString)

.setString(0, searchString)

.setDate(1, minDate);

Java Persistence also supports positional parameters:

String queryString = "from Item item"

+ " where item.description like ?1"

+ " and item.date > ?2";

Query q = em.createQuery(queryString)

.setParameter(1, searchString)

.setParameter(2, minDate, TemporalType.DATE);

You can disable flushing of the persistence context with setFlushMode() on a

Session or EntityManager. Or, if you want to disable flushing only before a particular

query, you can set a FlushMode on the Query (Hibernate and JPA) object:

Query q = session.createQuery(queryString)

.setFlushMode(FlushMode.COMMIT);

Criteria criteria = session.createCriteria(Item.class)

.setFlushMode(FlushMode.COMMIT);

Hibernate won’t flush the persistence context before executing any of these

queries.

You can control how long a query is allowed to run by setting a *timeout*:

Query q = session.createQuery("from Item")

.setTimeout(60); // 1 minute

The JDBC fetch size is an optimization hint for the database driver; it may not

result in any performance improvement if the driver doesn’t implement this functionality.

If it does, it can improve the communication between the JDBC client

and the database, by retrieving many rows in one batch when the client operates

on a query result (that is, on a ResultSet).

Query q = session.createQuery("from Item")

.setFetchSize(50);

***Executing a query***

***Listing all results***

In Hibernate, the list() method executes the query and returns the results as a

java.util.List:

List result = myQuery.list();

The Criteria interface also supports this operation:

List result = myCriteria.list();

***Iterating through the results***