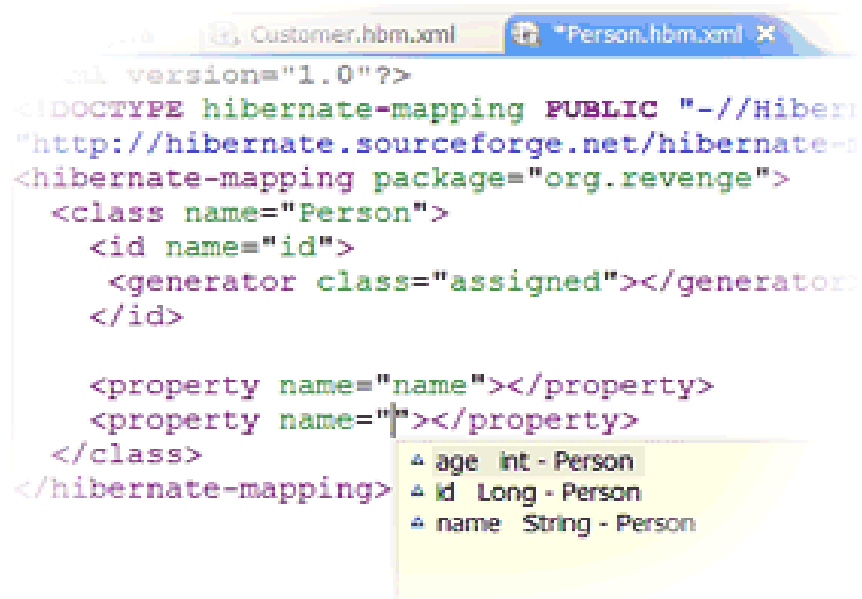


Guida Introduttiva ad Hibernate



```
Customer.hbm.xml *Person.hbm.xml x
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate
"http://hibernate.sourceforge.net/hibernate-s
<hibernate-mapping package="org.revenge">
  <class name="Person">
    <id name="id">
      <generator class="assigned"></generator>
    </id>

    <property name="name"></property>
    <property name="age"></property>
  </class>
</hibernate-mapping>
```

- ▲ age Int - Person
- ▲ id Long - Person
- ▲ name String - Person

di [Leonardo Puleggi](#)

Indice generale

Introduzione.....	3
Basi.....	3
Esempi.....	5
Operazioni Fondamentali.....	5
Esempio_01.....	7
Esempio_02.....	10
Esempio_03.....	12
Esempio_04.....	14
Esempio_05.....	15
Collegamenti Utili	18

Introduzione

Da Wikipedia: “**Hibernate** (spesso chiamato **H8**) è una soluzione *Object-relational mapping* (ORM) per il linguaggio di programmazione Java.

È un software free, open source e distribuito sotto licenza LGPL. Hibernate fornisce un framework semplice da usare, che mappa un modello di dominio orientato agli oggetti in un classico database relazionale.

Hibernate non si occupa solo di mappare dalle classi Java in tabelle del database (e da tipi Java ai tipi SQL), ma fornisce anche degli aiuti per le query di dati, il recupero di informazioni e riduce significativamente il tempo che altrimenti sarebbe speso lavorando manualmente in SQL e con JDBC.”

Da questa definizione abbiamo chiaro che Hibernate si pone in un fascia di prodotti già molto ampia; ricordiamo ad esempio Torque, SimpleORM , Ibatis e altri. Perché allora perdere tempo su un strumento quando magari se ne conosce già qualcuno dei precedenti oppure il mapping tra database e oggetti del dominio è gestito manualmente?

Semplice: perché Hibernate è lo standard *de facto* per quanto riguarda gli ORM (in quanto è il motore della persistenza di un prodotto quale JBoss) e perché semplifica enormemente operazioni tediose come scrivere codice per interagire con un database.

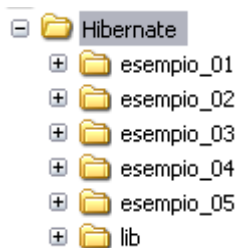
Hibernate, dunque, va usato sempre ed ovunque??Ovviamente no. Vanno valutate una serie di considerazioni quali: la possibilità di cambiare DBMS, l'uso API standard o aspetti specifici del DBMS e altre; ovviamente queste considerazioni non ci riguardano in questa che è una guida solo introduttiva e vedremo solo come usare Hibernate su esempi molto semplici, che però permetteranno di concentrarsi meglio sugli aspetti che ci interessano.

L'approccio che seguiremo infatti è di proporre esempi via via più complessi seguendo un approccio *divide et impera* che, secondo noi, semplificherà l'apprendimento.

Basi

In questi esempi useremo Hibernate per applicazioni esclusivamente testuali riducendo al massimo qualsiasi aspetto che non sia legato propriamente alla persistenza; per questo motivo gli esempi saranno forniti in cartelle in cui ci sarà un file per eseguire la compilazione da riga di comando(sia per Windows che per Linux), in modo da eliminare ogni complicazione o semplificazione apportata da un qualunque IDE.

Per prima cosa ovviamente dobbiamo scaricare gli esempi :); non è necessario scaricare il pacchetto completo di Hibernate perché le librerie importanti per questi esempi sono già inserite. Scompattando l'archivio in una directory otterremo una struttura come quella in figura.



*Illustrazione 1:
Directory degli
esempi*

Gli esempi sono numerati in base alla difficoltà.

Attenzione : Non copiate le definizioni di classi da un esempio ad un altro; negli esempi vengono infatti definiti via via nuovi metodi o altre parti per la maggiore difficoltà.

Nella cartella lib sono inserite due cartelle: una con le librerie fondamentali per Hibernate (il minimo indispensabile per funzionare) e l'altra con il driver JDBC per il DBMS scelto (in questo caso MySQL).

Attenzione : Nel caso doveste usare un altro tipo di DBMS i cambiamenti da effettuare sono quelli generali (indicati sotto, quali ad esempio l'ip del server l'user id e la password per la connessione):

1. Modificare il CLASSPATH definito nei file di esecuzione(il file esegui.bat o esegui.sh) relativo al driver del DBMS
2. Modificare la proprietà connection.driver_class nel file hibernate.cfg.xml nella cartella bin; per vedere quale è il nome della classe si consiglia di leggere la documentazione del proprio driver.
3. Modificare la proprietà connection.url nel file hibernate.cfg.xml nella cartella bin; si consiglia sempre di leggere la documentazione del proprio driver.
4. Modificare la proprietà dialect nel file hibernate.cfg.xml nella cartella bin; si consiglia sempre di leggere la documentazione del proprio driver.

In ogni esempio abbiamo una cartella src, che contiene il codice necessario per l'esempio e una cartella bin che contiene gli eseguibili, cioè i punto class, più i file di configurazione di Hibernate e di Log4J.

I passi per rendere operativi questi esempi nel caso si disponga di un database MySQL sono molto semplici:

1. Creare uno schema chiamato Hibernate e caricare il file che contiene le istruzioni SQL per creare le tabelle e inserire dei valori.
2. Modificare l'indirizzo ip del server nella proprietà connection.url nel file hibernate.cfg.xml
3. Modificare l'user id con cui connettersi al DBMS specificando la proprietà connection.username
4. Modificare la password con cui connettersi al DBMS specificando la proprietà connection.password

Prima di vedere cosa bisogna definire vediamo cosa sia possibile fare con strumenti quali Hibernate; vediamo come sia possibile fare le tipiche operazioni CRUD relativamente ad oggetti del dominio.

Otteniamo la lista di oggetti del database che soddisfano certi requisiti :

```
Query q = session.createQuery("from Dipartimento");  
List<Dipartimento> result=q.list();
```

Creiamo un oggetto del dominio e salviamolo nel database :

```
Dipartimento d=new Dipartimento(nome,sede);  
session.persist(d);
```

Carichiamo un oggetto salvato nel database:

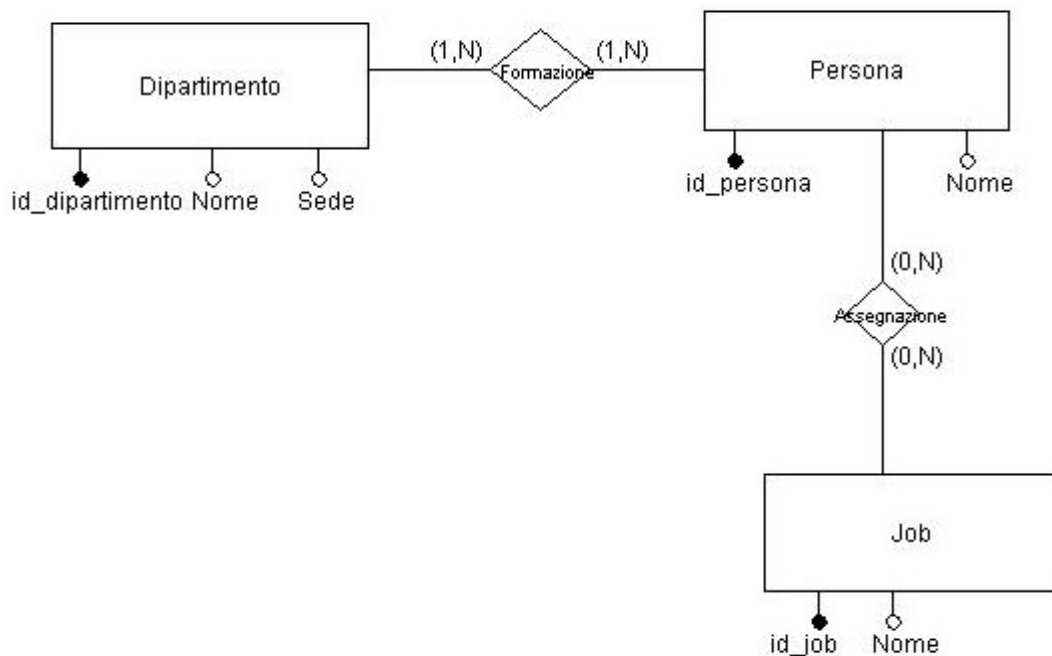
```
Dipartimento result=null;
result=(Dipartimento)session.load(Dipartimento.class,id);
```

Esempi

Vediamo ora singolarmente i vari esempi partendo ovviamente dai più semplici fino ad arrivare a quelli più complessi (in questo articolo ovviamente vedremo solo problematiche molto semplici lasciando eventuali approfondimenti al lettore).

In tutti gli esempi ipotizziamo di voler realizzare una applicazione che abbia bisogno di interagire con una base di dati che rappresenta in modo banalissimo una agenzia che ha un certo numero di Dipartimenti in cui lavorano certe Persone che hanno determinati compiti.

Lo schema di questa base di dati è banalmente:



Come è possibile vedere dallo schema e successivamente dal codice per ogni Entità saranno memorizzate pochissime informazioni proprio per focalizzarci solo sulla persistenza.

Operazioni Fondamentali

Prima di cominciare ad illustrare gli esempi è necessario identificare quali file e/o azioni saranno necessari per utilizzare Hibernate.

Come vedremo in tutti gli esempi abbiamo bisogno di almeno:

1. la classe che mappa le entità in oggetti
2. il file di mapping che esprime la corrispondenza tra oggetti e le entità
3. un file di configurazione di Hibernate

Il primo di dei file richiesti non è altro che un Plain Old Java Object, ovvero un oggetto come ne abbiamo sempre definiti con una sola particolarità; è conveniente specificare (ma Hibernate riesce a lavorare anche senza):

- **un campo id**, che rappresenterà il valore del campo id corrispondente nel database quando l'oggetto viene caricato
- **un costruttore no-args**, che Hibernate sfrutterà tramite la reflection per caricare le entità dal database

Il secondo file, come detto, rappresenta il mapping tra Entità ed Oggetti e verrà trattato ed affrontato gradualmente durante gli esempi.

Il terzo file serve a configurare Hibernate e sarà lo stesso per tutti gli esempi, quindi verrà trattato subito. Esso permette di definire quale DBMS si sta usando, quale dialetto (per sfruttare a fondo le capacità dei diversi DBMS), l'indirizzo IP del server e la porta, il nome utente e la password ed infine permette di specificare quali file devono essere usati per il mapping (possono essere più d'uno).

Le altre opzioni non ci interessano in questa guida introduttiva, quindi si rimanda alla Hibernate reference [[1a](#)] per ulteriori dettagli.

Vediamo ora il file hibernate.cfg.xml che si trova in ogni cartella bin degli esempi.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://IP:PORTA/DB</property>
    <property name="connection.username">USER</property>
    <property name="connection.password">PASSWORD</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">2</property>
    <!-- SQL dialect -->
    <property
name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>
    <!-- Disable the second-level cache -->
    <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">>false</property>
    <mapping resource="it/demo/dominio/Dipartimento.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Ovviamente per far funzionare gli esempi bisognerà modificare opportunamente tutti i parametri relativi al DBMS usato come specificato in precedenza.

Esempio_01

In questo primo esempio cercheremo di fare l'operazione più semplice ovvero ricevere dal database tutte le Entità memorizzate (ottenere una lista di oggetti corrispondenti ad una 'select * from ..'); per fare questa operazione dobbiamo definire i due file di cui abbiamo parlato in precedenza.

Partiamo dal POJO dell'entità (qui il Dipartimento ma il procedimento è lo stesso) che è sicuramente più familiare:

```
package it.demo.dominio;

public class Dipartimento {
    private Long id;
    private String nome;
    private String sede;

    public Dipartimento() {
    }

    public Dipartimento(String nome, String sede) {
        this.nome = nome;
        this.sede = sede;
    }

    public Long getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSede() {
        return sede;
    }

    public void setSede(String sede) {
        this.sede = sede;
    }

    public String toString() {
        ....
    }
}
```

Come è possibile vedere abbiamo definito il campo id, il costruttore no-args e i vari metodi setter e getter per ogni proprietà; niente di impossibile

Passiamo ora a definire il file per il mapping, che è certamente l'operazione più importante e "nuova".

Hibernate fornisce per questo file di configurazione, che è certamente il più importante, una espressività enorme, che in questi esempi può solo essere sfiorata senza scendere nel dettaglio; qualora servissero chiarimenti oppure in caso di approfondimenti si consiglia di cercare nella ottima Hibernate Reference[1].

Nel file di configurazione di Hibernate avevamo specificato la posizione di uno o più file di mapping; Hibernate sa, quindi, che nella posizione `it/demo/dominio/Dipartimento.hbm.xml` troverà un file come questo:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Dipartimento" table="dipartimento" lazy="false">
        <id name="id" column="id_dipartimento">
            <generator class="native"/>
        </id>
        <property name="nome"/>
        <property name="sede"/>
    </class>

</hibernate-mapping>
```

Escluse le righe di intestazione il resto del file è fondamentale ai fini della persistenza. Come possiamo vedere definiamo un nodo **hibernate-mapping** che è relativo ad un database('hibernate') e ad un package di default per la classi che appartengono a quel mapping.

All'interno di un mapping è possibile definire varie **classi**; per la classe Dipartimento specifichiamo che essa dovrà corrispondere ad una tupla della tabella 'dipartimento' e che vogliamo che gli oggetti siano caricati subito(lazy="false", vedremo dopo in dettaglio cosa significhi).

All'interno del nodo di una classe possiamo specificare il **mapping** tra le colonne del database e gli attributi di istanza della classe(qui in realtà non dobbiamo definire niente se non le proprietà che vogliamo siano settate nell'oggetto caricato dal database perché i nomi degli attributi di istanza sono uguali ai nomi delle colonne del database).

L'ultima cosa interessante è il campo **id**, che deve derivare dal valore della colonna `id_dipartimento`(qui infatti i nomi dell'attributo e della colonna sono diversi e quindi bisogna specificare questa corrispondenza) e che per generare id univoci nel database si sfrutta il **generatore** `native`(esistono diversi tipi di generatori, dai nativi a quelli corrispondenti a tecniche di High-Low[1]).

Finalmente abbiamo finito di definire tutti i file che ci servono per il mapping; ora possiamo vedere Hibernate in azione; quelli che seguono sono una serie di metodi che sfruttano alcune delle possibilità offerte da Hibernate tramite la propria API(per maggior dettaglio si rimanda sempre alla Hibernate Reference[1])

```
public static List<Dipartimento> listaTuttiDipartimenti() {
    List<Dipartimento> result=null;

    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery("from Dipartimento");
        result=q.list();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null)
            tx.rollback();
        throw he;
    }
}
```



```

    }
    finally {
        session.close();
    }

    return result;
}

/**Crea un oggetto a partire dall'identificativo o solleva un'eccezione
 *
 */
public static Dipartimento cercaDipartimento(long id) {
    Dipartimento result=null;

    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        result=(Dipartimento)session.load(Dipartimento.class,id);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null)
            tx.rollback();
        System.out.println("Non e' stato trovato l'oggetto con questo
id");
    }
    finally {
        session.close();
    }
    return result;
}

/**Crea un Dipartimento dati nome e sede e lo salva nel database
 */
public static Dipartimento creaDipartimento(String nome,String sede) {
    Dipartimento d=new Dipartimento(nome,sede);

    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.persist(d);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null)
            tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return d;
}
}

```

Tutti i metodi aprono una sessione tramite la quale possono eseguire una serie di azioni: il primo metodo crea una Query che riporterà una lista dei tutti i Dipartimenti presenti nel database; il secondo metodo prova a caricare il Dipartimento con un certo id; l'ultimo metodo permette di salvare molto semplicemente un oggetto nel database delegando a Hibernate la creazione di un id univoco, la esecuzione delle varie *insert* necessarie e le gestione degli errori.

L'output del programma di prova, che non fa altro che lanciare i tre metodi uno dopo l'altro sarà qualcosa del tipo:

```
Stampo tutti i Dipartimenti
Dipartimento : id=1, nome=programmazione 1, sede=Roma
Dipartimento : id=2, nome=programmazione 2, sede=Roma
Dipartimento : id=3, nome=personale, sede=Milano

Ora stampo solo quello che ha id=2 (Metodo classico)
Dipartimento : id=2, nome=programmazione 2, sede=Roma

Ora stampo solo quello che ha id=18(dovrebbe essere null)
Non e' stato trovato l'oggetto con questo id
Infatti e' nullo
Ora credo un Dipartimento con nome leo e sede roma
Dipartimento : id=4, nome=leo, sede=roma
Notare che l'id e' stato generato automaticamente
```

Esempio_02

Nel primo esempio abbiamo visto alcune operazioni molto semplici e che tutto sommato si ottengono semplicemente anche con JDBC; come sappiamo il vero problema tra database e linguaggio di programmazione ad oggetti è il cosiddetto problema della ??????.

Vediamo allora di aggiungere la navigabilità tramite riferimenti; vogliamo che una volta che sia caricato un Dipartimento dal database esso contenga una lista delle persone che vi lavorano. Questa operazione in JDBC standard richiederebbe di eseguire una query e analizzare il risultato per caricare ogni Persona in una lista. In Hibernate questo si fa molto più semplicemente; innanzitutto si deve aggiungere un attributo di istanza al Dipartimento, ovvero una lista di Persone che vi lavorano:

```
public class Dipartimento {
    private Long id;
    private String nome;
    private String sede;
    private Set persone;
    public Dipartimento() {
        persone=new HashSet();
    }

    public Dipartimento(String nome,String sede) {
        this.nome=nome;
        this.sede=sede;
        persone=new HashSet();
    }
}
```

Dobbiamo ovviamente creare anche la classe Persona e aggiungere il mapping tra la tabella Persone e la classe. La classe che rappresenta una Persona è banalmente uguale a quanto visto con il Dipartimento quindi non ci soffermeremo oltre; per quanto riguarda il mapping invece dobbiamo definire un nuovo mapping(e aggiungerlo nella lista dei mapping in hibernate.cfg.xml) e modificare quello già usato.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Dipartimento" table="dipartimento" lazy="false">
        <id name="id" column="id_dipartimento">
            <generator class="native"/>
        </id>
        <property name="nome"/>
        <property name="sede"/>
        <set name="persone" lazy="false">
            <key column="fk_dipartimento" not-null="true"/>
            <one-to-many class="Persona"/>
        </set>
    </class>

</hibernate-mapping>

```

Come è possibile vedere abbiamo aggiunto un set al mapping del Dipartimento specificando che le Persone vengano caricate subito (lazy="false"), quale è la chiave esterna e che si tratta di una relazione uno a molti con la classe Persona unidirezionale (nella reference vi è un capitolo intero dedicato al modo di specificare le varie associazioni [[1b](#)]).

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Persona" table="persona">
        <id name="id" column="id_persona">
            <generator class="native"/>
        </id>
        <property name="nome"/>
    </class>

</hibernate-mapping>

```

Il mapping tra la classe Persona e la tabella persone è molto banale e richiama il mapping originale che avevamo definito per il Dipartimento.

Ora possiamo sfruttare questi nuovi mapping per ottenere un livello di interazione maggiore; sfruttando infatti lo **stesso** codice che abbiamo visto prima per caricare la lista dei Dipartimenti e le altre opzioni, questa volta dal database verranno caricate automaticamente le Persone del Dipartimento, verranno salvate se si dovesse creare un nuovo Dipartimento con altre Persone; il tutto modificando **solo** i file di configurazione del mapping.

L'output dell'esempio_02 per quanto detto sarà qualcosa del tipo:

```

Stampo tutti i Dipartimenti
Dipartimento : id=1, nome=programmazione 1, sede=Roma
La lista delle persone che vi partecipano:[leonardo, ale]
Dipartimento : id=2, nome=programmazione 2, sede=Roma

```

```
La lista delle persone che vi partecipano:[leo, claudio]
Dipartimento : id=3, nome=personale, sede=Milano
La lista delle persone che vi partecipano:[alex, nazzarena]
Dipartimento : id=4, nome=leo, sede=roma
La lista delle persone che vi partecipano:[]

Ora stampo solo quello che ha id=2
Dipartimento : id=2, nome=programmazione 2, sede=Roma
La lista delle persone che vi partecipano:[leo, claudio]

Ora stampo solo quello che ha id=18(dovrebbe essere null)
Errore : non trovato dipartimento con id 18
Infatti e' nullo
Ora credo un Dipartimento con nome leo e sede roma
Dipartimento : id=5, nome=leo, sede=roma
La lista delle persone che vi partecipano:[]
Notare che l'id e' stato generato automaticamente
```

Esempio_03

Nell'esempio_02 per le Persone abbiamo definito un mapping semplice; abbiamo cioè caricato solo l'id e il nome. Possiamo però aggiungere la possibilità di sfruttare la relazione di appartenenza ad un Dipartimento anche nel verso opposto, ovvero da Persona al Dipartimento dove lavora(supposto unico per semplicità). Andremo quindi a definire un mapping bidirezionale tra Dipartimento e Persona. Per farlo innanzitutto dobbiamo aggiungere un attributo di istanza per il Dipartimento all'interno della classe Persona:

```
package it.demo.dominio;

public class Persona {
    ..
    private Dipartimento dipartimento;
    ...

    public Dipartimento getDipartimento() {
        return dipartimento;
    }

    public void setDipartimento(Dipartimento dipartimento) {
        this.dipartimento = dipartimento;
    }
}
```

Dobbiamo inoltre aggiungere il mapping bidirezionale tra Persona e Dipartimento; per farlo modifichiamo il file Persona.hbm.xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Persona" table="persona">
        <id name="id" column="id_persona">
```

```

        <generator class="native"/>
        </id>
        <property name="nome"/>
        <many-to-one name="dipartimento" column="fk_dipartimento" not-
null="true"/>

    </class>
</hibernate-mapping>

```

Come si può vedere è bastato mettere un elemento che rappresenta l'opposto dell'elemento che abbiamo nel mapping per il Dipartimento; per i dettagli si rimanda al capitolo della Hibernate reference ricco di esempi[[1b](#)].

Il programma di esempio dell'esempio_03 server a verificare la bidirezionalità della relazione; carica cioè il primo Dipartimento, prende il primo elemento della lista delle Persone che lavorano presso quel Dipartimento e verifica che quella Persona abbia il riferimento corretto al Dipartimento precedente.

Nell'esempio_03 è inoltre considerato un altro problema: le sessioni divise. Ipotizziamo di dover caricare un oggetto dal database e di dover usare questo oggetto in strati più alti della nostra applicazione. Usando JDBC una volta caricato l'oggetto dovremmo valutare attentamente come agire in una situazione del genere, con Hibernate invece possiamo semplicemente caricare l'oggetto, chiudere la sessione in cui abbiamo eseguito queste azioni, lavorare per un certo tempo con l'oggetto e poi aggiornare lo stato nel database in maniera che rimanga consistente. In Hibernate tutto questo è spaventosamente semplice:

```

Dipartimento d=creaDipartimento(".....","....."); //crea un nuovo Dipartimento
//ed esegue il codice per salvarlo nel database

//ipotizziamo che l'oggetto venga passato allo strato superiore
//e che la sessione sia chiusa
try {
    Thread.sleep(2000);
}
catch (Exception e) {
}

d.setNome("programmazione 3");
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    session.saveOrUpdate(d);
    tx.commit();
}
catch (HibernateException he) {
    if (tx!=null)
        tx.rollback();
    throw he;
}
finally {
    session.close();
}

```

L'output del programma fornito nell'esempio_03 sarà qualcosa del genere:

```
Stampo la lista delle persone del primo dipartimento  
[leonardo 'programmazione 1', ale 'programmazione 1']
```

Dato il primo dipartimento prendo la prima persona che vi lavora e verifico che questa persona abbia il riferimento corretto al dipartimento(cioe' la **navigabilita' e' bidirezionale**)

La prima persona e' leonardo

Il nome del dipartimento della prima persona: programmazione 1

La sede del dipartimento della prima persona: Roma

Ora facciamo la prova delle **due sessioni**, nella prima creo un oggetto Dipartimento e lo salvo nel database, faccio finta di eseguire operazioni in uno strato superiore e poi chiedo di aggiornarlo

Ora sto fermo per 2 secondi per far finta che l'oggetto subisca qualche elaborazione

Verifichiamo che esista il Dipartimento 'programmazione 3' con sede a roma dalla lista di tutti i dipartimenti

'programmazione 1' Roma

'programmazione 2' Roma

'personale' Milano

'leo' roma

'leo' roma

'programmazione 3' roma

Esempio_04

Come alcuni avranno notato lanciando gli esempi più volte sembra che vengano salvati nel database più oggetti con lo stesso nome. Questo ovviamente deriva dal codice di esempio che è banale e non considera la possibilità di dover analizzare prima il database alla ricerca dell'oggetto e solo nel caso questo non sia presente di crearlo. Nell'esempio_04 sfruttiamo Hibernate proprio per creare un Dipartimento a patto che non ne esista già uno con gli stessi parametri, che altrimenti sarà semplicemente caricato dal database.

Il codice per realizzare un controllo del genere è:

```
Dipartimento d=null;  
  
Session session = HibernateUtil.getSessionFactory().openSession();  
Transaction tx = null;  
try {  
    tx = session.beginTransaction();  
    List<Dipartimento> l=session.createQuery("from Dipartimento where  
nome=:nome and sede=:sede").  
setString("nome", nome).setString("sede", sede).list();  
    if (l.size()==0) {  
        d=new Dipartimento(nome, sede);  
        session.save(d);  
    }  
    else
```

```

        d=l.iterator().next();

        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null)
            tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return d;
}

```

Come possiamo vedere abbiamo finalmente avuto modo di usare delle query più complesse in cui è necessario indicare dei parametri attuali:

```

List<Dipartimento> l=
session.createQuery("from Dipartimento where nome=:nome and sede=:sede").
setString("nome",nome).setString("sede",sede).list();

```

Esempio_05

In questo ultimo esempio cercheremo di vedere come la configurazione di Hibernate possa portare ad una differenza sostanziale nelle prestazioni e nell'uso di questo strumento. Ipotizziamo di voler ottenere una lista delle Persone con i rispettivi lavori; dopo quanto visto sembrerebbe una operazione facile, quasi banale; e invece, proprio qui stanno le insidie.

Cominciamo innanzitutto definendo il mapping per i Job; non dedicheremo commenti al mapping perchè esso è molto simile a quanto visto finora:

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Job" table="job">
        <id name="id" column="id_job">
            <generator class="native"/>
        </id>
        <property name="lavoro" column="nome"/>
        <set name="persone" table="assegnazione_compiti" inverse="true">
            <key column="fk_job"/>
            <many-to-many column="fk_persona" class="Persona"/>
        </set>

    </class>

</hibernate-mapping>

```

Modifichiamo anche il mapping per Persona in modo da aver per ogni Persona la lista dei suoi lavori (ovviamente dobbiamo modificare anche la classe Persona, ma in modo del tutto uguale a quanto fatto negli esempi precedenti):

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping schema="hibernate" package="it.demo.dominio">

    <class name="Persona" table="persona">
        <id name="id" column="id_persona">
            <generator class="native"/>
        </id>
        <property name="nome"/>
        <set name="job" table="assegnazione_compiti" lazy="false">
            <key column="fk_persona"/>
            <many-to-many column="fk_job" class="Job" fetch="join"/>
        </set>
        <many-to-one name="dipartimento" column="fk_dipartimento" not-
null="true" fetch="join" lazy="false"/>
    </class>

</hibernate-mapping>

```

Finalmente possiamo vedere le differenze di cui parlavamo prima; per caricare una lista delle Persone con i rispettivi Job potremmo pensare di usare un codice del genere:

```
Query q = session.createQuery("from Persona");
```

Una sintassi del genere chiede a Hibernate di caricare tutte le Persone memorizzate nel database e visto che il mapping richiede che i job siano caricati insieme che le liste siano popolate con i rispettivi Job. Di seguito possiamo vedere l'output con abilitata la visualizzazione delle query necessarie per ricavare le informazioni (si tratta di un utile strumento di analisi per evidenziare degli errori e può essere abilitato tramite la proprietà `log4j.logger.org.hibernate.SQL` del file `log4j.properties`)

```

Stampo tutte le persone con i rispettivi job
Hibernate: select persona0_.id_persona ....
Hibernate: select job0_.fk_persona as fk1_1_ ....
Hibernate: select job0_.fk_persona as fk1_1_ ....
Hibernate: select job0_.fk_persona as fk1_1_ ....
Hibernate: select job0_.fk_persona as fk1_1_ ....
Hibernate: select job0_.fk_persona as fk1_1_ ....
leonardo
Job: [scrivere documentazione, scrivere codice]
ale
Job: [rispondere alle mail, dialogare con i clienti]

```

Come possiamo facilmente vedere dai log, Hibernate esegue una query per ottenere tutte le Persone e successivamente una per ogni Persona per ricercare i suoi Job. E' inutile dire che tale comportamento è inutile quanto dannoso, in quanto non sfrutta le caratteristiche dei DBMS attuali ed esegue un numero di query che è lineare con il numero di Persone.

E' possibile, ragionando un secondo, trovare la soluzione e riuscire ad applicarla; si tratta semplicemente di eseguire una query unica in cui ottengo sia la lista delle Persone che dei Job relativi. Hibernate si occuperà per me di ritornare una lista di Persone correttamente inizializzate

con i relativi Job senza dover analizzare autonomamente il risultato della query.
La query da eseguire è la seguente:

```
Query q = session.createQuery("from Persona as p left outer join fetch p.job");
```

Possiamo verificare come questa volta non ci sia bisogno di eseguire un numero molto alto di query, ma come ne basti semplicemente una.

```
Stampo tutte le persone con i rispettivi job  
Hibernate: select persona0_.id_persona as ....  
leonardo  
Job: [scrivere codice, scrivere documentazione]
```

Collegamenti Utili

[1] Hibernate Reference, [online](#)

- [1a] [Configuration](#)
- [1b] [Collection Mapping](#)