# Introduzione ad Hibernate

Testi di riferimento:

- Just Hibernate: A Lightweight Introduction to the Hibernate Framework ; Madhusudhan Konda, O'Reilly
- Java Persistence with Hibernate; Christian Bauer,Gavin King, Gary Gregory, Manning
- https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html

# Definizione

Framework open source ORM (Object Relation Mapping)
http://hibernate.org

- Persistenza, trasparente ed automatica, di oggetti di una applicazione Java nelle tabelle di un database relazionale (RDBMS)
- Utilizza metadati che descrivono la mappatura tra gli oggetti e il db
- Consente di eseguire query e gestire i risultati in modo semplificato

# Benefici

- Produttività: possibilità di concentrarsi sulla logica di business
- Manutenibilità: riduzione delle righe di codice (LOC)
- Performance: es. tuning del caching più efficiente e facile
- Indipendenza dal DBMS

# JDBC

## E jdbc? (Java DataBase Connectivity)
## Connettore + API

## Es. di persistenza in una tabella (JDBC)

| ID | TITOLO | REGISTA | SINOSSI |
|---|---|---|---|
| 1 | Dracula di Bram Stoker | Francis Ford Coppola | Per la falsa notizia della sconfitta... |
| 2 | Lo Squalo | Steven Spielberg | Ti farà perdere il gusto del bagno al mare! |

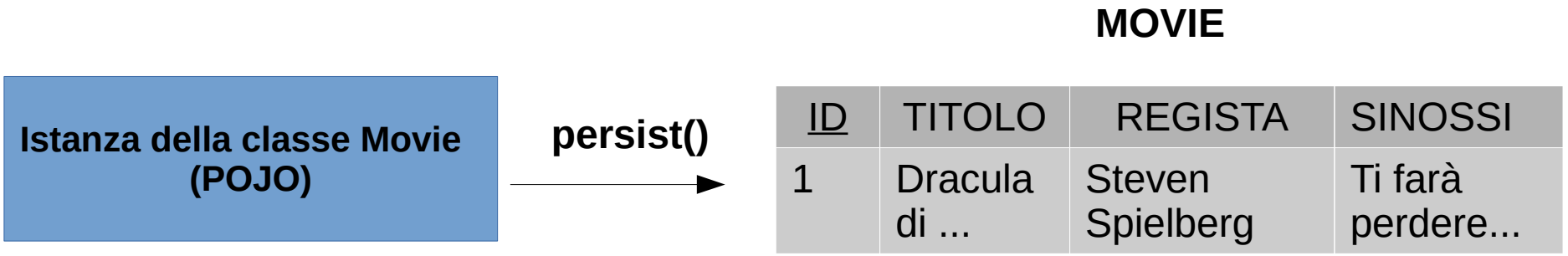
# 1. Creazione di una connessione

```
public class MovieManagerJDBC {

     private Connection connection = null;
     private String username = "federico";
     private String password = "password";
     private String url = "jdbc:mysql://localhost/corso_hibernate";
     private String driverClass = "com.mysql.jdbc.Driver";
     …


     private void createConnection() {
          try {
               Class.forName(driverClass).newInstance();
               connection = DriverManager.getConnection(url, username, password);
          } catch (Exception ex) {
               System.err.println("Exception while creating a connection:" +
ex.getMessage());
          }
          System.out.println("Connection created successfully");
     }
```

## 2. Utilizzo di PreparedStatement

```
private void persistMovie() {
          try {
               PreparedStatement pst = getConnection().prepareStatement(insertSql);

               pst.setInt(1, 1);
               pst.setString(2, "Dracula di Bram Stoker");
               pst.setString(3, "Francis Ford Coppola");
               pst.setString(4, "Per la falsa notizia della sconfitta e morte "
                         + "sul campo di Vlad III, la moglie Elisabetta si suicida... ");

               pst.execute();
               System.out.println("Movie persisted successfully!");

          } catch (SQLException ex) {
               System.err.println(ex.getMessage());
          }
     }
```

# Approccio ORM



# Hibernate – Passaggi fondamentali

- Configurazione della connessione del database

**hibernate.cfg.xml**

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/corso_hibernate</property>
    <property name="hibernate.connection.username">federico</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="current_session_context_class">thread</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <mapping resource="Movie.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

**oppure hibernate.properties**

```
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost:3307/JH
hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

**oppure tramite codice applicativo**

# Hibernate – Passaggi fondamentali

- Definizione della classe da salvare (POJO/AJO)

```
public class Movie {
     private int id = 0;
     private String  title = null;
     private String  synopsis = null;
     private String  director = null;

     public void setTitle(String title) {
          this.title = title;
     }

     public String getTitle() {
          return title;
     }

     public void setDirector(String director)
{
          this.director = director;
     }

     public String getDirector() {
          return director;
     }

 ...
```

# Hibernate - Passaggi fondamentali

- Definizione del mapping

**Movie.hbm.xml**

```
<hibernate-mapping>
    <class name="it.consoftinformatica.corsoHibernate.domain.Movie" table="MOVIES">
        <id name="id" column="ID">
            <generator class="assigned"/>
        </id>
        <property name="title" column="TITLE"/>
        <property name="director" column="DIRECTOR"/>
        <property name="synopsis" column="SYNOPSIS"/>
    </class>
</hibernate-mapping>
```

**Vedremo dopo un'alternativa (annotations)**

# Hibernate – Passaggi fondamentali

- Persistenza degli oggetti

```
private void init() {

          try {
               StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
                         .configure("hibernate.cfg.xml").build();
               Metadata metaData = new MetadataSources(standardRegistry).getMetadataBuilder().build();
               sessionFactory = metaData.getSessionFactoryBuilder().build();
          }
...


      private void persistMovie() {
           Movie movie = new Movie();
           movie.setId(2);
           movie.setDirector("Steven Spielberg");
           movie.setTitle("Lo Squalo");
           movie.setSynopsis("Ti farà perdere il gusto del bagno al mare!");
           Session session = sessionFactory.getCurrentSession();
           session.beginTransaction();
           session.save(movie);
           session.getTransaction().commit();
      }
```

# Hibernate – Passaggi fondamentali

- SessionFactory (org.hibernate.SessionFactory)

  thread-safe, rappresenta i mappaggi compilati per un database singolo, mantiene servizi usati da tutte le sezioni (es. cache secondo livello)

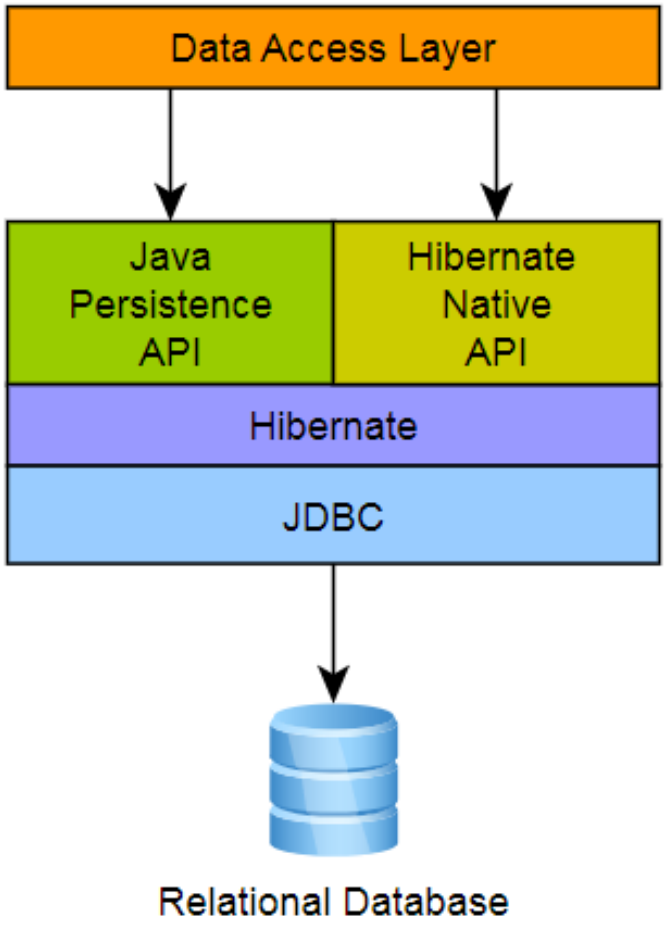
- Session (org.hibernate.Session)

  single-threaded, wrappa la java.sql.Connection, mantiene cache di primo livello, factory di Transaction

- Transaction (org.hibernate.Transaction)

  single-threaded, delimita transazioni

# JPA (Java Persistence API) ed Hibernate



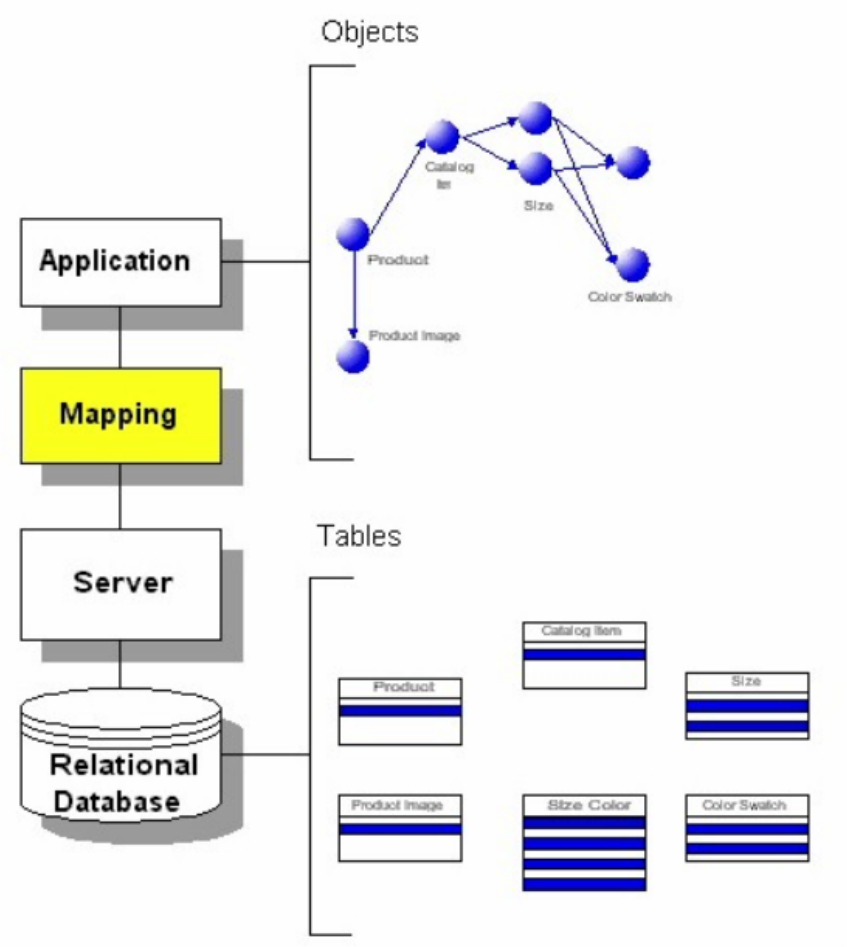

Hibernate può essere usato nativamente, ma è anche un JPA provider

hibernate.cfg.xml => persistence.xml

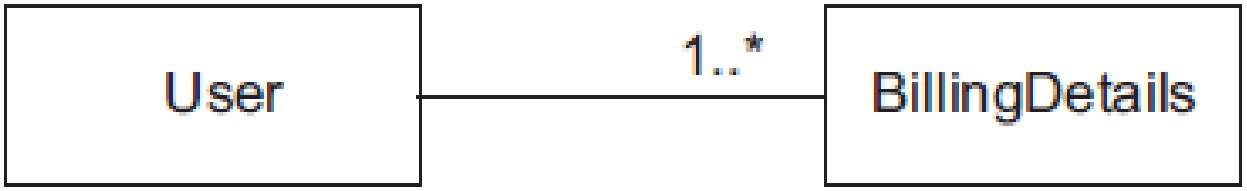
SessionFactory => EntityManagerFactory

Session => EntityManager

# The object/relational paradigm mismatch



# The object/relational paradigm mismatch



# The object/relational paradigm mismatch

```
public class User {

    String username;
    String address;
    Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
}


public class BillingDetails {

    String account;
    String bankname;
    User user;

    // Accessor  methods (getter/setter), business methods, etc.
}
```

# The object/relational paradigm mismatch

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS varchar(255) not null
);
```

```
create table BILLINGDETAILS (
    ACCOUNT varchar(15) not null primary key,
    BANKNAME varchar(255) not null,
    USERNAME varchar(15) not null,
    foreign key (USERNAME) references USERS
);
```

# The object/relational paradigm mismatch
# Granularità

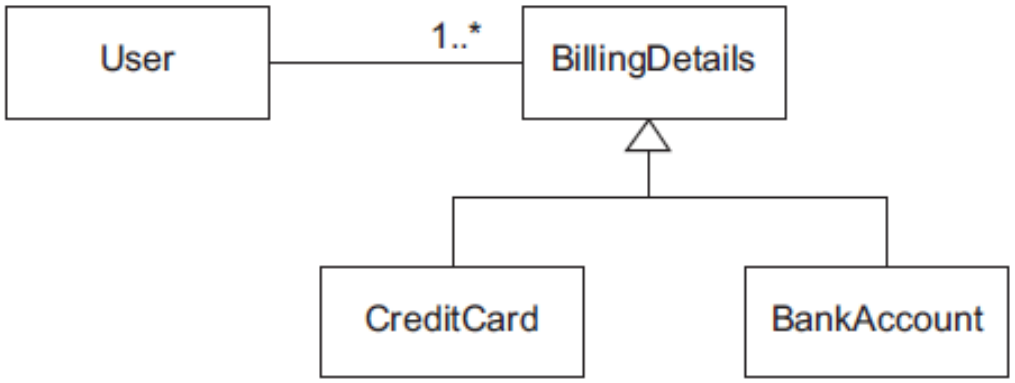

- Creare la nuova tabella ADDRESS?
- Aggiungere colonne alla tabella USER?
- Creare un nuovo tipo SQL (UDT)?

# The object/relational paradigm mismatch
# Granularità

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS_STREET varchar(255) not null,
    ADDRESS_ZIPCODE varchar(5) not null,
    ADDRESS_CITY varchar(255) not null
);
```

SQL ha solo due livelli di granularità: rischio di “appiattire” il modello orientato agli oggetti.

# The object/relational paradigm mismatch
# Ereditarietà




RDBMS non implementano ereditarietà

L'associazione user-billingdetails è polimorfica… vogliamo una query polimorfica (Definire una chiave esterna che si riferisce a tabelle multiple?)

# The object/relational paradigm mismatch
# Identità

JAVA:

- Identità tra due istanze  = =
- Uguaglianza tra due istanze  equals()

SQL:

- Chiave primaria

In applicazioni concorrenti multi-threaded istanze non identiche possono rappresentare lo stesso record del db

Come rappresentare nel model domain le chiavi primarie surrogate generate?

Problema rilevante nella gestione delle transazioni e del caching

# The object/relational paradigm mismatch
# Le associazioni

## Object references

## vs.

## foreign key-constrained column

Le reference hanno una direzione (puntatori), possono essere bidirezionali, si può “navigare” tra gli oggetti

Le chiavi esterne non sono direzionali, Il modello relazionale utilizza join e proiezioni per comporre i dati

# The object/relational paradigm mismatch
# Le associazioni

Le associazioni tra oggetti possono avere una molteplicità molti-a-molti

```
public class User {
    Set billingDetails;
}

public class BillingDetails {
    Set users;
}
```

Le associazioni tra tabelle sono sempre uno-a-molti o uno-a-uno

Per ottenere una relazione molti-a-molti nel modello relazionale bisogna introdurre un'altra tabella di link

```
create table USER_BILLINGDETAILS (
    USER_ID bigint,
    BILLINGDETAILS_ID bigint,
    primary key (USER_ID, BILLINGDETAILS_ID),
    foreign key (USER_ID) references USERS,
    foreign key (BILLINGDETAILS_ID) references BILLINGDETAILS
);
```

# The object/relational paradigm mismatch
# Navigazione dei dati

Accesso a dati (run-time):

someUser.getBillingDetails().iterator().next()

```
select * from USERS u
    left outer join BILLINGDETAILS bd
        on bd.USER_ID = u.ID
where u.ID = 123
```

# The object/relational paradigm mismatch
# Navigazione dei dati

La join richiede di sapere prima quali saranno gli oggetti da navigare (facendo attenzione a non prelevarne troppi…)

=>

il lazy loading è inefficiente in ambito sql (problema n+1)

# Domain Model

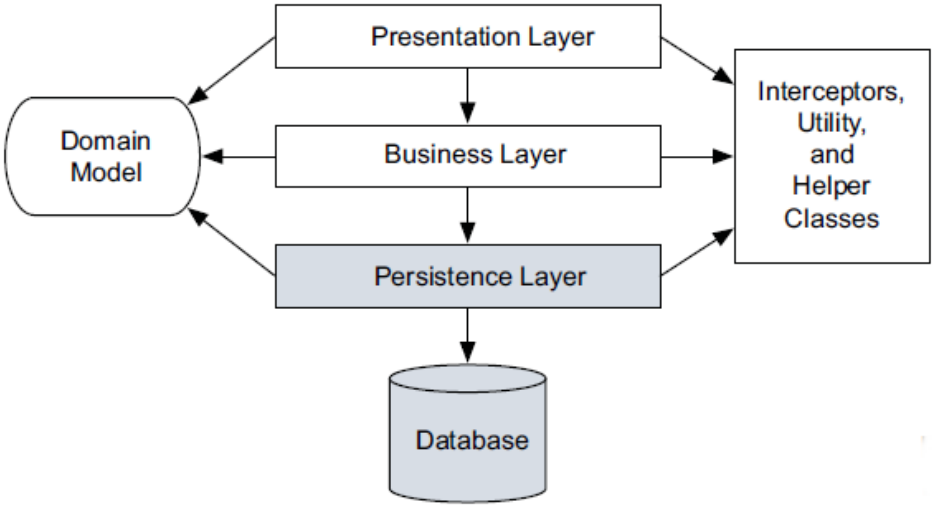
Classi da rendere persistenti

=

Domain Model

(Data Model)

## Architettura “ a strati”



# Implementazione del domain model

Hibernate favorisce l'architettura n-tier (trasparenza e automatismo)

- Le classi persistenti non devono estendere o implementare particolari classi
- Le classi persistenti possono essere usate al di fuori del contesto di persistenza

questo consente di avere codice più facile da mantenere

# Implementazione del domain model

## Il modello per rappresentare oggetti del data model è POJO

```
public class User implements Serializable {

    protected String username;

    public User() {
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public BigDecimal calcShippingCosts(Address fromLocation) {
        // Empty implementation of business method
        return null;
    }

    // ...
}
```

# Implementazione del domain model

- La classe può essere astratta
- può estendere una classe non persistente
- può implementare un'interfaccia
- non può essere interna
- non può essere final (nemmeno i metodi) (JPA)
- dev'esserci un costruttore senza argomenti (eventualmente quello di default)
- I get ed i set non sono obbligatori, Hibernate può accedere direttamente agli attributi con la reflection

# Implementazione del domain model

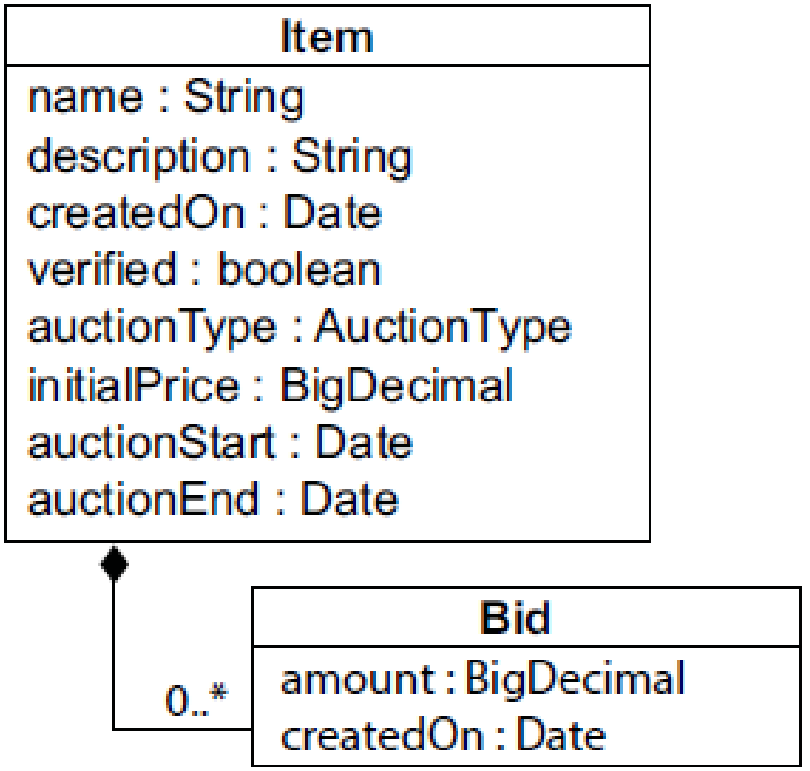
I metodi accessori possono essere utili, garantiscono l'incapsulamento:

```
public class User {

    protected String firstname;
    protected String lastname;

    public String getName() {
        return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer t = new StringTokenizer(name);
        firstname = t.nextToken();
        lastname = t.nextToken();
    }
}
```

# Implementazione del domain model
# Le associazioni

```
public class Bid {

    protected Item item;

    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }
}
```

```
public class Item {

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

    public Set<Bid> getBids() {
        return bids;
    }

    public void setBids(Set<Bid> bids) {
        this.bids = bids;
    }
}
```



# Implementazione del domain model
# Le associazioni

Associazione bidirezionale, mantenere l'integrità dei dati (un bid deve avere un item...) in Java meno semplice che in sql

```
anItem.getBids().add(aBid);
aBid.setItem(anItem);
```

```
public void addBid(Bid bid) {
    if (bid == null)                                          <-- Be defensive
        throw new NullPointerException("Can't add null Bid");
    if (bid.getItem() != null)
        throw new IllegalStateException("Bid is already assigned to an
        Item");

    getBids().add(bid);
    bid.setItem(this);
}
```

# Implementazione del domain model
# Le associazioni

Altro problema: getBids() ritorna una collection modificabile… potrei aggiungere bids che non puntano a nessun item e avere uno stato inconsistente nel db

```
public class Bid {

    protected Item item;

    public Bid(Item item) {
        this.item = item;
        item.getBids().add(this);
    }

    public Item getItem() {
        return item;
    }
}
```

Bidirectional

# I metadati del domain model
# Annotations

```java
import javax.persistence.Entity;

@Entity
public class Item {

}
```

```java
import javax.persistence.Entity;

@Entity
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Item {

}
```

# I metadati del domain model
# file esterni xml (JPA) META-INF/orm.xml

```
<entity-mappings
        version="2.1"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
            http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

    <persistence-unit-metadata>

        <xml-mapping-metadata-complete/>

        <persistence-unit-defaults>
            <delimited-identifiers/>
        </persistence-unit-defaults>

    </persistence-unit-metadata>

    <entity class="org.jpwh.model.simple.Item" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
            <basic name="name"/>
            <basic name="auctionEnd">
                <temporal>TIMESTAMP</temporal>
            </basic>
        </attributes>
    </entity>

</entity-mappings>
```

First, global metadata

Ignore all annotations and all mapping metadata in XML files.

Some default settings

Escape all SQL column, table, and other names: for example, if your SQL names are keywords (such as a "USER").

# I metadati del domain model
# file esterni xml (JPA) META-INF/orm.xml

Si possono utilizzare nomi alternativi per orm.xml, basta indicarli in persistence.xml

```
<persistence-unit name="SimpleXMLCompletePU">
    ...
    <mapping-file>simple/Mappings.xml</mapping-file>
    <mapping-file>simple/Queries.xml</mapping-file>
    ...
</persistence-unit>
```

# I metadati del domain model
# file esterni xml (JPA)  META-INF/orm.xml

Per l'ovverride delle annotations, non marcare il descrittore "complete" e indicare classe e property da sovrascrivere

```
<entity class="org.jpwh.model.simple.Item">
    <attributes>
        <basic name="name">
            <column name="ITEM_NAME"/>
        </basic>
    </attributes>
</entity>
```

Override SQL column name

# I metadati del domain model
# file esterni xml (Hibernate) *.hbm.xml


```
<?xml version="1.0"?>
<hibernate-mapping
        xmlns="http://www.hibernate.org/xsd/orm/hbm"
        package="org.jpwh.model.simple"
        default-access="field">

    <class name="Item">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <property name="auctionEnd" type="timestamp"/>

    </class>

    <query name="findItemsHibernate">select i from Item i</query>

    <database-object>
        <create>create index ITEM_NAME_IDX on ITEM(NAME)</create>
        <drop>drop index if exists
     ITEM_NAME_IDX</drop>
    </database-object>

</hibernate-mapping>
```