OBJECT™
TECHNOLOGIES

Hibernate Mappings

What Will Be Covered

OBJECT

- Advantages
- Inheritance mappings and it's types
  - Mapped superclass
  - Single Table
  - Joined Table
  - Table per class
- Association mapping
  - Example One-To-One
  - Example One-To-Many and Many-To-One
  - Example Many-To-Many
- Component mapping

Advantages

OBJECT

- Hibernate provides lots of mapping features that allow you to map complex domain and table models.
- Hibernate mappings are one of the key features of hibernate . They establish the relationship between two database tables as attributes in the model that allows to easily navigate the associations in the model and criteria queries.
- This mapping can be established either unidirectional or bidirectional i.e you can either model them as an attribute on only one of the associated entities or on both. It will not impact your database mapping tables, but it defines in which direction you can use the relationship in your model and criteria queries.
- These features may avoid writing complex joins and will allow to fetch master with all children or child with it's master information.

Inheritance mappings and it's types

OBJECT

- Although relational database systems don't provide support for inheritance, Hibernate provides several strategies to leverage this object-oriented trait onto domain model entities:
- **MappedSuperclass**
- Inheritance is implemented in the domain model only without reflecting it in the database schema.
- **Single table**
- The domain model class hierarchy is materialized into a single table which contains entities belonging to different class types.
- **Joined table**
- The base class and all the subclasses have their own database tables and fetching a subclass entity requires a join with the parent table as well.
- **Table per class**
- Each subclass has its own table containing both the subclass and the base class properties.

Mapped superclass

OBJECT

- When using MappedSuperclass, the inheritance is visible in the domain model only, and each database table contains both the base class and the subclass properties.
- e.g DebitAccount and a CreditAccount share the same Account base class.
- The **MappedSuperclass** inheritance model is not mirrored at the database level.

```
@MappedSuperclass
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;
```

```
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;
```

```
@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;
```

Mapped superclass

OBJECT

- Only 2 tables will be created in the database for both the subclasses which will have columns related to super class attributes

```
CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

```
CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Single Table

OBJECT

- The single table inheritance strategy maps all subclasses to only one database table. Each subclass declares its own persistent properties. Version and id properties are assumed to be inherited from the root class.
- Each subclass in a hierarchy must define a unique discriminator value, which is used to differentiate between rows belonging to separate subclass types. If this is not specified, the DTYPE column is used as a discriminator, storing the associated subclass name.
- The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the following restricted set of types as discriminator column: String, char, int, byte, short, boolean
- **@DiscriminatorColumn** is used to define the discriminator column as well as the discriminator type.

Single Table

OBJECT

```
@Entity
@Table(name = "employee101")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="employee")
public class Employee {

}
```

```
@Entity
@DiscriminatorValue("regularemployee")
//only mention additional columns
public class Regular_Employee extends Employee{  }

@Entity
@DiscriminatorValue("contractemployee")
 //only mention additional columns
public class Contract_Employee extends Employee{  }
```

Joined Table

OBJECT

- Each subclass can also be mapped to its own table. This is also called table-per-subclass mapping strategy. An inherited state is retrieved by joining with the table of the superclass.
- A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier.
- The primary keys of the tables for subclasses are also foreign keys to the superclass table primary key and described by the @PrimaryKeyJoinColumns.

Joined Table


```java
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Account {

        @Id
        private Long id;

@Entity(name = "DebitAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class DebitAccount extends Account {

        private BigDecimal overdraftFee;

@Entity(name = "CreditAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class CreditAccount extends Account {

        private BigDecimal creditLimit;
```

```sql
CREATE TABLE DebitAccount (
    overdraftFee NUMERIC(19, 2) ,
    account_id BIGINT NOT NULL ,
    PRIMARY KEY ( account_id )
)

CREATE TABLE CreditAccount (
    creditLimit NUMERIC(19, 2) ,
    account_id BIGINT NOT NULL ,
    PRIMARY KEY ( account_id )
)
```

Table per class

OBJECT

- A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state.
- In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish to use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public static class Account {

        @Id
        private Long id;
```

```
CREATE TABLE Account (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    PRIMARY KEY ( id )
)
```

Table per class

OBJECT

```java
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

        private BigDecimal overdraftFee;
```

```
CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

```java
@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

        private BigDecimal creditLimit;
```

```
CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Association Mapping

OBJECT

- Association mappings are one of the key features of JPA and Hibernate. They model the relationship between two database tables as attributes in your domain model. That allows you to easily navigate the associations in your domain model and JPQL or Criteria queries.
- JPA and Hibernate support the same associations as you know from your relational database model. You can use:
  - one-to-one associations,
  - many-to-one associations and
  - many-to-many associations.
- You can map each of them as a uni- or bidirectional association.

Example One-To-One

OBJECT

- One-to-one relationships are rarely used in relational table models. You, therefore, won't need this mapping too often.
- An example for a one-to-one association could be a Customer and the ShippingAddress. Each Customer has exactly one ShippingAddress and each ShippingAddress belongs to one Customer. On the database level, this mapped by a foreign key column either on the ShippingAddress or the Customer table.

```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = "fk_shippingaddress")
    private ShippingAddress shippingAddress;

    ...
}
```

```
@Entity
public class ShippingAddress{

    @OneToOne(mappedBy = "shippingAddress")
    private Customer customer;
```

Example Many-To-One and One-To-Many OBJECT

- An order consists of multiple items, but each item belongs to only one order. That is a typical example for a many-to-one association. If you want to model this in your database model, you need to store the primary key of the Order record as a foreign key in the OrderItem table.
- With JPA and Hibernate, you can model this in 3 different ways. You can either model it as a bidirectional association with an attribute on the Order and the OrderItem entity. Or you can model it as a unidirectional relationship with an attribute on the Order or the OrderItem entity.

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = "fk_order")
    private Order order;

    ...
```

Example Many-To-One and One-To-Many

OBJECT

- If @JoinColumn annotation is omitted, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute.
- The unidirectional one-to-many mapping definition is very similar to the many-to-one association. It consist of the List items attribute which stores the associated entities and a @OneToMany association.

```
@Entity
public class Order {

    @OneToMany
    @JoinColumn(name = "fk_order")
    private List<OrderItem> items = new ArrayList<OrderItem>();
```

- The bidirectional Many-to-One association mapping is the most common way

Example Many-To-One and One-To-Many OBJECT

- The mapping definition consists of 2 parts:
- the to-many side of the association which owns the relationship mapping and
- the to-one side which just references the mapping

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = "fk_order")
    private Order order;
```

```
@Entity
public class Order {

    @OneToMany(mappedBy = "order")
    private List<OrderItem> items = new ArrayList<OrderItem>();

    ...
```

Example Many-To-One and One-To-Many OBJECT

```
Order o = em.find(Order.class, 1L);

OrderItem i = new OrderItem();
i.setOrder(o);

o.getItems().add(i);

em.persist(i);
```

- Many developers prefer to implement it in a utility method which updates both entities.

```
@Entity
public class Order {
    …

    public void addItem(OrderItem item) {
        this.items.add(item);
        item.setOrder(this);
    }
    …
}
```

Example Many-To-Many

OBJECT

- Many-to-Many relationships are another often used association type. On the database level, it requires an additional association table which contains the primary key pairs of the associated entities. But as you will see, you don't need to map this table to an entity.
- A typical example for such a many-to-many association are Products and Stores. Each Store sells multiple Products and each Product gets sold in multiple Stores.
- It can even be modeled a many-to-many relationship as a uni- or bidirectional relationship between two entities.

Example Many-To-Many

OBJECT

```
@Entity
public class Store {

    @ManyToMany
    @JoinTable(name = "store_product",
            joinColumns = { @JoinColumn(name = "fk_store") },
            inverseJoinColumns = { @JoinColumn(name = "fk_product") })
    private Set<Product> products = new HashSet<Product>();
```

- For bidirectional association

```
@Entity
public class Product{

    @ManyToMany(mappedBy="products")
    private Set<Store> stores = new HashSet<Store>();
```

Example Many-To-Many

OBJECT

```
@Entity
public class Store {

    @ManyToMany
    @JoinTable(name = "store_product",
            joinColumns = { @JoinColumn(name = "fk_store") },
            inverseJoinColumns = { @JoinColumn(name = "fk_product") })
    private Set<Product> products = new HashSet<Product>();
```

- For bidirectional association

```
@Entity
public class Product{

    @ManyToMany(mappedBy="products")
    private Set<Store> stores = new HashSet<Store>();
```

Example Many-To-Many

OBJECT

- You need to update both ends of a bidirectional association when you want to add or remove an entity. Doing that in your business code is verbose and error-prone. It's, therefore, a good practice to provide helper methods which update the associated entities.

```
@Entity
public class Store {

    public void addProduct(Product p) {
        this.products.add(p);
        p.getStores().add(this);
    }

    public void removeProduct(Product p) {
        this.products.remove(p);
        p.getStores().remove(this);
    }

    ...
```

Component Mapping

OBJECT

- In component mapping, we will map the dependent object as a component. An component is an object that is stored as an value rather than entity reference. This is mainly used if the dependent object doen't have primary key. It is used in case of composition (HAS-A relation).
- Emp class has a property called address which is of type Address having properties as area, city and pincode.
- Address is a dependent object. Hibernate framework provides the facility to map the dependent object as a component.

```
@Embeddable
public class Address
{
    @Column
    String area;
    @Column
    String city;
    @Column
    String pincode;
```

```
@Entity
@Table(name="students")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    int sid;

    @Column
    String name;

    @Embedded
    Address address;
```

Collection Mapping

OBJECT

- JPA and Hibernate provide 3 main options to map a Collection. If it's a Collection of other entities, you can model it as a to-many association. This is the most common mapping. But you can also map it as an @ElementCollection or as a basic type.
- **Map a Collection as an Association**
- These are the most common Collection mappings because they are easy to implement, fit a relation table model very well, and provide great performance. To model such an association in your domain model, you only need to add an attribute of type java.util.List or java.util.Set to your entity class and annotate it with @ManyToOne or @ManyToMany.
- **Map a Collection as an @ElementCollection**
- An @ElementCollection enables you to map a Collection of values that are not an entity itself. This might seem like an easy solution for lists of basic attributes, like the phone numbers of a person. In the database, Hibernate maps the @ElementCollection to a separate table. Each value of the collection gets stored as a separate record.

Collection Mapping

OBJECT

```
@Entity
public class Author {

    @ElementCollection
    private List<String> phoneNumbers = new ArrayList<>();

    ...
```

- This often becomes a performance issue if you need to change the elements in the collection. Because they don't have their own identity, all elements of an @ElementCollection are always read, removed, and written, even if you only add, change, or remove one of them.
- **Map a Collection as a Basic Type**
- Hibernate can map a Collection as a basic type that gets mapped to 1 database column. You only rarely see this kind of mapping in a project.

Hibernate mappings

The most important advantage of any ORMapping tool is it's strength in mapping the different kinds of relationship between entities which will get reflected in database design like composition, association, inheritance etc.

Need and advantages

Hibernate provides lots of mapping features that allow you to map complex domain and table models.

Hibernate mappings are one of the key features of hibernate . They establish the relationship between two database tables as attributes in the model that allows to easily navigate the associations in the model and criteria queries.

This mapping can be established either unidirectional or bidirectional i.e you can either model them as an attribute on only one of the associated entities or on both. It will not impact your database mapping tables, but it defines in which direction you can use the relationship in your model and criteria queries.

These features may avoid writing complex joins and will allow to fetch master with all children or child with it's master information.

Inheritance mapping

There are four inheritance mapping strategies in hibernate.

MappedSuperclass

Inheritance is implemented in the domain model only without reflecting it in the database schema. In this strategy, the parent classes can't be entities. Tables get created for sub classes by merging the properties of super class and super class to be annotated with @MappedSuperclass

```
@MappedSuperclass
public static class Account {
   @Id
   private Long id;
   private String owner;
}
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {
   private BigDecimal overdraftFee;
}
@Entity(name = "CreditAccount")
```

```
@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {
   private BigDecimal creditLimit;
}
```

Single Table Per Class:

In Single table per subclass, the union of all the properties from the inheritance hierarchy is mapped to one table. As all the data goes in one table, a discriminator is used to differentiate between different type of data.

// DiscriminatorColumn - Tells about the type of data

// DiscriminatorValue - Is the data is representing User type, the // value is "USER"

Super class - User

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("USER")
@Table(name="BASIC_USER")
public class User {
   protected long id;
   protected String name;
   ...
}
```

Customer Class (Subclass of User)

//There is no id column

```
@Entity
@DiscriminatorValue("CUST")
public class Customer extends User{
   protected double creditLimit;
   ...
}
```

Employee Class (Subclass of User)

Employee Class (Subclass of User)

//There is no id column

```
@Entity
@DiscriminatorValue("EMP")
public class Employee extends User{
   protected String rank
   ...
}
```

The table structure is as follows :

Basic_User => id, name, DISCRIMINATOR, creditLimit, rank

Advantage : Performance wise better than all strategies because no joins or sub-selects need to be performed.

Disadvantage : Many column of table are nullable so the NOT NULL constraint cannot be applied and tables are not normalized.

Table Per Class

In a Table per class inheritance strategy, each concrete subclass has its own table containing both the subclass and the base class properties. It is better to declare super class as abstract otherwise separate table gets created for super class with it's own attributes and super class attributes are repeated in subclass mapped tables

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Account {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String owner;
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
   private double creditLimit;
}
```

Table Per Class

In a Table per class inheritance strategy, each concrete subclass has its own table containing both the subclass and the base class properties. It is better to declare super class as abstract otherwise separate table gets created for super class with it's own attributes and super class attributes are repeated in subclass mapped tables

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Account {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String owner;
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
   private double creditLimit;
}
@Entity(name = "DebitAccount")
public class DebitAccount extends Account {
   private double overdraftFee;
}
```

The table structure is as follows :

Account => id, owner (gets created only if Account is not abstract)

CreditAccount => id, owner, creditLimit

DebitAccount => id, owner, overdraftFee

Advantage : Possible to define NOT NULL constraint on the table

Disadvantage : Tables are not normalized

Table Per Sub Class(JOINED strategy)

In this case all the classes are mapped to its own table. It's highly normalized but performance is not good. Here, foreign key is maintained between the tables.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Account {
```

Table Per Sub Class(JOINED strategy)

In this case all the classes are mapped to its own table. It's highly normalized but performance is not good. Here, foreign key is maintained between the tables.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Account {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String owner;
}
```

The primary key of this table is also a foreign key to the superclass table and is described by the @PrimaryKeyJoinColumns. The table name still defaults to the non-qualified class name. Also, if @PrimaryKeyJoinColumn is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

```
@Entity(name = "CreditAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public class CreditAccount extends Account {
   private double creditLimit;
}
@Entity(name = "DebitAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public class DebitAccount extends Account {
   private double overdraftFee;
}
```

The table structure is as follows :

Account => id, owner

CreditAccount => account_id(primary key as well as foreign key), creditLimit

DebitAccount => account_id(primary key as well as foreign key), overdraftFee

Association mapping

[illegible] entities, either directly as an embedded property or field or indirectly via a collection

Association mapping

Entities can contain references to other entities, either directly as an embedded property or field or indirectly via a collection of some sort (arrays, sets, lists, etc.). These associations are represented using foreign key relationships in the underlying tables. These foreign keys will rely on the primary ids used by participating tables.

Types of association mappings

Hibernate supports 4 types of associations as follows

One to One

You can use a One To One association mapping to represent a one-to-one relationship between database tables. A one-to-one relationship occurs between tables when one record from the main table corresponds to only one record in the second table. For example, a Person can have only one address. So one record in a Person table will correspond to only one record in the address table.

One To Many

You can use a One To Many association mapping to represent a one to many relationships between tables. A one-to-many relationship occurs between tables when one record from a table can correspond to many records from another table. So for example, a Person can have many email ids. So one record in the Person table will correspond to many records in the Email table.

Many To One

You can use a Many To One association mapping to represent a many to one relationship between tables. A many-to-one relationship occurs when many records from the main table correspond to only one record from the second table. So for example, many cities belong to the same state, so many records from the city table will correspond to the same record from the state table.

Many To Many

You can use a Many To Many association mapping to represent a many-to-many relationship between tables. A many-to-many relationship occurs when many records from one table correspond to many records from another table. So for example, many students can belong to a class and a class can have many students. So many records from the student table can correspond to many records from the class table.

Direction of Associations

An association can be unidirectional or bidirectional. This specifies the direction in which navigation can occur in the classes corresponding to the tables.

Direction of Associations

An association can be unidirectional or bidirectional. This specifies the direction in which navigation can occur in the classes corresponding to the tables.

Unidirectional Association

In a Unidirectional association, navigation access is only in one direction. So for example, if there is a Person class and an Address class, a person class will have an Address object

Bidirectional Association

In a Bidirectional association, navigation access is in both directions. So the Person class will have an address object and the address class will have a person object.

Example

An order consists of multiple items, but each item belongs to only one order. That is a typical example for a many-to-one association. You can either model it as a bidirectional association with an attribute on the Order and the OrderItem entity. Or you can model it as a unidirectional relationship with an attribute on
the Order or the OrderItem entity.

Unidirectional Many-to-One Association

```
@Entity
public class OrderItem {
   @ManyToOne
   @JoinColumn(name = "fk_order")
   private Order order;
   ...
}
```

Unidirectional One-to-Many Association

```
@Entity
public class Order {
   @OneToMany
   private List<OrderItem> items = new ArrayList<OrderItem>();
   ...
}
```

Bidirectional Many-to-One Associations

Bidirectional Many-to-One Associations

The bidirectional Many-to-One association mapping is the most common way to model this relationship with JPA and Hibernate. It uses an attribute on the Order and the OrderItem entity. This allows you to navigate the association in both directions

Cascade operations

Entity relationships often depend on the existence of another entity. Cascade is the feature provided by hibernate to automatically manage the state of mapped entity whenever the state of its relationship owner entity is affected.

```
@Entity
public class Person {
   @Id
   private Long id;
   private String name;
   @OneToMany(mappedBy = "owner", cascade = CascadeType.ALL)
   private List <Phone> phones = new ArrayList < > ();
}
@Entity
public class Phone {
   @Id
   private Long id;
   @Column(name = "number")
   private String number;
   @ManyToOne(fetch = FetchType.LAZY)
   private Person owner;
}
```

Component mapping

In general, a student/employee can have an address. For these kind of requirements, we can follow Component mapping. It is nothing but a class having a reference to another class as a member variable. i.e. inside the 'student' class, we can have the 'address' class as a member variable. In MySQL, only a single table is enough which holds the information of all the attributes of both classes. Here in our example, we can see 'employee' as primary class and it has 'address' as member variable. Component Mapping represents the has-a relationship, the composition is stronger association where the contained object has no existence of its own. Contained object's class is annotated with @Embeddable and the property in the container class is annotaed with @Embedded

Component mapping

In general, a student/employee can have an address. For these kind of requirements, we can follow Component mapping. It is nothing but a class having a reference to another class as a member variable. i.e. inside the 'student' class, we can have the 'address' class as a member variable. In MySQL, only a single table is enough which holds the information of all the attributes of both classes. Here in our example, we can see 'employee' as primary class and it has 'address' as member variable. Component Mapping represents the has-a relationship, the composition is stronger association where the contained object has no existence of its own. Contained object's class is annotated with @Embeddable and the property in the container class is annotaed with @Embedded

```
@Entity
@Table(name="EMPLOYEE")
public class Employee
{
   @Id
   @GeneratedValue
   @Column(name="EMP_ID")
   private int id;
   @Column(name="EMP_NAME")
   private String name;
   @Embedded
   private EmployeeAddress address;
}
@Embeddable
public class EmployeeAddress
{
   @Column(name="STREET")
   private String street;
   @Column(name="CITY")
   private String city;
   @Column(name="STATE")
   private String state;
}
```

The table structure is as follows :

EMPLOYEE => EMP_ID, EMP_NAME, STREET, CITY, STATE

Assignments

Assignments

1. Create Student class having information as id(int), name(String and address(Address). Use Address as embeddable entity.
2. Create Emp class having information as id(int), name(String), email(String), contactno(string) and basic(float) . Inherit this class to create PermanentEmp having information as , allowances(float), bonus(float) and deductions(float) and TemporaryEmp having information as extrapay(float) and taxes(float) .Use hibernate DDL auto functionality to create required tables. Use JOINED strategy for inheritance.
3. Create Product class with information pid(pk), name, price and catid(fk) with Category having information as cid(pk), cname, cdesc. Save and retrieve information about categories and products using association mapping.
   a. Save Category instance
   b. Use this category for creating product and then save it
   c. Create new category with all the products and save it.
   d. Create new product with new category and save it.
4. Create Subject class with properties subjectid, name and desc. Create Question class with properties qid, text, opt1, opt2, opt3, opt4, answer and information about subjectid. Use above classes and maintain one to many and many to one association mapping as required.
   1. Save Subject instance
   2. Use this Subject for creating Question and then save it.
   3. Create new subject with 2 questions as child entiies and save it together.
   4. Fetch information about all Questions and display subject name with all the questions.
5. Develop the web application for implementing login feature.