Java Persistence Architecture (JPA)

Session-1

What is Persistence

- **Persistence** is the ability of an object to survive beyond the lifetime of the program that creates it.

- Persistence is relevant for objects with an internal state. The state needs to be retained between object deactivation and object activation.
  - Storing object state on persistent storage before de-activation
  - Upon activation, load object state from persistent storage

- For an object-oriented language like Java, persistence ensures that the state of an object is accessible even after the application that created it has stopped executing.

- Persistent storage can be
  a. File system b. Relational Database c. Object-Database d. Flash-RAM

JDBC & Persistence

Java applications traditionally used **JDBC (Java Database Connectivity) API** to persist data into **relational databases**.

The JDBC API uses SQL statements to perform create, read, update, and delete (CRUD) operations. JDBC code is embedded in Java classes -- in other words, it's tightly coupled to the business logic.

This code also relies heavily on SQL, which is not standardized across databases; that makes migrating from one database to another difficult.

Relational database technology emphasizes data and its relationships, whereas the object-oriented paradigm used in Java concentrates not only on the data but also on the operations performed on that data.

Object-Relational Mapping (ORM)

***Object-relational mapping* (ORM) is the solution.**

ORM is a technique that transparently persists *application objects to the tables in a relational database* where in *entities/classes are mapped to tables*, *instances are mapped to rows* and *attributes of instances are mapped to columns of table*.

- ORM behaves like a virtual database, hiding the underlying database architecture from the user.
- It provides functionality to perform complete CRUD operations and encourages object-oriented querying.
- Also supports metadata mapping and helps in the transaction management of the application.

Java Persistence Architecture(JPA)

The **Java Persistence Architecture API (JPA)** is a Java specification for accessing, persisting, and managing data between Java objects and relational database.

JPA was defined as part of the **EJB 3.0** specification and is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

JPA itself is just a specification with some set of interfaces, and requires an implementation.

There are open-source and commercial JPA implementations such as :

Ex. Hibernate
EclipseLink
iBATIS SQL Maps
Java Ultra-Lite Persistence

•A persistence framework maps the objects in the application domain to data in a database using either XML files or metadata annotations.

Hibernate Framework

- Hibernate is an open-source ORM solution for Java applications.

- **Hibernate** is an object-relational mapping (ORM) library for the Java language, providing a **framework** for mapping an object-oriented domain model to a traditional relational database.

- Hibernate was developed by a team headed by **Gavin King**. The development of Hibernate began in 2001 and the team was later acquired by JBoss, which now manages it.

- Hibernate was developed initially for Java; in 2005 a .Net version named **NHibernate** was introduced.

- The current version of Hibernate is 4.x, and it supports Java annotations.

Interacting with Databases through the Java Persistence API

- Earlier versions of J2EE used Entity Beans as the standard approach for object relational mapping.
- Entity Beans attempted to keep the data in memory always synchronized with database data, a good idea in theory, however, in practice this feature resulted in poorly performing applications.
- Several object relational mapping APIs were developed to overcome the limitations of Entity Beans, such as Hibernate, iBatis, Cayenne, and Toplink among others.
- **With Java EE 5, Entity Beans were deprecated in favour of JPA**. JPA took ideas from several object relational mapping tools and incorporated them into the standard.

JPA

Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes

An entity class must follow these requirements.

- The class must be annotated with the **javax.persistence.Entity** annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Interacting with Databases through the Java Persistence API

- JPA entities are Java classes whose fields are persisted to a database by the JPA API.
- JPA entities are **Plain Old Java Objects (POJOs)**, as such, they don't need to extend any specific parent class or implement any specific interface.
- A Java class is designated as a JPA entity by decorating it with the @Entity annotation.

What differentiates a JPA entity from other Java objects are a few JPA-specific annotations.

- The @Entity annotation is used to indicate that our class is a JPA entity. Any object we want to persist to a database via JPA must be annotated with this annotation.
- The @Id annotation is used to indicate what field in our JPA entity is its primary key. The primary key is a unique identifier for our entity. No two entity instances may have the same value for their primary key field.
- This annotation can be placed just above the getter method for the primary key class.

The @Entity and the @Id annotations are the bare minimum two annotations that a class needs in order to be considered a JPA entity. JPA allows primary keys to be automatically generated. In order to take advantage of this functionality, the **@GeneratedValue** annotation can be used.

JPA Annotations Vs Hibernate Annotations

Hibernate is an implementation of the JPA specification.

JPA is a standards specification, and a set of annotations and interfaces.

You need an implementation of JPA to use it, and Hibernate is one of them. Just like to use JDBC, you need a database driver.

The package of the annotation is ***javax.persistence***, so they're JPA annotations. Hibernate annotations are in the package ***org.hibernate.xxx.***

The javax.persistence annotations are a standards specification. The hibernate annotations represent Hibernate's specific implementation.

They mostly overlap. Generally, use the javax.persistence annotations whenever possible.

Where to place hibernate annotations?

- We can place them either on the *field* or on the *getter method*.
- The EJB3 spec requires that we declare annotations on the element type that will be accessed, i.e. the getter method .
- ***Mixing annotations in both fields and methods should be avoided.***
- Hibernate will guess the access type from the position of @Id or @EmbeddedId.

Customer.java : Annotations before properties

```
package org.asr.hibernate;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name= "customer")
public class Customer {
@Id
@GeneratedValue
@Column(name= "id")
private int id;
@Column(name="firstname")
private String firstName;
@Column(name="lastname")
private String lastName;
@Column(name="balance")
private double balance;

public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}
public String getFirstName() {
return firstName;
}
public void setFirstName(String firstName) {
this.firstName = firstName;
}
public String getLastName() {
return lastName;
}
public void setLastName(String lastName) {
this.lastName = lastName;
}
public double getBalance() {
return balance;
}
public void setBalance(double balance) {
this.balance = balance;
}
@Override
public String toString() {
return "Customer [id=" + id + ", firstname=" + firstname + ",
lastname="
+ lastname + ", balance=" + balance + "]";
}
}
```

Customer.java : Annotations before getter methods

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

private int id;
private String firstname;
private String lastname;
private double balance;

@Id
@GeneratedValue
@Column(name= "id")
public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}

@Column(name="firstname")
public String getFirstname() {
return firstname;
}
public void setFirstname(String firstname) {
this.firstname = firstname;
}

@Column(name="lastname")
public String getLastname() {
return lastname;
}
public void setLastname(String lastname) {
this.lastname = lastname;
}

@Column(name="balance")
public double getBalance() {
return balance;
}
public void setBalance(double balance) {
this.balance = balance;
}

@Override
public String toString() {
return "Customer [id=" + id + ", firstname=" + firstname + ", lastname="
+ lastname + ", balance=" + balance + "]";
}

}
```

It is sufficient annotating getter method of primary key property, not necessary to annotate other getter methods

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="OrderProcessing">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>com.cgs.jpa.businesstier.User</class>
<properties>
<property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.connection.driver_class" value="oracle.jdbc.OracleDriver"/>
<property name="hibernate.connection.username" value="scott"/>
<property name="hibernate.connection.password" value="tiger"/>
<property name="hibernate.connection.url" value="jdbc:oracle:thin:@//localhost:1521/orcl"/>
</properties>
</persistence-unit>
</persistence>
```

Maven Dependencies

```
<!-- http://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.0.Final</version>
</dependency>
<!-- http://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.1.0.Final</version>
</dependency>
```

```
<!-- http://mvnrepository.com/artifact/javax.transaction/jta -->
<dependency>
    <groupId>javax.transaction</groupId>
    <artifactId>jta</artifactId>
    <version>1.1</version>
</dependency>
```

Service Class

```
public class UserService implements IUser{

private static EntityManagerFactory emf;

private static EntityManager entityManager;
static{
emf= Persistence.createEntityManagerFactory("Bigparty_V2");
entityManager = emf.createEntityManager();
}
@Override
public Boolean isValidUser(Long id) {
// TODO Auto-generated method stub
User user=entityManager.find(User.class, id);
if(user == null){
return false;
}
return true;

}
```

Service Class

Retrieving the data

```
@Override
public User getUserById(Long id) {
try{
System.out.println("In UserService");
//String sql="SELECT u FROM User u where u.id=:uid";
String sql="FROM User where id=:uid";
entityManager.getTransaction().begin();
Query query=entityManager.createQuery(sql);
query.setParameter("uid", id);
User user=(User)query.getSingleResult();
return user;
}
catch(PersistenceException e){
e.printStackTrace();
}
return null;
}
```

```
@Override
public List<User> getAllUsers() {
String sql="From User";
try{
entityManager.getTransaction().begin();
Query query=entityManager.createQuery(sql);
    List<User> userList=query.getResultList();
    entityManager.getTransaction().commit();
    return userList;
}
catch(PersistenceException e){
e.printStackTrace();
}
return null;
}
```

Service Class

Using sequence in Insert operation for Oracle database

1. ***Create a sequence in you Oracle account.***

   Ex. Create sequence customer_new_seq start with 101 increment by 1;

```
@Id
@SequenceGenerator(name="SEQ_GEN", sequenceName="CUSTOMER_NEW_SEQ", allocationSize=1)
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_GEN")
private Long customerId;
```

Note: **allocationSize** should be same as the value specified in **increment by** clause of the sequence

```
try{
entityManager.getTransaction().begin();
entityManager.persist(customer);
entityManager.getTransaction().commit();
return "SUCCESS: Customer record added to database";

} catch(PersistenceException e){
Logger logger=Logger.getLogger(this.getClass());
logger.error(e.getMessage(),e);
e.printStackTrace();
}
```

Service Class

Insert operation : Using Auto Increment in MySQL

```
.......
try{
entityManager.getTransaction().begin();

/* Do not set an ID before you save or persist it. Hibernate will look at the Entity
 * you've passed and it assumes that because it has its Primary Key populated that it
 * is already in the database. So set Id to null. Our table ( MySQL)  Id is auto increment, hence
 * takes care of automatically generating primary key.
 */
user.setId(null);
entityManager.persist(user);
entityManager.getTransaction().commit();
return true;
}
catch(PersistenceException e){
e.printStackTrace();
}
.....
```

How to create and handle composite primary key in JPA

Method I

You can make an Embedded class, which contains your two keys, and then have a reference to that class as EmbeddedId in your Entity.
You would need the @EmbeddedId and @Embeddable annotations.

```
    @Entity
    public class YourEntity {
        @EmbeddedId
        private MyKey myKey;
        @Column(name = "ColumnA")
        private String columnA;
        /** Your getters and setters **/
}
```

```
@Embeddable
public class MyKey implements Serializable {
    @Column(name = "Id", nullable = false)
    private int id;
    @Column(name = "Version", nullable = false)
    private int version;
    /** getters and setters **/
}
```

How to create and handle composite primary key in JPA

Method II

Another way to achieve this task is to use `@IdClass` annotation, and place both your `id` in that `IdClass`. Now you can use normal `@Id` annotation on both the attributes

```
public class MyKey implements_Serializable {
@Column(name="line_number",nullable=false)
private Integer lineNumber;
@Column(name="order_id",nullable=false)
private Integer orderId;

//getter and setter methods
//override equals() and hashCode() methods
```

```
@Entity
@IdClass(MyKey.class)
@Table(name="order_detail_jpa")
public class OrderDetail {
@Id
@Column(name="line_number")
private Integer lineNumber;
@Id
@Column(name="order_id")
private Integer orderId;
……
```

Thank You!