**JPA Tutorial**

JPA tutorial မွာ Java Persistence API ရဲ႕ basic နဲ႔ advance concept မ်ားကို သိရမွာ ၿဖစ္ပါတယ္။ JPA က အဓိက ဘာကိုလုပ္ေဆာင္ေပးသလဲဆိုရင္ Java applications မ်ားထဲမွာ ရွိတဲ့ relational data မ်ားကို manage လုပ္ရန္ object-relational mapping ကို လုပ္ေဆာင္ေပးၿခင္း ၿဖစ္ပါတယ္။ SQL statements မ်ားကို အသံုးမွဳမ်ားရဲ႕ objects instead မ်ားကို တိုက္ရိုက္ လုပ္ေဆာင္ႏိုင္ရန္ platform တစ္ခု ေပးစြမ္းပါတယ္။

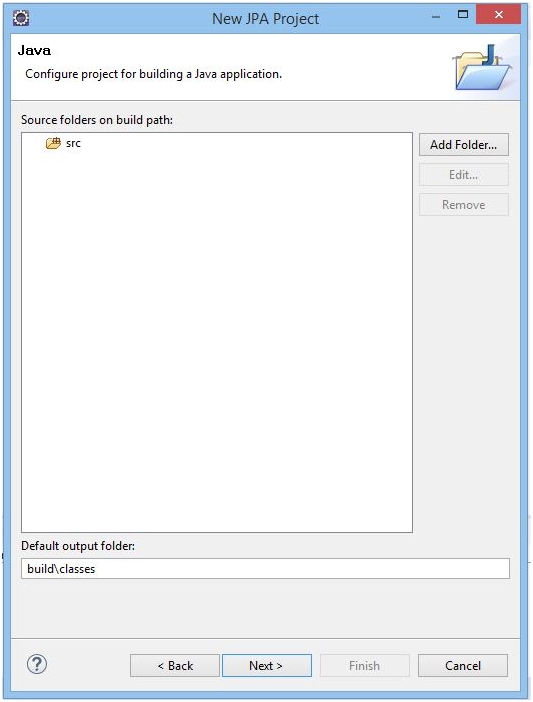
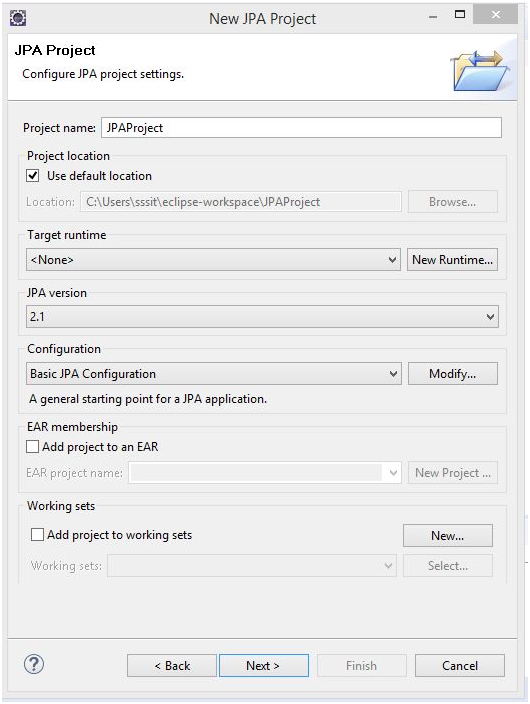
1. JPA Introduction

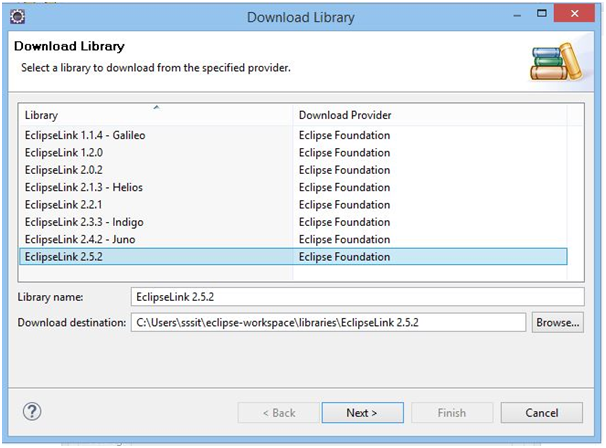
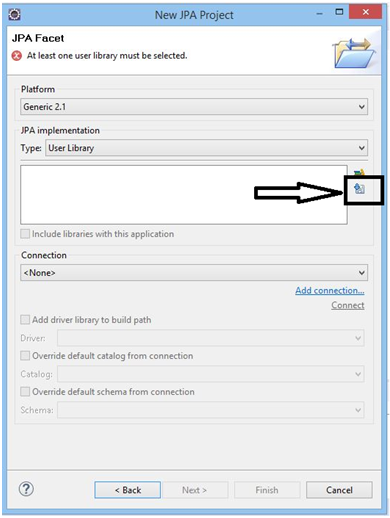
JPA ကို Java object နဲ႔ relational database ၾကားထဲမွာ data မ်ား persist လုပ္ေဆာင္ႏိုင္ရန္ အတြက္အသံုးၿပဳပါတယ္။ ဆိုလိုတာက JPA ဆိုတာ object-oriented domain models မ်ားနဲ႔ relational database systems မ်ားကို ဆက္သြယ္ေပးထားတဲ့ တံတားတစ္ခု ၿဖစ္ပါတယ္။

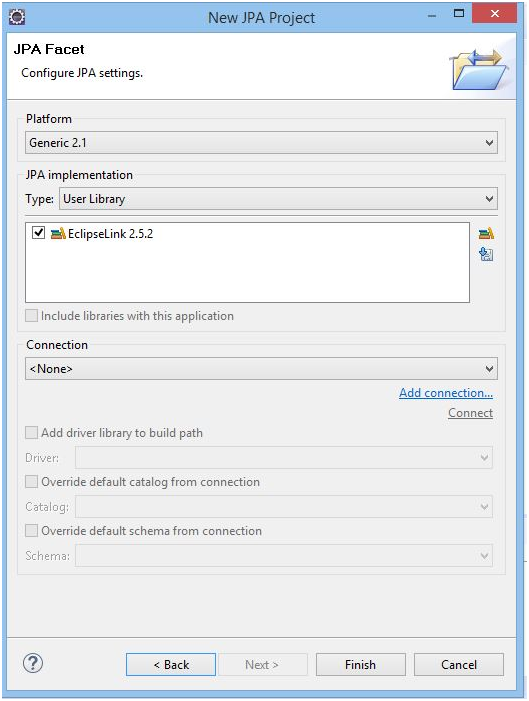
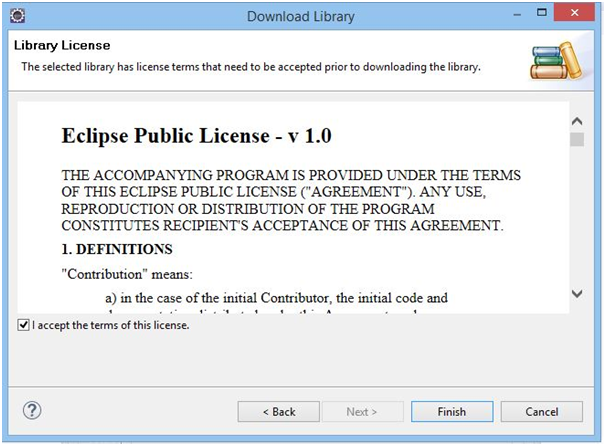
JPA ဆိုတာက specification တစ္ခုပဲ ၿဖစ္တယ္။ သူ႕ဟာနဲ႔သူ operation မလုပ္ေဆာင္ႏိုင္ဘူး။ ဆိုလိုတာက သူအေနနဲ႔ implementation တစ္ခု လိုအပ္တယ္။ ဒါေၾကာင့္ ORM tools မ်ားၿဖစ္ၾကတဲ့ Hibernate, TopLink နဲ႔ iBatis အစရွိသည္တို႔က data persistence အတြက္ JPA specifications မ်ား implements လုပ္ၾကၿခင္း ၿဖစ္ပါတယ္။

2. JPA Installation

File>New>JPA Project







3. JPA ORM

JPA Object Relational Mapping

Object Relational Mapping (ORM) ဆိုတာ functionality တစ္ခု ၿဖစ္ပါတယ္။ ORM ကို object တစ္ခု နဲ႔ relational database ၾကားထဲမွာ object state နဲ႔ database column လုပ္ၿခင္းအားၿဖင့္ အသံုးၿပဳလို႔ရပါတယ္။ အမ်ိဳးမ်ိဳးေသာ database ေတြနဲ႔လည္း အဆင္ေၿပေၿပ

ခ်ိတ္ဆက္ႏိုင္ပါတယ္။



ORM Frameworks

ေအာက္မွာၿပထားတဲ့ အမ်ိဳးမ်ိဳးေသာ frameworks မ်ားက ORM mechanism ေပၚမွာ လုပ္ေဆာင္ၾကပါတယ္။

1. Hibernate
2. TopLink
3. ORMLite
4. iBATIS
5. JPOX

Mapping Directions

Mapping Directions ကို အပိုင္း ၂ ပိုင္းခြဲၿခားထားပါတယ္။

1. **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
2. **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

1. **One-to-one -** @OneToOne annotation အေနနဲ႔ ကိုယ္စားၿပဳပါတယ္။ entity တစ္ခုခ်င္းစီတုိင္းရဲ႕ instance ဟာ အၿခား entity တစ္ခုရဲ႕ single instance နဲ႔ ဆက္စပ္ေနမွာ ၿဖစ္ပါတယ္။
2. **One-to-many -** @OneToMany annotation အေနနဲ႔ ကိုယ္စားၿပဳပါတယ္။ entity တစ္ခုရဲ႕ instance တစ္ခုဟာ အၿခား entity တစ္ခုရဲ႕ တစ္ခုထက္ပိုေသာ instance မ်ားနဲ႔ relate ၿဖစ္ခဲ့မယ္ဆိုရင္ ဒီ relationship ကို အသံုးၿပဳမွာ ၿဖစ္ပါတယ္။
3. **Many-to-one -** @ManyToOne annotation အေနနဲ႔ ကိုယ္စားၿပဳပါတယ္။ entity တစ္ခုရဲ႕ multiple instances မ်ားက အၿခား entity တစ္ခုရဲ႕ single instance နဲ႔ relate ၿဖစ္ေနမယ္ဆိုရင္ ဒီ relationship ကို အသံုးၿပဳပါတယ္။
4. **Many-to-many -** @ManyToMany annotation အေနနဲ႔ ကိုယ္စားၿပဳပါတယ္။ entity တစ္ခုရဲ႕ multiple instances မ်ားက အၿခား entity တစ္ခုရဲ႕ multiple instances မ်ားနဲ႔ ဆက္စပ္ေနမယ္ဆိုရင္ ဒီ relationship ကို အသံုးၿပဳႏိုင္ပါတယ္။ ဒီ mapping မွာဆိုရင္ မည္သည့္ side မဆို owing side ၿဖစ္ႏိုင္ပါတယ္။

**JPA Entity**

1. JPA Entity Introduction

ေယဘုယေၿပာရမယ္ဆိုရင္ entity ဆိုတာက single unit တစ္ခုထဲမွာ states ေတြကို အတူတကြစုေပါင္းထားတဲ့ group တစ္ခုၿဖစ္ပါတယ္။ entity တစ္ခုဆိုတာက object တစ္ခုအေနနဲ႔ ၿပဳမူၿပီးေတာ့ object-oriented paradigm ရဲ႕ အဓိက အစိတ္အပိုင္းအေနနဲ႔ ၿဖစ္လာတယ္။ အဲ့ေတာ့ entity တစ္ခုဆိုတာက Java Persistence Library ထဲမွာ application-defined object တစ္ခုၿဖစ္ပါတယ္။

Entity Properties

**Persistability** – object တစ္ခုၿဖစ္ၿပီးေတာ့ သူ႕ကို database ထဲမွာ သိမ္းခဲ့မယ္ဆိုရင္ေတာ့ persistent လို႔ေခၚပါတယ္။ ၿပီးေတာ့ သူ႕ကို အခ်ိန္မေရြး access လုပ္ႏိုင္ပါတယ္။

**Persistent Identity** – java မွာ entity တိုင္းမွာ unique ၿဖစ္ပါတယ္။ object identity တစ္ခုအၿဖစ္လည္း ကိုယ္စားၿပဳပါတယ္။ အလားတူ object identity ကို db တစ္ခုတည္းမွာ သိမ္းဆည္းလိုက္ခ်ိန္မွာ object identity အၿဖစ္ ကိုယ္စားၿပဳပါတယ္။ အဲ့ object identity ဟာ db ထဲက primary key နဲ႔ equivalent ၿဖစ္ပါတယ္။

**Transactionality** – entity တစ္ခုဟာ operations မ်ားစြာကို လုပ္ေဆာင္ႏိုင္တယ္ ဥပမာ create, delete, update အစရွိသၿဖင့္ေပါ့။ အဲ့ operation တစ္ခုဟာ db ထဲမွာ ေၿပာင္းလဲမွဳ ၿဖစ္ေစပါတယ္။ db ထဲမွာ change လုပ္တဲ့ လုပ္ေဆာင္ခ်က္ဟာ success or fail ၿဖစ္တယ္ဆိုတာကို ၾကည့္ၿခင္းအားၿဖင့္ သိႏိုင္တယ္။

**Granuality** – Entities မ်ားဟာ primitives, primitive wrappers သို႔မဟုတ္ single dimensional state နဲ႔ built-in objects မ်ား မၿဖစ္သင့္ပါ။

Entity Metadata

entity တစ္ခုစီတိုင္းဟာ အခ်ိဳ႕ေသာ metadata မ်ားနဲ႔ ဆက္စပ္ေနပါတယ္။ metadata ဆိုတာကလည္း entity ရဲ႕ information မ်ားကို represent လုပ္ထားတယ္။ database အစား အဲ့ metadata က class ရဲ႕ အၿပင္ဘက္မွာေရာ အတြင္းထဲမွာေရာ တည္ရွိႏိုင္တယ္။ metadata က ေအာက္ေဖာ္ၿပပါ form အတိုင္း follow လုပ္ပါတယ္။

**Annotation** – Java မွာ annotations မ်ားဟာ tags ေတြရဲ႕ form ၿဖစ္ပါတယ္။ သူက metadata ကို ကိုယ္စားၿပဳပါတယ္။ class အတြင္းထဲမွာ အဲ့ metadata က persist လုပ္ပါတယ္။

**XML** – In this form, XML file ထဲမွာ class ရဲ႕ အၿပင္ဘက္ metadata က persist ၿဖစ္ပါတယ္။

2. JPA Creating an Entity

java class တစ္ခုက entity တစ္ခုအၿဖစ္ အလြယ္တစ္ကူ transformed ၿဖစ္ႏိုင္ပါတယ္။ အေၿခခံလိုအပ္ခ်က္မ်ားကေတာ့

1. No-argument Constructor
2. Annotation

Simple Student class

**public** **class** Student {

**private** **int** id;

**private** String name;

**private** **long** fees;

**public** Student() {}

**public** Student(**int** id)

      {

**this**.id = id;

         }

**public** **int** getId()

     {

**return** id;

         }

**public** **void** setId(**int** id)

     {

**this**.id = id;

         }

**public** String getName()

     {

**return** name;

         }

**public** **void** setName(String name)

     {

**this**.name = name;

         }

**public** **long** getFees()

     {

**return** fees;

         }

**public** **void** setFees (**long** fees)

     {

**this**.fees = fees;

     }   }

@Entity - marker annotation ၿဖစ္တယ္။ ဒီ class က entity တစ္ခုၿဖစ္တယ္ဆိုတာ ညႊန္းဆိုတယ္။ အဲ့ annotation က class name ရဲ႕အေပၚမွာ ရွိရမွာ ၿဖစ္ပါတယ္။

@Id - ဒီ annotation က specific field တစ္ခုရဲ႕ အေပၚမွာ ရွိရမွာ ၿဖစ္တယ္။ သူက persistent identifying properties မ်ားကို hold လုပ္ေပးထားပါတယ္။ အဲ့ field က db ထဲက primary key အေနနဲ႔ သတ္မွတ္ၿခင္း ခံရပါတယ္။

Simple Entity Class

**import** javax.persistence.\*;

@Entity

**public** **class** Student {

    @Id

**private** **int** id;

**private** String name;

**private** **long** fees;

**public** Student() {}

**public** Student(**int** id)

     {

**this**.id = id;

         }

**public** **int** getId()

     {

**return** id;

         }

**public** **void** setId(**int** id)

     {

**this**.id = id;

         }

**public** String getName()

     {

**return** name;

         }

**public** **void** setName(String name)

     {

**this**.name = name;

         }

**public** **long** getFees()

     {

**return** fees;

         }

**public** **void** setFees (**long** fees)

     {

**this**.fees = fees;

     }

}

**JPA Entity Manager**

1. Entity Manager

ေအာက္ေဖာ္ၿပတဲ့အခ်က္မ်ားဟာ entity manager တစ္ခုရဲ႕ အခ်ိဳ႕ေသာအေရးၾကီးတဲ့ roles မ်ားၿဖစ္ပါတယ္။

1. Entity manager က api ကို implements လုပ္ပါတယ္ ၿပီးေတာ့ single interface တစ္ခုအတြင္းမွာ encapsulates လုပ္ပါတယ္။
2. Entity manager ကို read, delete and entity ကို write လုပ္ရန္ အသံုးၿပဳပါတယ္။
3. object referenced ကို entity manager က manage လုပ္ပါတယ္။

entity object ကို persiste လုပ္ရန္ steps မ်ား

1) entity manager factory object မ်ားကို create လုပ္ၿခင္း

java.persistence package ထဲမွာ ရွိတဲ့ EntityManagerFactory interface ကို entity manager အား provide လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student\_details");

Persistence - bootstrap class တစ္ခုၿဖစ္ပါတယ္။ သူ႕ကို EntityManagerFactory interface ရယူဖို႔ရန္ အသံုးၿပဳပါတယ္။

createEntityManagerFactory() method - ဒီ method ရဲ႕ role က named persistence unit အတြက္ EntityManagerFactory ကို create လုပ္ဖို႔ရန္ႏွင့္ return လုပ္ဖို႔ရန္ ၿဖစ္ပါတယ္။ ဒီ method မွာ Persistence.xml file ထဲမွာ passed ၿဖစ္တဲ့ persistence unit ရဲ႕ name ပါ၀င္ပါတယ္။

2) factory မွ entity manager ကို ရယူနည္း

EntityManager em=emf.createEntityManager();

EntityManager – EntityManager တစ္ခုဆိုတာက interface တစ္ခုၿဖစ္တယ္။

createEntityManager() method - သူက new application-managed EntityManager ကို create လုပ္ေပးပါတယ္။

3) entity manager ကို initializing လုပ္ၿခင္း

em.getTransaction().begin();

getTransaction() method - ဒီ method က resource-level EntityTransaction object ကို return ၿပန္ေပးပါတယ္။

begin() method - ဒီ method ကို transaction တစ္ခု start လုပ္ဖို႔ အသံုးၿပဳပါတယ္။

4) relational database ဆီသို႔ data တစ္ခုကို persist လုပ္ၿခင္း

em.persist(s1);

persist() - instance managed နဲ႔ persistent ကို ၿပဳလုပ္ဖို႔ရန္ ဒီ method ကို အသံုးၿပဳပါတယ္။ ဒီ method ထဲမွာ entity instance ကို passed လုပ္ပါတယ္။

5) transaction ကို ပိတ္ၿခင္း

em.getTransaction().commit();

6) factory resources မ်ားကို Releasing လုပ္ၿခင္း

emf.close();

em.close();

factory resources ကို releasing လုပ္ရန္ ဒီ method ကို အသံုးၿပဳပါတယ္။

Entity Operations

1. Inserting an Entity
2. Finding an Entity
3. Updating an Entity
4. Deleting an Entity

2. Inserting an Entity

JPA မွာ entities ကိုၿဖတ္သန္းၿပီး database ထဲသို႔ data insert လုပ္ရတာ လြယ္ကူပါတယ္။ EntityManager က insert records လုပ္ရန္ persist() method ကို provide လုပ္ပါတယ္။

JPA Entity Insertion Example

persistence.xml

<persistence-unit name=*"JPAThirdProject"*>

<properties>

<property name=*"javax.persistence.jdbc.driver"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"javax.persistence.jdbc.url"* value=*"jdbc:mysql://localhost:3306/jpa"* />

<property name=*"javax.persistence.jdbc.user"* value=*"root"* />

<property name=*"javax.persistence.jdbc.password"* value=*"root"* />

<property name=*"eclipselink.logging.level"* value=*"SEVERE"* />

<property name=*"eclipselink.ddl-generation"* value=*"create-or-extend-tables"* />

</properties>

<class>jpa\_entity\_manager.StudentEntity</class>

</persistence-unit>

StudentEntity.java (ေနာက္ topic မ်ားၿဖစ္တဲ့ find , update, delete မ်ားအတြက္ common သံုးမွာၿဖစ္ပါတယ္)

@Entity

@Table(name="student")

**public** **class** StudentEntity {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

**public** StudentEntity() {}

**public** StudentEntity(**int** s\_id, String s\_name, **int** s\_age) {

**super**();

**this**.s\_id = s\_id;

**this**.s\_name = s\_name;

**this**.s\_age = s\_age;}

//getter/setter

}

PersistStudent.java

**public** **class** PersistStudent {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentEntity s1 = **new** StudentEntity();

persistence.xml ထဲက

<persistence-unit name="JPAThirdProject">

တူရမယ္

s1.setS\_id(101);

s1.setS\_name("Gaurav");

s1.setS\_age(24);

StudentEntity s2 = **new** StudentEntity();

s2.setS\_id(102);

s2.setS\_name("Ronit");

s2.setS\_age(22);

StudentEntity s3 = **new** StudentEntity();

s3.setS\_id(103);

s3.setS\_name("Rahul");

s3.setS\_age(26);

em.persist(s1);

em.persist(s2);

em.persist(s3);

em.getTransaction().commit();

emf.close();

em.close();}}

3. Finding an entity

entity တစ္ခုကို ရွာရန္ EntityManger interface က find() method ကို provide လုပ္ပါတယ္။ အဲ့ method က primary key တစ္ခုအေပၚ အေၿခခံၿပီး element တစ္ခုကို search လုပ္ပါတယ္။

JPA Entity Finding Example

FindStudent.java

**public** **class** FindStudent {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

StudentEntity s = em.find(StudentEntity.**class**, 101);

System.***out***.println("Student id = " + s.getS\_id());

System.***out***.println("Student Name = " + s.getS\_name());

System.***out***.println("Student Age = " + s.getS\_age());}}

4. Update an Entity

entity ကို update လုပ္ၿခင္းအားၿဖင့္ database ရဲ႕ record ကို ေၿပာင္းလဲေပးပါတယ္။

JPA Entity Finding Example

UpdateStudent.java

**public** **class** UpdateStudent {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

StudentEntity s = em.find(StudentEntity.**class**, 102);

System.***out***.println("Before Updation");

System.***out***.println("Student id = " + s.getS\_id());

System.***out***.println("Student Name = " + s.getS\_name());

System.***out***.println("Student Age = " + s.getS\_age());

s.setS\_age(30);

System.***out***.println("After Updation");

System.***out***.println("Student id = " + s.getS\_id());

System.***out***.println("Student Name = " + s.getS\_name());

System.***out***.println("Student Age = " + s.getS\_age());}}

5. Deleting an Entity

database ထဲက record ကို ဖ်က္ရန္ Entity manager interface က remove() method ကို provide လုပ္ပါတယ္။ primary key ကို အေၿခခံၿပီး ဖ်က္မွာ ၿဖစ္ပါတယ္။

JPA Entity Delete Example

DeleteStudent.java

**public** **class** DeleteStudent {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentEntity s = em.find(StudentEntity.**class**, 102);

em.remove(s);

em.getTransaction().commit();

emf.close();

em.close();}}

**JPA Collection Mapping**

1. Collection Mapping

collection တစ္ခုဆိုတာက java framework တစ္ခုၿဖစ္ပါတယ္။ that groups multiple objects into a single unit.

collection mapping ကို data မ်ား store, retrieve and manipulate လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

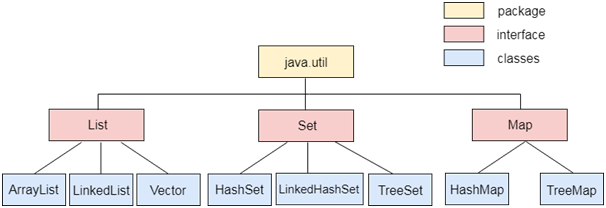
JPA ထဲမွာ ကၽြန္ေတာ္တို႔က collections မ်ားကို အသံုးၿပဳၿပီးေတာ့ wrapper classes မ်ားရဲ႕ object နဲ႔ string ကို persist လုပ္ႏိုင္ပါတယ္။

mapping collections မ်ားထဲမွာ objects မ်ားကို store လုပ္ပံု ၃ မ်ိဳးရွိပါတယ္။ သူတို႔ကေတာ့ Basic Types, Entities နဲ႔ Embeddables တို႔ၿဖစ္ၾကပါတယ္။

Collection Types

objects မ်ားကို persist လုပ္ရန္ မတူညီတဲ့ collection အမ်ိဳးအစားမ်ားကို အသံုးၿပဳပါတယ္။

1. List
2. Set
3. Map



collection framework ရဲ႕ classes မ်ားနဲ႔ interfaces အကုန္လံုးဟာဆိုရင္ java.util package ထဲမွာ ပါ၀င္ပါတယ္။

2. List Mapping

list တစ္ခုဆိုတာက interface တစ္ခုၿဖစ္ပါတယ္။ သူ႕ကို index ကို အေၿခခံၿပီး elements မ်ားကို insert နဲ႔ delete လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။ သူ႕ကို user-defined order ထဲမွာ retrieving elements မ်ားရဲ႕ requirement တစ္ခုရွိတဲ့အခါ အသံုးၿပဳပါတယ္။

List Mapping Example

ကၽြန္ေတာ္တို႔က entity class တစ္ခုထဲမွာ object တစ္ခုကို embed လုပ္ထားပါတယ္ ထို object ကိုလည္း collection list အၿဖစ္သတ္မွတ္ထားပါတယ္။

**private** List<Address> address=**new** ArrayList<Address>();

Employee.java

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** e\_id;

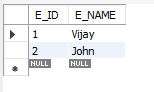
**private** String e\_name;

@ElementCollection

**private** List<Address> address = **new** ArrayList<Address>();

//getter and setter

}



Address.java

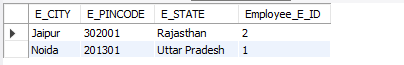
@Embeddable

**public** **class** Address {

**private** **int** e\_pincode;

**private** String e\_city;

**private** String e\_state;



ListMapping.java

**public** **class** ListMapping {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Address a1 = **new** Address();

a1.setE\_pincode(201301);

a1.setE\_city("Noida");

a1.setE\_state("Uttar Pradesh");

Address a2 = **new** Address();

a2.setE\_pincode(302001);

a2.setE\_city("Jaipur");

a2.setE\_state("Rajasthan");

Employee e1 = **new** Employee();

e1.setE\_id(1);

e1.setE\_name("Vijay");

e1.getAddress().add(a1);

Employee e2 = **new** Employee();

e2.setE\_id(2);

e2.setE\_name("John");

e2.getAddress().add(a2);

em.persist(e1);

em.persist(e2);

em.getTransaction().commit();

emf.close();

em.close();}}

Persistence.xml

<class>jpa\_collection\_mapping.list\_mapping.Employee</class>

<class>jpa\_collection\_mapping.list\_mapping.Address</class>

3. JPA Set Mapping

set ဆိုတာက interface တစ္ခု။ သူ႕ထဲမွာ unique ၿဖစ္တဲ့ elements မ်ား ပါ၀င္ပါတယ္။ ထို elements မ်ားဟာ any order ကို မဆို maintain မလုပ္ေဆာင္ပါဘူး။ unordered manner တစ္ခုမွာ unique elements မ်ားကို retrieving လုပ္ဖို႔ လိုအပ္လာတဲ့အခါ set ကို အသံုးၿပဳပါတယ္။

Set Mapping Example

private Set<Address> address=new HashSet<Address>();

Employee1.java

@Entity

**public** **class** Employee1 {

@Id

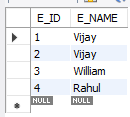
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** e\_id;

**private** String e\_name;

@ElementCollection

**private** Set<Address1> address = **new** HashSet<Address1>();



Address1.java

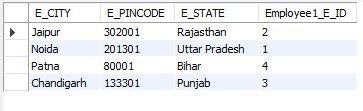
@Embeddable

**public** **class** Address1 {

**private** **int** e\_pincode;

**private** String e\_city;

**private** String e\_state;



SetMapping.java

**public** **class** SetMapping {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Address1 a1 = **new** Address1();

a1.setE\_pincode(201301);

a1.setE\_city("Noida");

a1.setE\_state("Uttar Pradesh");

Address1 a2 = **new** Address1();

a2.setE\_pincode(302001);

a2.setE\_city("Jaipur");

a2.setE\_state("Rajasthan");

Address1 a3 = **new** Address1();

a3.setE\_pincode(133301);

a3.setE\_city("Chandigarh");

a3.setE\_state("Punjab");

Address1 a4 = **new** Address1();

a4.setE\_pincode(80001);

a4.setE\_city("Patna");

a4.setE\_state("Bihar");

Employee1 e1 = **new** Employee1();

e1.setE\_id(1);

e1.setE\_name("Vijay");

Employee1 e2 = **new** Employee1();

e2.setE\_id(2);

e2.setE\_name("Vijay");

Employee1 e3 = **new** Employee1();

e3.setE\_id(3);

e3.setE\_name("William");

Employee1 e4 = **new** Employee1();

e4.setE\_id(4);

e4.setE\_name("Rahul");

e1.getAddress().add(a1);

e2.getAddress().add(a2);

e3.getAddress().add(a3);

e4.getAddress().add(a4);

em.persist(e1);

em.persist(e2);

em.persist(e3);

em.persist(e4);

em.getTransaction().commit();

em.close();

emf.close();}}

4. JPA Map Mapping

map ဆိုတာက interface တစ္ခုၿဖစ္ၿပီးေတာ့ သူက unique key တစ္ခုခ်င္းစီက value object တစ္ခုခ်င္းစီတိုင္းနဲ႔ associate ၿဖစ္ေနတယ္။ ဒါေၾကာင့္ သူ႕ကို သံုးတဲ့အခါ operations အမ်ားစုက key ကို အေၿခခံၿပီး လုပ္ေဆာင္ပါတယ္။

Map Mapping Example

**private** Map<Integer,Address> map=**new** HashMap<Integer,Address>();

Employee2.java

@Entity

**public** **class** Employee2 {

@Id

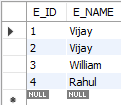
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** e\_id;

**private** String e\_name;

@ElementCollection

**private** Map<Integer, Address2> map = **new** HashMap<Integer, Address2>();



Address2.java

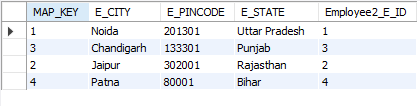
@Embeddable

**public** **class** Address2 {

**private** **int** e\_pincode;

**private** String e\_city;

**private** String e\_state;



MapMapping.java

**public** **class** MapMapping {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Address2 a1 = **new** Address2();

a1.setE\_pincode(201301);

a1.setE\_city("Noida");

a1.setE\_state("Uttar Pradesh");

Address2 a2 = **new** Address2();

a2.setE\_pincode(302001);

a2.setE\_city("Jaipur");

a2.setE\_state("Rajasthan");

Address2 a3 = **new** Address2();

a3.setE\_pincode(133301);

a3.setE\_city("Chandigarh");

a3.setE\_state("Punjab");

Address2 a4 = **new** Address2();

a4.setE\_pincode(80001);

a4.setE\_city("Patna");

a4.setE\_state("Bihar");

Employee2 e1 = **new** Employee2();

e1.setE\_id(1);

e1.setE\_name("Vijay");

Employee2 e2 = **new** Employee2();

e2.setE\_id(2);

e2.setE\_name("Vijay");

Employee2 e3 = **new** Employee2();

e3.setE\_id(3);

e3.setE\_name("William");

Employee2 e4 = **new** Employee2();

e4.setE\_id(4);

e4.setE\_name("Rahul");

e1.getMap().put(1, a1);

e2.getMap().put(2, a2);

e3.getMap().put(3, a3);

e4.getMap().put(4, a4);

em.persist(e1);

em.persist(e2);

em.persist(e3);

em.persist(e4);

em.getTransaction().commit();

em.close();

emf.close();}}

**JPA Type of Mapping**

1. One-To-One Mapping

 One-To-One mapping က single-valued association ကို ကိုယ္စားၿပဳတယ္။ ဆိုလိုတာက entity တစ္ခုရဲ႕ instance က ေနာက္ထပ္ entity တစ္ခုရဲ႕ instance နဲ႔ associate ၿဖစ္ေနၿခင္းကို ဆိုလိုတယ္။

1. Student က One
2. Library လည္း One

StudentOTO.java

@Entity

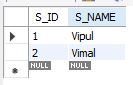
**public** **class** StudentOTO {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** s\_id;

**private** String s\_name;



LibraryOTO.java

@Entity

**public** **class** LibraryOTO {

@Id

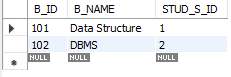
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** b\_id;

**private** String b\_name;

@OneToOne

**private** StudentOTO stud;



OneToOneExample.java

**public** **class** OneToOneExample {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*( "JPAThirdProject" );

EntityManager em = emf.createEntityManager( );

em.getTransaction( ).begin( );

StudentOTO st1=**new** StudentOTO();

st1.setS\_id(1);

st1.setS\_name("Vipul");

StudentOTO st2=**new** StudentOTO();

st2.setS\_id(2);

st2.setS\_name("Vimal");

em.persist(st1);

em.persist(st2);

LibraryOTO lib1=**new** LibraryOTO();

lib1.setB\_id(101);

lib1.setB\_name("Data Structure");

lib1.setStud(st1);

LibraryOTO lib2=**new** LibraryOTO();

lib2.setB\_id(102);

lib2.setB\_name("DBMS");

lib2.setStud(st2);

em.persist(lib1);

em.persist(lib2);

em.getTransaction().commit();

em.close();

emf.close(); } }

2. One-To-Manay Mapping

One to Many mapping က collection-valued association ကို ညႊန္းဆိုတယ္။ entity တစ္ခုက အၿခား entities မ်ားရဲ႕ collection တစ္ခုနဲ႔ ဆက္စပ္ေနတယ္။ entity တစ္ခုရဲ႕ instance တစ္ခုဟာ အၿခား entities မ်ားရဲ႕ instance မ်ားနဲ႔ ဆက္စပ္ေနႏိုင္ပါတယ္။

@OneToMany Example

1. Student က one
2. Library က many

StudentOTM.java

@Entity

**public** **class** StudentOTM {

@Id

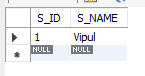
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** s\_id;

**private** String s\_name;

@OneToMany(targetEntity = LibraryOTM.**class**)

**private** List books\_issued;



LibraryOTO.java

@Entity

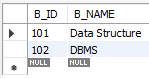
**public** **class** LibraryOTM {

@Id

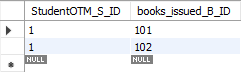
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** b\_id;

**private** String b\_name;



Detail Table



3. Many-To-One Mapping

 Many-To-One mapping က single-valued association တစ္ခုကို ကိုယ္စားၿပဳပါတယ္။ entities မ်ားရဲ႕ collection တစ္ခုက similar entity တစ္ခုနဲ႔ associate ၿဖစ္တယ္။ db ထဲမွာ entity တစ္ခုရဲ႕ တစ္ခုထက္ပိုတဲ့ row တစ္ေၾကာင္းဟာ တစ္ၿခား entity တစ္ခုရဲ႕ similar rows မ်ားကို refer လုပ္ႏိုင္ပါတယ္။

@ManyToOne Example

1. Student က Many
2. Library က One

StudentMTO.java

@Entity

**public** **class** StudentMTO {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** s\_id;

**private** String s\_name;

@ManyToOne

**private** LibraryMTO lib;

LibraryMTO.java

@Entity

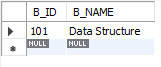
**public** **class** LibraryMTO {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** b\_id;

**private** String b\_name;



ManyToOneExample.java

**public** **class** ManyToOneExample {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

LibraryMTO lib1 = **new** LibraryMTO();

lib1.setB\_id(101);

lib1.setB\_name("Data Structure");

em.persist(lib1);

StudentMTO st1 = **new** StudentMTO();

st1.setS\_id(1);

st1.setS\_name("Vipul");

st1.setLib(lib1);

StudentMTO st2 = **new** StudentMTO();

st2.setS\_id(2);

st2.setS\_name("Vimal");

st2.setLib(lib1);

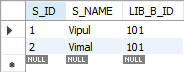
em.persist(st1);

em.persist(st2);

em.getTransaction().commit();

em.close();

emf.close();}}



4. Many-To-Many Mapping

Many-To-Many mapping က  collection-valued association တစ္ခုကို ကိုယ္စားၿပဳပါတယ္။ entities မ်ားက အၿခား entities မ်ားရဲ႕ collection တစ္ခုနဲ႔ associate လုပ္ႏိုင္ပါတယ္။ db မွာ entity တစ္ခုရဲ႕ rows မ်ားဟာ တစ္ၿခား entity တစ္ခုရဲ႕ rows မ်ားကို referred လုပ္ႏိုင္ပါတယ္။

@ManyToOne Example

1. Student ေရာ Library ေရာ many

StudentMTM.java

@Entity

**public** **class** StudentMTM {

@Id

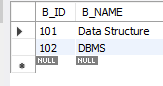
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** s\_id;

**private** String s\_name;

@ManyToMany(targetEntity = LibraryMTM.**class**)

**private** List lib;



LibraryMTM.java

@Entity

**public** **class** LibraryMTM {

@Id

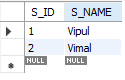
@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** b\_id;

**private** String b\_name;

@ManyToMany(targetEntity = StudentMTM.**class**)

**private** List stud;



ManyToManyExample.java

**public** **class** ManyToManyExample {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentMTM st1 = **new** StudentMTM(1, "Vipul", **null**);

StudentMTM st2 = **new** StudentMTM(2, "Vimal", **null**);

em.persist(st1);

em.persist(st2);

ArrayList<StudentMTM> al1 = **new** ArrayList<StudentMTM>();

ArrayList<StudentMTM> al2 = **new** ArrayList<StudentMTM>();

al1.add(st1);

al1.add(st2);

al2.add(st1);

al2.add(st2);

LibraryMTM lib1 = **new** LibraryMTM(101, "Data Structure", al1);

LibraryMTM lib2 = **new** LibraryMTM(102, "DBMS", al2);

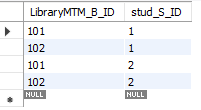
em.persist(lib1);

em.persist(lib2);

em.getTransaction().commit();

em.close();

emf.close();}}



**JPA Cascading**

1. Cascading Operations

JPA မွာ မည္သည့္ operation ကိုမဆို entity တစ္ခုမွာ အသံုးၿပဳခဲ့မယ္ဆိုရင္ ထို entity တစ္ခုတည္းေပၚမွာ သက္ေရာက္မွာ ၿဖစ္တယ္။ ထို operations မ်ားသည္ တစ္ၿခား entities မ်ားနဲ႔ သက္ဆိုင္မွာ မဟုတ္ဘူး။

သက္ဆိုင္ရာ entities မ်ားၾကားထဲမွာ dependency တစ္ခုကို တည္ေဆာက္ရန္ JPA က **javax.persistence.CascadeType** ဆိုတဲ့ enumerated types မ်ားကို သတ္မွတ္ေပးထားပါတယ္။ cascading operations မ်ားကို မည္သည့္ mapping type မ်ိဳးနဲ႔မဆို သတ္မွတ္ႏိုင္ပါတယ္။  i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

Cascade Enums

|  |  |
| --- | --- |
| Cascade Operations | Description |
| PERSIST | parent entity က persist လုပ္တယ္ဆိုရင္ related entity မ်ားကလည္း persist လုပ္ၾကတယ္။ |
| MERGE | parent entity က merge လုပ္တယ္ဆိုရင္ related entity မ်ားကလည္း merge လုပ္ၾကတယ္။ |
| DETACH | parent entity က detache လုပ္တယ္ဆိုရင္ related entity မ်ားကလည္း detache လုပ္ၾကတယ္။ |
| REFRESH | parent entity က refresh လုပ္တယ္ဆိုရင္ related entity မ်ားကလည္း refresh လုပ္ၾကတယ္။ |
| REMOVE | parent entity က remove လုပ္တယ္ဆိုရင္ related entity မ်ားကလည္း remove လုပ္ၾကတယ္။ |
| ALL | အထက္မွာ ေဖာ္ၿပတဲ့ operations မ်ားကို entities မ်ားမွာ အသံုးၿပဳခဲ့မယ္ဆိုရင္ သူတို႔နဲ႔သက္ဆိုင္တဲ့ parent entities မ်ားတြင္လည္း သက္ေရာက္မွာ ၿဖစ္တယ္။ |

2. Cascade Persist

entity တစ္ခုက persist လုပ္ၿခင္း ခံလိုက္ရတယ္ဆိုရင္ သူနဲ႔သက္ဆိုင္တဲ့ child entities မ်ားကလည္း persist လုပ္ၿခင္း ခံရမွာ ၿဖစ္ပါတယ္။

@OneToOne(cascade=CascadeType.PERSIST)

Cascade Persist Example

StudentPersistRemove.java

@Entity

@Table(name = "student\_persist\_remove")

**public** **class** StudentPersistRemove {

@Id

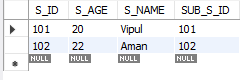
**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

@OneToOne(cascade = CascadeType.***PERSIST***)

**private** SubjectPersistRemove sub;



SubjectPersistRemove.java

@Entity

@Table(name = "subject\_persist\_remove")

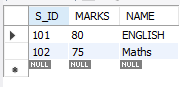
**public** **class** SubjectPersistRemove {

**private** String name;

**private** **int** marks;

@Id

**private** **int** s\_id;



StudentCascade.java

**public** **class** StudentCascade {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentPersistRemove s1 = **new** StudentPersistRemove();

s1.setS\_id(101);

s1.setS\_name("Vipul");

s1.setS\_age(20);

StudentPersistRemove s2 = **new** StudentPersistRemove();

s2.setS\_id(102);

s2.setS\_name("Aman");

s2.setS\_age(22);

SubjectPersistRemove sb1 = **new** SubjectPersistRemove();

sb1.setName("ENGLISH");

sb1.setMarks(80);

sb1.setS\_id(s1.getS\_id());

SubjectPersistRemove sb2 = **new** SubjectPersistRemove();

sb2.setName("Maths");

sb2.setMarks(75);

sb2.setS\_id(s2.getS\_id());

s1.setSub(sb1);

s2.setSub(sb2);

em.persist(s1);// No need to perform persist operation separately for different entities.

em.persist(s2);

em.getTransaction().commit();

em.close();

emf.close();}}

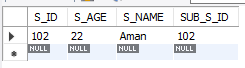
3. Cascade Remove

entity တစ္ခုက remove လုပ္ၿခင္း ခံလိုက္ရတယ္ဆိုရင္ သူနဲ႔သက္ဆိုင္တဲ့ child entities မ်ားကလည္း remove လုပ္ၿခင္း ခံရမွာ ၿဖစ္ပါတယ္။

@OneToOne(cascade=CascadeType. REMOVE)

အေပၚက StudentPersistRemove class မွာ persist ေနရာ remove က ထည့္လိုက္ရံုပဲ။

@OneToOne(cascade = CascadeType.***REMOVE***)



StudentCascade.java

**public** **class** StudentCascade {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

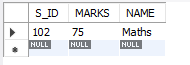
StudentPersistRemove s = em.find(StudentPersistRemove.**class**, 101);

em.remove(s);

em.getTransaction().commit();

em.close();

emf.close();}}



**JPA JPQL**

1. JPQL Introduction

 JPQL (Java Persistence Query Language) ဆိုတာက object-oriented query language တစ္ခုၿဖစ္တယ္။ သူ႕ကို persistence entities မ်ားမွာ database operations မ်ားလုပ္ေဆာင္ဖို႔ရန္ အသံုးၿပဳပါတယ္။ JPQL က entity object model မ်ားကို SQL queries မ်ားကို operate လုပ္ရန္ database table အစား အသံုးၿပဳပါတယ္။ JPQL က developers မ်ားအေနနဲ႔ SQL tasks မ်ားကို လြယ္လြယ္ကူကူ handle လုပ္ေဆာင္ႏိုင္ရန္ platform တစ္ခု ထုတ္ေပးပါတယ္။

JPQL က Entity JavaBeans Query Language (EJBQL) ရဲ႕ extension  တစ္ခုၿဖစ္ပါတယ္။

1. join operations မ်ားလုပ္ေဆာင္ႏိုင္တယ္။
2. bulk တစ္ခုထဲမွာ data မ်ားကို update နဲ႔ delete လုပ္ႏိုင္တယ္။
3. sorting and grouping clauses မ်ားႏွင့္ aggregate function ကို လုပ္ေဆာင္ႏိုင္တယ္။
4. Single and multiple value result types

JPQL Features

platform-independent query language တစ္ခုၿဖစ္တယ္။

simple နဲ႔ robust ၿဖစ္တယ္။

မည္သည့္ dabase type မ်ိဳးနဲ႔ မဆိုအသံုးၿပဳႏိုင္တယ္။ ဥပမာ MySQL, Oracle.

JPQL ကို metadata ထဲမွာ statically  ေၾကညာၿပီးေတာ့ code ထဲမွာေတာ့ dynamically built လုပ္ပါတယ္။

Creating Queries in JPQL

JPQL က database records မ်ားကို access လုပ္ရန္ methods ၂ ခုကို ထုတ္ေပးပါတယ္။

1. Query createQuery(String name) - EntityManager interface ရဲ႕ method ၿဖစ္ပါတယ္။ သူ႕ကို executing JPQL statement လုပ္ဖို႔ရန္ အတြက္ Query interface ရဲ႕ instance ကို create လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

Query query = em.createQuery("Select s.s\_name from StudentEntity s");

ဒီ method ကေတာ့ dynamic query မ်ားကို create လုပ္ပါတယ္။ သူ႕ကို business logic အတြင္းမွာ သတ္မွတ္ႏိုင္တယ္။

2. Query createNamedQuery(String name) - EntityManager interface ရဲ႕ method ၿဖစ္တယ္။ သူ႕ကို executing named queries လုပ္ဖို႔ရန္အတြက္ Query interface ရဲ႕ instance ကို create လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

@NamedQuery(name = "find name" , query = "Select s from StudentEntity s")

ဒီ method ကေတာ့ static query မ်ား ကို create လုပ္ပါတယ္။ သူ႕ကို entity class အတြင္းမွာ သတ္မွတ္ႏိုင္တယ္။

ကၽြန္ေတာ္တို႔က ေအာက္ပါ query interface methods မ်ားကို အသံုးၿပဳၿခင္းအားၿဖင့္ query ရဲ႕ execution မ်ားကို control လုပ္ႏိုင္ပါတယ္။

int executeUpdate() - This method executes the update and delete operation.

int getFirstResult() - This method returns the first positioned result the query object was set to retrieve.

int getMaxResults() - This method returns the maximum number of results the query object was set to retrieve.

java.util.List getResultList() - This method returns the list of results as an untyped list.

Query setFirstResult(int startPosition) - This method assigns the position of first result to retrieve.

Query setMaxResults(int maxResult) - This method assigns the maximum numbers of result to retrieve.

2. JPQL Basic Operations

JPQL က static ေရာ dynamic မွာေရာ ႏွစ္ခုလံုးမွာ လုပ္ေဆာင္ႏိုင္တယ္။

Dynamic Query Example

StudentDynamicQuery.java

@Entity

@Table(name = "student\_dynamic\_query")

**public** **class** StudentDynamicQuery {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

FetchColumn.java

**public** **class** FetchColumn {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentDynamicQuery s1 = **new** StudentDynamicQuery();

s1.setS\_id(1);

s1.setS\_name("Lyan Kyaw");

s1.setS\_age(26);

StudentDynamicQuery s2 = **new** StudentDynamicQuery();

s2.setS\_id(2);

s2.setS\_name("Lyan Kyaw1");

s2.setS\_age(26);

StudentDynamicQuery s3 = **new** StudentDynamicQuery();

s3.setS\_id(3);

s3.setS\_name("Lyan Kyaw2");

s3.setS\_age(26);

StudentDynamicQuery s4 = **new** StudentDynamicQuery();

s4.setS\_id(4);

s4.setS\_name("Lyan Kyaw3");

s4.setS\_age(26);

StudentDynamicQuery s5 = **new** StudentDynamicQuery();

s5.setS\_id(5);

s5.setS\_name("Lyan Kyaw4");

s5.setS\_age(26);

em.persist(s1);

em.persist(s2);

em.persist(s3);

em.persist(s4);

em.persist(s5);

em.getTransaction().commit();

Query query = em.createQuery("Select s.s\_name from StudentDynamicQuery s");

@SuppressWarnings("unchecked")

List<String> list = query.getResultList();

System.***out***.println("Student Name :");

**for** (String s : list) {

System.***out***.println(s);}

em.close();

emf.close();}}

persistence.xml

<class>jpa\_jpql.basic\_operations.StudentDynamicQuery</class>

Static Query Example

StudentStaticeQuery.java

@Entity

@Table(name = "student\_static\_query")

@NamedQuery(name = "find name", query = "Select s from StudentStaticeQuery s")

**public** **class** StudentStaticeQuery {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

FetchColumn1.java

**public** **class** FetchColumn1 {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentStaticeQuery s1 = **new** StudentStaticeQuery();

s1.setS\_id(1);

s1.setS\_name("Lyan Kyaw");

s1.setS\_age(26);

StudentStaticeQuery s2 = **new** StudentStaticeQuery();

s2.setS\_id(2);

s2.setS\_name("Lyan Kyaw1");

s2.setS\_age(26);

StudentStaticeQuery s3 = **new** StudentStaticeQuery();

s3.setS\_id(3);

s3.setS\_name("Lyan Kyaw2");

s3.setS\_age(26);

StudentStaticeQuery s4 = **new** StudentStaticeQuery();

s4.setS\_id(4);

s4.setS\_name("Lyan Kyaw3");

s4.setS\_age(26);

StudentStaticeQuery s5 = **new** StudentStaticeQuery();

s5.setS\_id(5);

s5.setS\_name("Lyan Kyaw4");

s5.setS\_age(26);

em.persist(s1);

em.persist(s2);

em.persist(s3);

em.persist(s4);

em.persist(s5);

em.getTransaction().commit();

Query query = em.createNamedQuery("find name");

@SuppressWarnings("unchecked")

List<StudentStaticeQuery> list = query.getResultList();

System.***out***.println("Student Name :");

**for** (StudentStaticeQuery s : list) {

System.***out***.println(s.getS\_name());}}}

persistence.xml

<class>jpa\_jpql.basic\_operations.StudentStaticeQuery</class>

3. JPQL Bulk Data Operations

အေပၚက တစ္ခုက column တစ္ခုတည္းကိုပဲ output ထုတ္ၿခင္းၿဖစ္ၿပီး bulk data operations မွာဆိုရင္ေတာ့ data ေတြကို row by row ထုတ္မွာ ၿဖစ္တယ္။

JPQL Bulk Data Example

1. Bulk Fetch

StudentBulkFetch.java

@Entity

@Table(name = "student\_bulk\_fetch")

**public** **class** StudentBulkFetch {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

BulkFetch.java

**public** **class** BulkFetch {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StudentBulkFetch s1 = **new** StudentBulkFetch();

s1.setS\_id(1);

s1.setS\_name("Lyan Kyaw");

s1.setS\_age(26);

StudentBulkFetch s2 = **new** StudentBulkFetch();

s2.setS\_id(2);

s2.setS\_name("Lyan Kyaw1");

s2.setS\_age(26);

StudentBulkFetch s3 = **new** StudentBulkFetch();

s3.setS\_id(3);

s3.setS\_name("Lyan Kyaw2");

s3.setS\_age(26);

StudentBulkFetch s4 = **new** StudentBulkFetch();

s4.setS\_id(4);

s4.setS\_name("Lyan Kyaw3");

s4.setS\_age(26);

StudentBulkFetch s5 = **new** StudentBulkFetch();

s5.setS\_id(5);

s5.setS\_name("Lyan Kyaw4");

s5.setS\_age(26);

em.persist(s1);

em.persist(s2);

em.persist(s3);

em.persist(s4);

em.persist(s5);

Query query = em.createQuery("Select s from StudentBulkFetch s ");

@SuppressWarnings("unchecked")

List<StudentBulkFetch> list = (List<StudentBulkFetch>) query.getResultList();

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentBulkFetch s : list) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.print("\t" + s.getS\_age());

System.***out***.println();}

em.getTransaction().commit();

em.close();

emf.close();}}

2. BulkUpdate

**public** **class** BulkUpdate {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Query query = em.createQuery("update StudentBulkFetch SET s\_age=25 where s\_id>3");

query.executeUpdate();

em.getTransaction().commit();

em.close();

emf.close();}}

3. BulkDelete

**public** **class** BulkDelete {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Query query = em.createQuery("delete from StudentBulkFetch where s\_id=3");

query.executeUpdate();

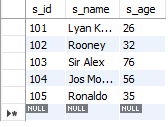
em.getTransaction().commit();

em.close();

emf.close();}}

4. JPQL Advanced Operations

ပထမဆံုးအေနနဲ႔ table ထဲမွာ ေအာက္ကအတိုင္း data မ်ားကို ထည့္ထားပါ။



StudentJPQLAdvancedOptions.java

@Entity

@Table(name = "student\_jpql\_advanced\_op")

**public** **class** StudentJPQLAdvancedOptions {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

JPQL Filter

Filter.java

**public** **class** Filter {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Query q1 = em.createQuery("Select s from StudentJPQLAdvancedOptions s where s.s\_age between 22 and 28");

@SuppressWarnings("unchecked")

List<StudentJPQLAdvancedOptions> l1 = (List<StudentJPQLAdvancedOptions>) q1.getResultList();

System.***out***.println("Between Clause");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentJPQLAdvancedOptions s : l1) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

Query q2 = em.createQuery("Select s from StudentJPQLAdvancedOptions s where s.s\_age IN(35,22,23)");

@SuppressWarnings("unchecked")

List<StudentJPQLAdvancedOptions> l2 = (List<StudentJPQLAdvancedOptions>) q2.getResultList();

System.***out***.println("IN Clause");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentJPQLAdvancedOptions s : l2) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

Query q3 = em.createQuery("Select s from StudentJPQLAdvancedOptions s where s.s\_name like '%a%'");

@SuppressWarnings("unchecked")

List<StudentJPQLAdvancedOptions> l3 = (List<StudentJPQLAdvancedOptions>) q3.getResultList();

System.***out***.println("Like Clause");

System.***out***.print("s\_id");

System.***out***.print("\t \t s\_name");

System.***out***.println("\t \t s\_age");

**for** (StudentJPQLAdvancedOptions s : l3) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t \t" + s.getS\_name());

System.***out***.println("\t \t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

JPQL Aggregate

Aggregate.java

**public** **class** Aggregate {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Query q1 = em.createQuery("Select count(s) from StudentJPQLAdvancedOptions s");

System.***out***.println("Number of Student : " + q1.getSingleResult());

Query q2 = em.createQuery("Select MAX(s.s\_age) from StudentJPQLAdvancedOptions s");

System.***out***.println("Maximum age : " + q2.getSingleResult());

Query q3 = em.createQuery("Select MIN(s.s\_age) from StudentJPQLAdvancedOptions s");

System.***out***.println("Minimum age : " + q3.getSingleResult());

em.getTransaction().commit();

em.close();

emf.close();}}

JPQL Sorting

Sorting.java

**public** **class** Sorting {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Query q1 = em.createQuery("Select s from StudentJPQLAdvancedOptions s order by s.s\_age desc");

@SuppressWarnings("unchecked")

List<StudentJPQLAdvancedOptions> l1 = (List<StudentJPQLAdvancedOptions>) q1.getResultList();

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentJPQLAdvancedOptions s : l1) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

**Criteria API**

1. API Introduction

Criteria API ဆိုတာက entites မ်ား နဲ႔ သူ႕တို႔ရဲ႕ persistence state အတြက္ queries မ်ားကို တည္ေဆာက္တဲ့ေနရာမွာ အသံုးၿပဳတဲ့ common ways မ်ား ၿဖစ္ပါတယ္။ defining JPA queries အတြက္ alternative method တစ္ခုၿဖစ္ပါတယ္။

Criteria API ကို အဓိပၸါယ္သတ္မွတ္ရမယ္ဆိုရင္ platform-independent criteria queries မ်ားကို Java programming language ထဲမွာ ေရးသားၿခင္း ၿဖစ္တယ္။ JPA 2.0 မွာ စၿပီး ပါလာတယ္။ အဓိက ရည္ရြယ္ခ်က္ကေတာ့ type-safe way to express a query တစ္ခုကို provide လုပ္ဖို႔အတြက္ ၿဖစ္ပါတယ္။

Criteria Query ကို create လုပ္ဖို႔ step မ်ား

* **EntityManager** interfaceေပၚမွာ **getCriteriaBuilder()** method ကို invoke လုပ္ၿခင္းအားၿဖင့္ **CriteriaBuilder** interface object တစ္ခုကို create လုပ္ပါမယ္။

EntityManager em = emf.createEntityManager();

CriteriaBuilder cb=em.getCriteriaBuilder();

* query object တစ္ခုကို create လုပ္ဖို႔ရန္ **CriteriaQuery** interface ရဲ႕ instance တစ္ခုကို build လုပ္ပါမယ္။

CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.**class**);

* query root ထည့္ရန္ **CriteriaQuery** object မွတစ္ဆင့္ from method ကို invoke လုပ္မယ္။

Root<StudentEntity> stud=cq.from(StudentEntity.**class**);

* query result ရဲ႕ type ကို specify လုပ္ရန္ CriteriaQuery Object ရဲ႕ select method ကို လွမ္းေခၚရမယ္။

CriteriaQuery<StudentEntity> select = cq.select(stud);

* instance of Query interface ကို create လုပ္ရန္နဲ႔ specify the type of method used to access the database records

Query q = em.createQuery(select);

* Query Interface ရဲ႕ mthod မ်ားကို call လုပ္ၿခင္းအားၿဖင့္ execution of query ကို control လုပ္ႏိုင္တယ္။

List<StudentEntity> list = q.getResultList();

Methods of Criteria API Query Clauses

|  |  |  |
| --- | --- | --- |
| Clause | **Criteria API Interface** | **Methods** |
| SELECT | CriteriaQuery | select() |
| FROM | AbstractQuery | from() |
| WHERE | AbstractQuery | where() |
| ORDER BY | CriteriaQuery | orderBy() |
| GROUP BY | AbstractQuery | groupBy() |
| HAVING | AbstractQuery | having() |

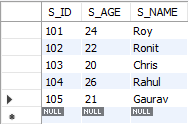
**CriteriaQuery interface က ဆိုရင္ၿဖင့္ AbstractQuery interface ရဲ႕ sub interface တစ္ခုၿဖစ္ပါတယ္။**

2. Criteria SELECT Clause

select clause ကို database ထဲက data မ်ားကို fetch လုပ္ရန္ အသံုးၿပဳတယ္။ single expression သို႔မဟုတ္ multiple expressions စသၿဖင့္ data မ်ားကို retrieved လုပ္ေဆာင္ႏိုင္ပါတယ္။

Criteria SELECT Example

ပထမဆံုးအေနနဲ႔ data မ်ားကို အရင္ၾကိဳထည့္ထားဖို႔ေတာ့ လိုပါတယ္။



StudentCritriaSelect.java

@Entity

@Table(name = "student\_critria\_select")

public class StudentCritriaSelect {

@Id

private int s\_id;

private String s\_name;

private int s\_age;

persistence.xml

<class>jpa\_critria\_api.select.StudentCritriaSelect</class>

**Selecting Single Expression**

**SingleFetch.java**

**public** **class** SingleFetch {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<StudentCritriaSelect> cq = cb.createQuery(StudentCritriaSelect.**class**);

Root<StudentCritriaSelect> stud = cq.from(StudentCritriaSelect.**class**);

cq.multiselect(stud.get("s\_name"));

CriteriaQuery<StudentCritriaSelect> select = cq.select(stud);

TypedQuery<StudentCritriaSelect> q = em.createQuery(select);

List<StudentCritriaSelect> list = q.getResultList();

System.***out***.println("s\_id");

**for** (StudentCritriaSelect s : list) {

System.***out***.println(s.getS\_id());}

em.getTransaction().commit();

em.close();

emf.close();}}

**Selecting Multiple Expression**

MultiFetch.java

**public** **class** MultiFetch {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<StudentCritriaSelect> cq = cb.createQuery(StudentCritriaSelect.**class**);

Root<StudentCritriaSelect> stud = cq.from(StudentCritriaSelect.**class**);

cq.multiselect(stud.get("s\_id"), stud.get("s\_name"), stud.get("s\_age"));

CriteriaQuery<StudentCritriaSelect> select = cq.select(stud);

TypedQuery<StudentCritriaSelect> q = em.createQuery(select);

List<StudentCritriaSelect> list = q.getResultList();

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentCritriaSelect s : list) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}}}

3. Criteria ORDER BY Clause

ORDER BY ကို သံုၿပီးေတာ့ data ေတြကို asc or desc အလုိက္ arrange လုပ္ႏိုင္တယ္။

ORDER BY Example

Sorting in ascending order

Asc.java

public class Asc {

public static void main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<StudentCritriaSelect> cq = cb.createQuery(StudentCritriaSelect.class);

Root<StudentCritriaSelect> stud = cq.from(StudentCritriaSelect.class);

cq.orderBy(cb.asc(stud.get("s\_age")));

CriteriaQuery<StudentCritriaSelect> select = cq.select(stud);

TypedQuery<StudentCritriaSelect> q = em.createQuery(select);

List<StudentCritriaSelect> list = q.getResultList();

System.*out*.print("s\_id");

System.*out*.print("\t s\_name");

System.*out*.println("\t s\_age");

for (StudentCritriaSelect s : list) {

System.*out*.print(s.getS\_id());

System.*out*.print("\t" + s.getS\_name());

System.*out*.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

Sorting in descending order

Desc.java

public class Desc {

public static void main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<StudentCritriaSelect> cq = cb.createQuery(StudentCritriaSelect.class);

Root<StudentCritriaSelect> stud = cq.from(StudentCritriaSelect.class);

cq.orderBy(cb.desc(stud.get("s\_age")));

CriteriaQuery<StudentCritriaSelect> select = cq.select(stud);

TypedQuery<StudentCritriaSelect> q = em.createQuery(select);

List<StudentCritriaSelect> list = q.getResultList();

System.*out*.print("s\_id");

System.*out*.print("\t s\_name");

System.*out*.println("\t s\_age");

for (StudentCritriaSelect s : list) {

System.*out*.print(s.getS\_id());

System.*out*.print("\t" + s.getS\_name());

System.*out*.println("\t" + s.getS\_age());}

em.getTransaction().commit();

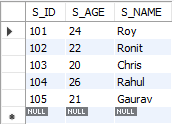
em.close();

emf.close();}}

4. Criteria WHERE Clause

database မွာ conditions မ်ားကို သံုးခ်င္ရင္ where clause မ်ားကို အသံုးၿပဳပါတယ္။ data မ်ားကို ဆြဲရင္ အဲ့ condition အေပၚမူတည္ၿပီး ဆြဲပါတယ္။ conditions မ်ားကို ထည့္ရန္ AbstractQuery interface ရဲ႕ where() method ကို အသံုးၿပဳပါတယ္။

***student\_order\_by***



***StudentOrderBy.java***

@Entity

@Table(name = "student\_order\_by")

**public** **class** StudentOrderBy {

@Id

**private** **int** s\_id;

**private** String s\_name;

**private** **int** s\_age;

***persistence.xml***

<class>jpa\_critria\_api.StudentOrderBy</class>

JPQL Greater Than and Less Than

***Comparison.java***

**public** **class** Comparison {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

AbstractQuery<StudentOrderBy> cq1 = cb.createQuery(StudentOrderBy.**class**);

AbstractQuery<StudentOrderBy> cq2 = cb.createQuery(StudentOrderBy.**class**);

Root<StudentOrderBy> stud1 = cq1.from(StudentOrderBy.**class**);

cq1.where(cb.greaterThan(stud1.<Integer>get("s\_age"), 22));

CriteriaQuery<StudentOrderBy> select1 = ((CriteriaQuery<StudentOrderBy>) cq1).select(stud1);

TypedQuery<StudentOrderBy> tq1 = em.createQuery(select1);

List<StudentOrderBy> list1 = tq1.getResultList();

System.***out***.println("Students having age greater than 22");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentOrderBy s : list1) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

Root<StudentOrderBy> stud2 = cq2.from(StudentOrderBy.**class**);

cq2.where(cb.lessThan(stud2.<Integer>get("s\_age"), 22));

CriteriaQuery<StudentOrderBy> select2 = ((CriteriaQuery<StudentOrderBy>) cq2).select(stud2);

TypedQuery<StudentOrderBy> tq2 = em.createQuery(select2);

List<StudentOrderBy> list2 = tq2.getResultList();

System.***out***.println("Students having age Less than 22");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentOrderBy s : list2) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

JPQL Between

***Between.java***

**public** **class** Between {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

AbstractQuery<StudentOrderBy> cq = cb.createQuery(StudentOrderBy.**class**);

Root<StudentOrderBy> stud = cq.from(StudentOrderBy.**class**);

cq.where(cb.between(stud.<Integer>get("s\_age"), 22,26));

CriteriaQuery<StudentOrderBy> select = ((CriteriaQuery<StudentOrderBy>) cq).select(stud);

TypedQuery<StudentOrderBy> tq = em.createQuery(select);

List<StudentOrderBy> list = tq.getResultList();

System.***out***.println("Students having age between 22 and 26");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentOrderBy s : list) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

JPQL Like

***Like.java***

**public** **class** Like {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

AbstractQuery<StudentOrderBy> cq = cb.createQuery(StudentOrderBy.**class**);

Root<StudentOrderBy> stud = cq.from(StudentOrderBy.**class**);

cq.where(cb.like(stud.<String>get("s\_name"), "R%"));

CriteriaQuery<StudentOrderBy> select = ((CriteriaQuery<StudentOrderBy>) cq).select(stud);

TypedQuery<StudentOrderBy> tq = em.createQuery(select);

List<StudentOrderBy> list = tq.getResultList();

System.***out***.println("Students name starting with R");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentOrderBy s : list) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

emf.close();}}

**JPQL In**

***In.java***

**public** **class** In {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

AbstractQuery<StudentOrderBy> cq = cb.createQuery(StudentOrderBy.**class**);

Root<StudentOrderBy> stud = cq.from(StudentOrderBy.**class**);

cq.where(cb.in(stud.get("s\_age")).value(22).value(24));

CriteriaQuery<StudentOrderBy> select = ((CriteriaQuery<StudentOrderBy>) cq).select(stud);

TypedQuery<StudentOrderBy> tq = em.createQuery(select);

List<StudentOrderBy> list = tq.getResultList();

System.***out***.println("Students having age 22 and 24");

System.***out***.print("s\_id");

System.***out***.print("\t s\_name");

System.***out***.println("\t s\_age");

**for** (StudentOrderBy s : list) {

System.***out***.print(s.getS\_id());

System.***out***.print("\t" + s.getS\_name());

System.***out***.println("\t" + s.getS\_age());}

em.getTransaction().commit();

em.close();

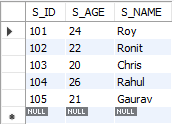
emf.close();}}

5. Criteria GROUP BY Clause

groupby clause ကို group တစ္ခုတည္းမွာ table တစ္ခု သို႔ တစ္ခုထက္ပိုတဲ့ tables ထဲက data မ်ားကို collect လုပ္ဖို႔ရန္ နဲ႔ သူတို႔ကို arrange လုပ္ဖုိ႔ရန္ အတြက္ အသံုးၿပဳပါတယ္။ AbstractQuery interface ရဲ႕ groupBy() method ကို အသံုးၿပဳပါတယ္။

***StudentOrderBy.java*** ကိုပဲ အသံုးၿပဳပါတယ္။

***student\_order\_by***



***StudentGroup.java***

**public** **class** StudentGroup {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Object[]> cq = cb.createQuery(Object[].**class**);

Root<StudentOrderBy> stud = cq.from(StudentOrderBy.**class**);

cq.multiselect(stud.get("s\_age"), cb.count(stud)).groupBy(stud.get("s\_age"));

System.***out***.print("s\_age");

System.***out***.println("\t Count");

List<Object[]> list = em.createQuery(cq).getResultList();

**for** (Object[] object : list) {

System.***out***.println(object[0] + " " + object[1]);}

em.getTransaction().commit();

em.close();

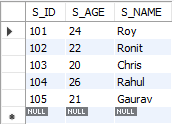
emf.close();}}

5. Criteria Having Clause

Table တစ္ခုတည္းမွာ ရွိတဲ့ data မ်ားကို filter လုပ္ရန္ having ကို GROUP BY clause နဲ႔ တြဲၿပီး အသံုးၿပဳပါတယ္။ Abstract interface ရဲ႕ having method ကို grouped လုပ္ထားေသာ data မ်ားတြင္ conditions မ်ားထည့္ရန္ အသံုးၿပဳပါတယ္။

***StudentOrderBy.java*** ကိုပဲ အသံုးၿပဳပါတယ္။

***student\_order\_by***



***Having.java***

**public** **class** Having {

**public** **static** **void** main(String args[]) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Object[]> cq = cb.createQuery(Object[].**class**);

Root<StudentOrderBy> stud = cq.from(StudentOrderBy.**class**);

cq.multiselect(stud.get("s\_age"), cb.count(stud)).groupBy(stud.get("s\_age"))

.having(cb.ge(stud.<Integer>get("s\_age"), 24));

System.***out***.print("s\_age");

System.***out***.println("\t Count");

List<Object[]> list = em.createQuery(cq).getResultList();

**for** (Object[] object : list) {

System.***out***.println(object[0] + " " + object[1]);}

em.getTransaction().commit();

em.close();

emf.close();}}

**JPA Inheritance**

1. JPA Inheritence Overview

Inheritence က object-oriented programming language ရဲ႕ key feature တစ္ခုၿဖစ္ပါတယ္။ child class က parent class ရဲ႕ properties မ်ားကို ရယူႏိုင္တယ္။ ထို feature ေၾကာင့္ code ေတြရဲ႕ reusability ကို ၿမွင့္တင္ႏိုင္တယ္။

relational database က inheritance ရဲ႕ mechanism  ကို support မလုပ္ႏိုင္ပါ။ ထို႔ေၾကာင့္ Java Persistence API (JPA) ကို relational database model ထဲမွာ inheritance  ရဲ႕ key features မ်ားကို map လုပ္ရန္ အသံုးၿပဳပါတယ္။

JPA Inheritence Annotations

1. **@Inheritence** - ဒီ annotation ကို inheritance strategy ကို define လုပ္ရန္ root entity class မွာ apply လုပ္ပါတယ္။ strategy type ကို မသတ္မွတ္ရေသးဘူးဆိုရင္ single table strategy အၿဖစ္ သတ္မွတ္ပါတယ္။

@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)

1. **@MappedSuperclass –** class တစ္ခုက သူ႕ရဲ႕ superclass ကို inheritance လုပ္ထားတယ္ဆိုရင္ ဒီ annotation ကို အသံုးၿပဳပါတယ္။ mapped superclass မွာ any separate table မွ မပါ၀င္ပါဘူး
2. **@DiscriminatorColumn -** discriminator attribute က entity တစ္ခုကေန တစ္ခုကို differentiates လုပ္ပါတယ္။ ထို႔ေၾကာင့္ ဒီ annotation ကို discriminator column ရဲ႕ name ကို ေပးဖို႔အတြက္ အသံုးၿပဳပါတယ္။ root entity class ထဲမွာပဲ annotation ကို specify လုပ္ရန္ လိုအပ္ပါတယ္။

@Entity

@Table(name = "employee\_2")

@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)

@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.***STRING***)

@DiscriminatorValue(value = "employee2")

**public** **class** Employee2 {

1. **@DiscriminatorValue** - ဒီ annotation ကို value ရဲ႕ type ကို specify လုပ္ရန္ အသံုးၿပဳပါတယ္ သူက particular entity ကို ကိုယ္စားၿပဳပါတယ္။ sub-entity classes မ်ားေပၚမွာ ဒီ annotation ကို specify လုပ္ရန္ သူ႕ကို လုိအပ္ပါတယ္။

@Entity

@DiscriminatorValue("contractemployee2")

**public** **class** Contract\_Employee2 **extends** Employee2 {

***discriminator column နဲ႔ သူ႕ရဲ႕ value က pass မၿဖစ္ခဲ့ဘူးဆိုရင္ JPA က default ၿဖစ္တယ္လို႔ စဥ္းစားတယ္။ JPA က DTYPE ကို discriminator column name အေနနဲ႔ စဥ္းစားၿပီးေတာ့ entity ရဲ႕ name ကိုေတာ့ discriminator value အေနနဲ႔ စဥ္းစားပါတယ္။***

JPA Inheritence Strategies

1. [Single table strategy](https://www.javatpoint.com/jpa-single-table-strategy)
2. [Joined strategy](https://www.javatpoint.com/jpa-joined-strategy)
3. [Table-per-class strategy](https://www.javatpoint.com/jpa-table-per-class-strategy)

2. Single Table Strategy

single table strategy ဆိုတာက inheritance ရဲ႕ implementation ကို define လုပ္ရန္ ထိေရာက္မွဳအရွိဆံုးနဲ႔ အရိုးရွင္းဆံုးနည္းလမ္း

ၿဖစ္ပါတယ္။ multiple entity classes မ်ားရဲ႕ instance မ်ားအား single table ထဲမွာပဲ attributes မ်ားအေနနဲ႔ stored လုပ္မွာ ၿဖစ္ပါတယ္။

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

Single Table Strategy Example

***EmployeeSingleTable.java***

@Entity

@Table(name="employee\_single\_table")

@Inheritance(strategy=InheritanceType.***SINGLE\_TABLE***)

**public** **class** EmployeeSingleTable **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id

**private** **int** e\_id;

**private** String e\_name;

***ActiveEmployeeSingleTable.java***

@Entity

**public** **class** ActiveEmployeeSingleTable **extends** EmployeeSingleTable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** e\_salary;

**private** **int** e\_experience

**public** ActiveEmployeeSingleTable(**int** e\_id, String e\_name, **int** e\_salary, **int** e\_experience) {

**super**(e\_id,e\_name);

**this**.e\_salary = e\_salary;

**this**.e\_experience = e\_experience;

}

***RetiredEmployeeSingleTable.java***

@Entity

**public** **class** RetiredEmployeeSingleTable **extends** EmployeeSingleTable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** e\_pension;

**public** RetiredEmployeeSingleTable(**int** e\_id, String e\_name,**int** e\_pension) {

**super**(e\_id,e\_name);

**this**.e\_pension = e\_pension;

}

***EmployeePersistSingleTable.java***

**public** **class** EmployeePersistSingleTable {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployeeSingleTable ae1 = **new** ActiveEmployeeSingleTable(101, "Karun", 10000, 5);

ActiveEmployeeSingleTable ae2 = **new** ActiveEmployeeSingleTable(102, "Rishab", 12000, 7);

RetiredEmployeeSingleTable re1 = **new** RetiredEmployeeSingleTable(103, "Ramesh", 5000);

RetiredEmployeeSingleTable re2 = **new** RetiredEmployeeSingleTable(104, "Raj", 4000);

em.persist(ae1);

em.persist(ae2);

em.persist(re1);

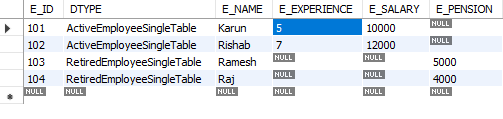
em.persist(re2);

em.getTransaction().commit();

em.close();

emf.close();}}

***employee\_single\_table***



2. Joined strategy

 joined strategy မွာဆုိရင္ entity class တိုင္းအတြက္ separate table  တစ္ခုထုတ္ေပးပါတယ္။ table တစ္ခုခ်င္းစီတုိင္းရဲ႕ attribute သည္ primary key နဲ႔ join ပါတယ္။ duplicacy ရဲ႕ possibility ကို ဖယ္ရွားေပးပါတယ္။

@Inheritance(strategy=InheritanceType.JOINED)

Joined Strategy Example

***EmployeeJoined.java***

@Entity

@Table(name = "employee\_joined")

@Inheritance(strategy = InheritanceType.***JOINED***)

**public** **class** EmployeeJoined **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id

**private** **int** e\_id;

**private** String e\_name;

***ActiveEmployeeJoined.java***

@Entity

public class ActiveEmployeeJoined extends EmployeeJoined {

private static final long serialVersionUID = 1L;

private int e\_salary;

private int e\_experience;

public ActiveEmployeeJoined(int e\_id, String e\_name, int e\_salary, int e\_experience) {

super(e\_id, e\_name);

this.e\_salary = e\_salary;

this.e\_experience = e\_experience;

}

***RetiredEmployeeJoined.java***

@Entity

public class RetiredEmployeeJoined extends EmployeeJoined {

private static final long serialVersionUID = 1L;

private int e\_pension;

public RetiredEmployeeJoined(int e\_id, String e\_name, int e\_pension) {

super(e\_id, e\_name);

this.e\_pension = e\_pension;

}

***EmployeePersistenceJoined.java***

public class EmployeePersistenceJoined {

public static void main(String[] args) {

EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPAThirdProject");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployeeJoined ae1 = new ActiveEmployeeJoined(101, "Karun", 10000, 5);

ActiveEmployeeJoined ae2 = new ActiveEmployeeJoined(102, "Rishab", 12000, 7);

RetiredEmployeeJoined re1 = new RetiredEmployeeJoined(103, "Ramesh", 5000);

RetiredEmployeeJoined re2 = new RetiredEmployeeJoined(104, "Raj", 4000);

em.persist(ae1);

em.persist(ae2);

em.persist(re1);

em.persist(re2);

em.getTransaction().commit();

em.close();

emf.close();}}