JPA - ORM COMPONENTS

 Copyright © tutorialspoint.com

Most contemporary applications use relational database to store data. Recently, many vendors switched to object database to reduce their burden on data maintenance. It means object database or object relational technologies are taking care of storing, retrieving, updating, and maintenance. The core part of this object relational technologies are mapping orm.xml file. As xml does not require compilation, we can easily make changes to multiple data sources with less administration.

Object Relational Mapping

Object Relational Mapping *ORM* briefly tells you about what is ORM and how it works. ORM is a programming ability to covert data from object type to relational type and vice versa.

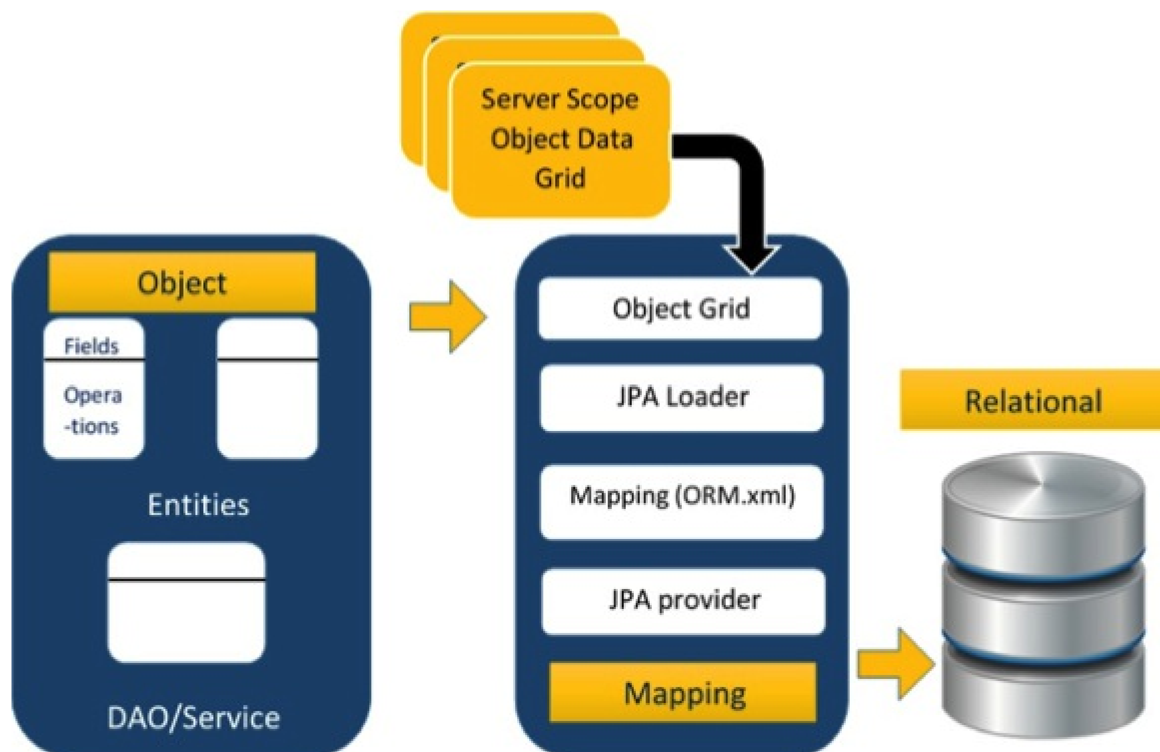
The main feature of ORM is mapping or binding an object to its data in the database. While mapping we have to consider the data, type of data and its relations with its self-entity or entity in any other table.

Advanced Features

- **Idiomatic persistence** : It enables you to write the persistence classes using object oriented classes.
- **High Performance** : It has many fetching techniques and hopeful locking techniques.
- **Reliable** : It is highly stable and eminent. Used by many industrial programmers.

ORM Architecture

Here follow the ORM architecture.



The above architecture explains how object data is stored into relational database in three phases.

Phase1

The first phase, named as the **Object data** phase contains POJO classes, service interfaces and classes. It is the main business component layer, which has business logic operations and attributes.

For example let us take an employee database as schema-

- Employee POJO class contain attributes such as ID, name, salary, and designation. And methods like setter and getter methods of those attributes.
- Employee DAO/Service classes contains service methods such as create employee, find employee, and delete employee.

Phase 2

The second phase named as **mapping** or **persistence** phase which contains JPA provider, mapping file $ORM.xml$, JPA Loader, and Object Grid.

- **JPA Provider** : The vendor product which contains JPA flavor $javax.persistence$. For example Eclipselink, Toplink, Hibernate, etc.
- **Mapping file** : The mapping file $ORM.xml$ contains mapping configuration between the data in a POJO class and data in a relational database.
- **JPA Loader** : The JPA loader works like cache memory, which can load the relational grid data. It works like a copy of database to interact with service classes for POJO data $AttributesofPOJOclass$.
- **Object Grid** : The Object grid is a temporary location which can store the copy of relational data, i.e. like a cache memory. All queries against the database is first effected on the data in the object grid. Only after it is committed, it effects the main database.

Phase 3

The third phase is the Relational data phase. It contains the relational data which is logically connected to the business component. As discussed above, only when the business component commit the data, it is stored into the database physically. Until then the modified data is stored in a cache memory as a grid format. Same is the process for obtaining data.

The mechanism of the programmatic interaction of above three phases is called as object relational mapping.

Mapping.xml

The mapping.xml file is to instruct the JPA vendor for mapping the Entity classes with database tables.

Let us take an example of Employee entity which contains four attributes. The POJO class of Employee entity named **Employee.java** is as follows:

```
public class Employee {

   private int eid;
   private String ename;
   private double salary;
   private String deg;

   public Employee(int eid, String ename, double salary, String deg) {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( ) {
      super();
   }

   public int getEid( ) {
      return eid;
```

```java
    }

    public void setEid(int eid) {
        this.eid = eid;
    }

    public String getEname( ) {
        return ename;
    }

    public void setEname(String ename) {
        this.ename = ename;
    }

    public double getSalary( ) {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public String getDeg( ) {
        return deg;
    }

    public void setDeg(String deg) {
        this.deg = deg;
    }
}
```

The above code is the Employee entity POJO class. It contain four attributes eid, ename, salary, and deg. Consider these attributes are the table fields in the database and eid is the primary key of this table. Now we have to design hibernate mapping file for it. The mapping file named **mapping.xml** is as follows:

```xml
<? xml version="1.0" encoding="UTF-8" ?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">

    <description> XML Mapping file</description>

    <entity >
        <table name="EMPLOYEETABLE"/>
        <attributes>

            <id name="eid">
                <generated-value strategy="TABLE"/>
            </id>

            <basic name="ename">
                <column name="EMP_NAME" length="100"/>
            </basic>

            <basic name="salary">
            </basic>

            <basic name="deg">
            </basic>

        </attributes>
    </entity>

</entity-mappings>
```

The above script for mapping the entity class with database table. In this file

- **<entity-mappings>** : tag defines the schema definition to allow entity tags into xml file.
- **<description>** : tag defines description about application.
- **<entity>** : tag defines the entity class which you want to convert into table in a database. Attribute class defines the POJO entity class name.
- **<table>** : tag defines the table name. If you want to keep class name as table name then this tag is not necessary.
- **<attributes>** : tag defines the attributes *fieldsinatable*.
- **<id>** : tag defines the primary key of the table. The **<generated-value>** tag defines how to assign the primary key value such as Automatic, Manual, or taken from Sequence.
- **<basic>** : tag is used for defining remaining attributes for table.
- **<column-name>** : tag is used to define user defined table field name.

Annotations

Generally Xml files are used to configure specific component, or mapping two different specifications of components. In our case, we have to maintain xml separately in a framework. That means while writing a mapping xml file we need to compare the POJO class attributes with entity tags in mapping.xml file.

Here is the solution: In the class definition, we can write the configuration part using annotations. The annotations are used for classes, properties, and methods. Annotations starts with '@' symbol. Annotations are declared before the class, property or method is declared. All annotations of JPA are defined in javax.persistence package.

Here follows the list of annotations used in our examples

Annotation	Description
@Entity	This annotation specifies to declare the class as entity or a table.
@Table	This annotation specifies to declare table name.
@Basic	This annotation specifies non constraint fields explicitly.
@Embedded	This annotation specifies the properties of class or an entity whose value instance of an embeddable class.
@Id	This annotation specifies the property, use for identity *primarykeyofatable* of the class.
@GeneratedValue	This annotation specifies, how the identity attribute can be initialized such as Automatic, manual, or value taken from sequence table.
@Transient	This annotation specifies the property which in not persistent i.e. the value is never stored into database.
@Column	This annotation is used to specify column or attribute for persistence property.
@SequenceGenerator	This annotation is used to define the value for the property which is specified in @GeneratedValue annotation. It creates a sequence.
@TableGenerator	This annotation is used to specify the value generator for property specified in @GeneratedValue annotation. It creates a table for value generation.
@AccessType	This type of annotation is used to set the access type. If you set

	@AccessType*FIELD* then Field wise access will occur. If you set @AccessType*PROPERTY* then Property wise assess will occur.
@JoinColumn	This annotation is used to specify an entity association or entity collection. This is used in many- to-one and one-to-many associations.
@UniqueConstraint	This annotation is used to specify the field, unique constraint for primary or secondary table.
@ColumnResult	This annotation references the name of a column in the SQL query using select clause.
@ManyToMany	This annotation is used to define a many-to-many relationship between the join Tables.
@ManyToOne	This annotation is used to define a many-to-one relationship between the join Tables.
@OneToMany	This annotation is used to define a one-to-many relationship between the join Tables.
@OneToOne	This annotation is used to define a one-to-one relationship between the join Tables.
@NamedQueries	This annotation is used for specifying list of named queries.
@NamedQuery	This annotation is used for specifying a Query using static name.

Java Bean Standard

Java class, encapsulates the instance values and behaviors into a single unit callled object. Java Bean is a temporary storage and reusable component or an object. It is a serializable class which has default constructor and getter & setter methods to initialize the instance attributes individually.

Bean Conventions

- Bean contains the default constructor or a file that contains serialized instance. Therefore, a bean can instantiate the bean.
- The properties of a bean can be segregated into Boolean properties and non-Boolean properties.
- Non-Boolean property contains **getter** and **setter** methods.
- Boolean property contain **setter** and **is** method.
- **Getter** method of any property should start with small lettered ‘get’ *javamethodconvention* and continued with a field name that starts with capital letter. E.g. the field name is ‘salary’ therefore the getter method of this field is ‘getSalary ’.
- **Setter** method of any property should start with small lettered ‘set’ *javamethodconvention*, continued with a field name that starts with capital letter and the argument value to set to field. E.g. the field name is ‘salary’ therefore the setter method of this field is ‘setSalary *doublesal*’.
- For Boolean property, is method to check if it is true or false. E.g. the Boolean property ‘empty’, the is method of this field is ‘isEmpty ’.

Loading [MathJax]/jax/output/HTML-CSS/jax.js