ORM,JPA with Hibernate

Pre requisites

Tools

Core Java

--JDBC

PostgresSQL

Maven

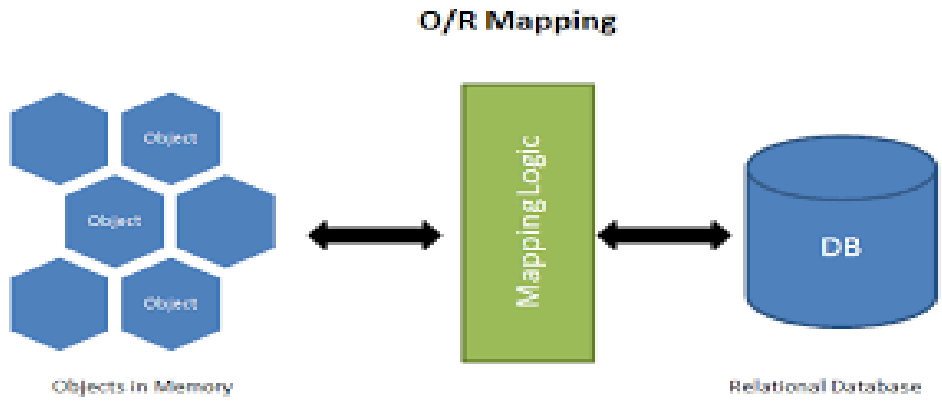
Eclipse

# ORM

An ORM provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries. It standardizes interfaces reducing boilerplate and speeding development time.

Object-oriented programming includes many states and codes in a format that is complex to understand and interpret. ORMs translate this data and create a structured map to help developers understand the underlying database structure. The mapping explains how objects are related to different tables.

ORMs use this information to convert data between tables and generate the SQL code for a relational database to insert, update, create and delete data in response to changes the application makes to the data object. Once written, the ORM mapping will manage the application's data needs and you will not need to write any more low-level code.




ORM

|

static relationship

class <----------> table

attributes<--------->columns

dynamic relation ship

object<---------->row

FAQ

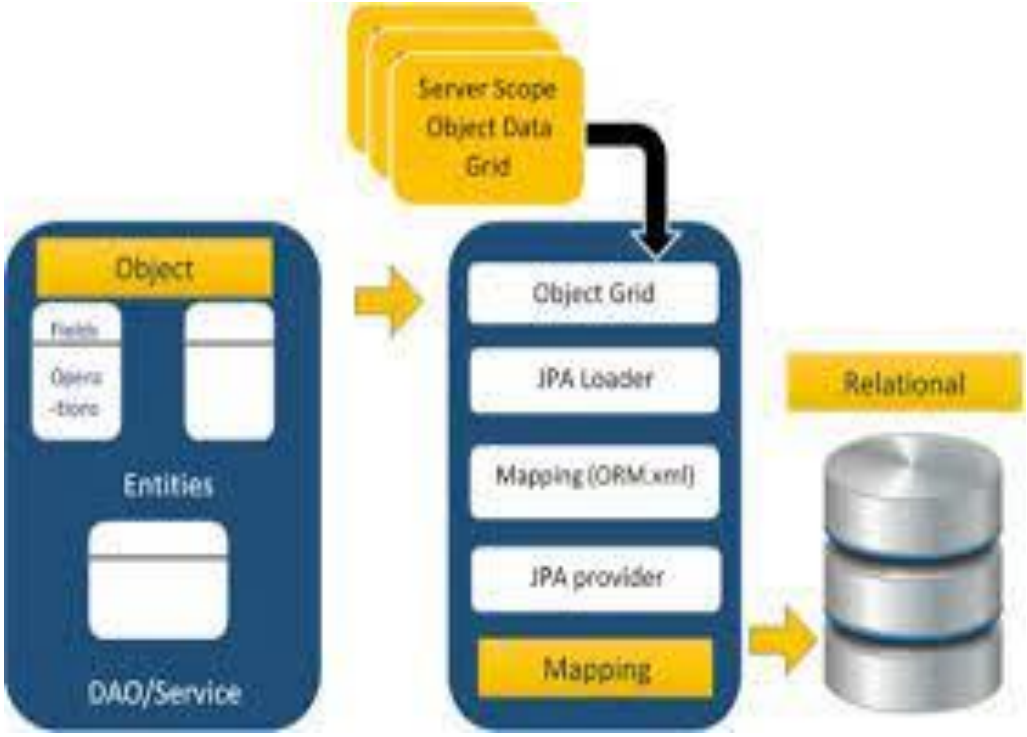
1.Why JPA is preferred over JDBC

2.What are other tools besides Hibernate

3.Any other specification  besides JPA for ORM

4.Drawbacks with JPA

# JPA




A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is considered as a standard approach for Object Relational Mapping.

JPA can be seen as a bridge between object-oriented domain models and relational database systems. Being a specification, JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence.

A Hibernate is a Java framework which is used to store the Java objects in the relational database system. It is an open-source, lightweight, ORM (Object Relational Mapping) tool.

Hibernate is an implementation of JPA. So, it follows the common standards provided by the JPA

Look at the code below

```
package jpaapp.sampleapp;


import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;



@Entity
@Table(name=”StudentTable”)
public class Student {

	@Id
	 private int id;



	 @Column(name=”StudentFirstName”)
	 private String firstName;



	 @Column(name=”StudentLastName”)

	 private String lastName;
```

```
private int marks;

 public Student() {

 }

 public Student(int id, String firstName, String lastName, int marks) {
         super();
         this.id = id;
         this.firstName = firstName;
         this.lastName = lastName;
         this.marks = marks;
 }

 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 int getMarks() {
		return marks;
	}

	public void setMarks(int marks) {
		this.marks = marks;
	}
}
```

The code is a POJO class that is decorated(configured) with annotations

JPA 1.0: ORM 3.2+

JPA 2.0: ORM 3.5+

JPA 2.1: ORM 4.3+

```
<dependency>
 <groupId>org.eclipse.persistence</groupId>
<artifactId>javax.persistence</artifactId>
<version>2.0.0</version>
</dependency>
```

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.5.6-Final</version>
</dependency>



<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-slf4j-impl</artifactId>
   <version>2.1</version>

</dependency>



<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>42.2.23</version>
</dependency>
```

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"

        version="2.0">
        <persistence-unit name="BookPU">
```

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>

                        <property name="hibernate.connection.url"
value="jdbc:postgresql://localhost:5432/sample" />

                        <property name="hibernate.connection.driver_class"
value="org.postgresql.Driver" />

                        <property name="hibernate.connection.username"
value="postgres" />

                        <property name="hibernate.connection.password" value="123"
/>

                        <property name="hibernate.archive.autodetection"
value="class" />

                        <property name="hibernate.show_sql" value="true" />

                        <property name="hibernate.format_sql" value="true" />

                        <property name="hibernate.hbm2ddl.auto" value="update" />

                </properties>

        </persistence-unit>



</persistence>
```

values in hbm2ddl.auto

validate: validate the schema, makes no changes to the database.

update: update the schema.

create: creates the schema, destroying previous data.

create-drop: drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.

none: does nothing with the schema, makes no changes to the database

Persistence Context

An EntityManager instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Entity Lifecycle stages in JPA

Transient

Managed

Removed

Detached

## Transient

The lifecycle state of a newly instantiated entity object is called transient. The entity hasn't been persisted yet, so it doesn't represent any database record.

Your persistence context doesn't know about your newly instantiate object. Because of that, it doesn't automatically perform an SQL INSERT statement or tracks any changes. As long as your entity object is in the lifecycle state transient, you can think of it as a basic Java object without any connection to the database and any JPA-specific functionality.

## Managed

All entity objects attached to the current persistence context are in the lifecycle state managed. That means that your persistence provider, e.g. Hibernate, will detect any changes on the objects and generate the required SQL INSERT or UPDATE statements when it flushes the persistence context.

There are different ways to get an entity to the lifecycle state managed

1.Calling EntityManager's Persist Method

2.Loading and Entity from Database using find method

3.Merge a detached entity using merge method

## Detached

An entity that was previously managed but is no longer attached to the current persistence context is in the lifecycle state detached.

An entity gets detached when you close the persistence context. That typically happens after a request got processed. Then the database transaction gets committed, the persistence context gets closed, and the entity object gets returned to the caller. The caller then retrieves an entity object in the lifecycle state detached.

You can also programmatically detach an entity by calling the detach method on the EntityManager.

You can reattach an entity by calling the update method on your Hibernate Session or the merge method on the EntityManager.

Removed

When you call the remove method on your EntityManager, the mapped database record doesn't get removed immediately. The entity object only changes its lifecycle state to removed.

During the next flush operation, Hibernate will generate an SQL DELETE statement to remove the record from the database table.

life cycle call back functions

```
@PrePersist
public void newStudentCreated()
{
          System.out.println("********************************");
          System.out.println("book data about to be saved");
}

@PostPersist
public void newStudentSaved()
{
          System.out.println("********************************");
          System.out.println("book data saved");
}

@PreRemove
public void logUserRemovalAttempt() {
```

```
        System.out.println("book to be removed");

}

@PostRemove
public void logUserRemoval() {
        System.out.println("book removed");

}

@PreUpdate
public void logUserUpdateAttempt() {
        System.out.println("book about to be updated");

}

@PostUpdate
public void logUserUpdate() {
        System.out.println("book updated");

}


@PostLoad
      public void loadlog(Book i)
      {
            System.out.println("entity loaded");
            System.out.println(i.getBookName());
            System.out.println(i.getBookId());
      }
```

JPA API

@Entity

@Table

@Column

@Id

@Transient

Java primitive types

Java primitive wrapper types

java.lang.String

java.util.Date (the temporal type should be DATE)

@Temporal(TemporalType.DATE)

private java.util.Date creationDate;

/*

this annotation must be specified for persistent fields or properties of type java.util.Date and java.util.Calendar. It may only be specified for fields or properties of these types.

The Temporal annotation may be used in conjunction with the Basic annotation, the Id annotation, or the ElementCollection annotation (when the element collection value is of such a temporal type.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data, you might want to describe the expected precision in database. Temporal data can have DATE, TIME, or TIMESTAMP precision (i.e., the actual date, only the time, or both). Use the @Temporal annotation to fine tune that.

The temporal data is the data related to time. For example, in a content management system, the creation-date and last-updated date of an article are temporal data. In some cases, temporal data needs precision and you want to store precise date/time or both (TIMESTAMP) in database table.

The temporal precision is not specified in core Java APIs. @Temporal is a JPA annotation that converts back and forth between timestamp and java.util.Date. It also converts time-stamp into time. For example, in the snippet below, @Temporal(TemporalType.DATE) drops the time value and only preserves the date.

*/

java.sql.Date

java.math.BigDecimal

java.math.BigInteger

Composite Primary Keys

A composite primary key, also called a composite key, is a combination of two or more columns to form a primary key for a table.

In JPA, we have two options to define the composite keys: the @IdClass and @EmbeddedId annotations.

In order to define the composite primary keys, we should follow some rules:

The composite primary key class must be public.

It must have a no-arg constructor.

It must define the equals() and hashCode() methods.

It must be Serializable.

GeneratedValue

AUTO, IDENTITY, SEQUENCE, TABLE.

AUTO

If we're using the default generation type, the persistence provider will determine values based on the type of the primary key attribute.

@Entity public class Student { @Id @GeneratedValue private long studentId; // ... }

IDENTITY Generation

This type of generation relies on the IdentityGenerator which expects values generated by an identity column in the database, meaning they are auto-incremented.

*SEQUENCE* **Generation**

To use a sequence-based id, Hibernate provides the *SequenceStyleGenerator* class.

This generator uses sequences if they're supported by our database, and switches to table generation if they aren't.

**TABLE Generation**

The *TableGenerator* uses an underlying database table that holds segments of identifier generation values.

Let's customize the table name using the *@TableGenerator* annotation:

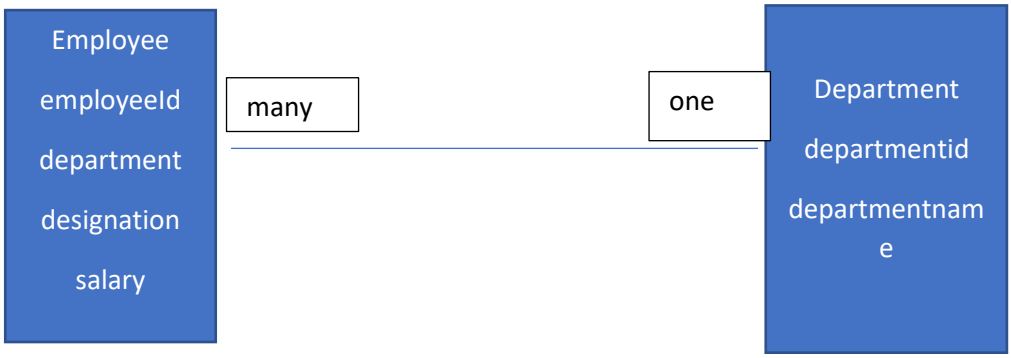
Relationship between Classes

OneToOne

OneToMany

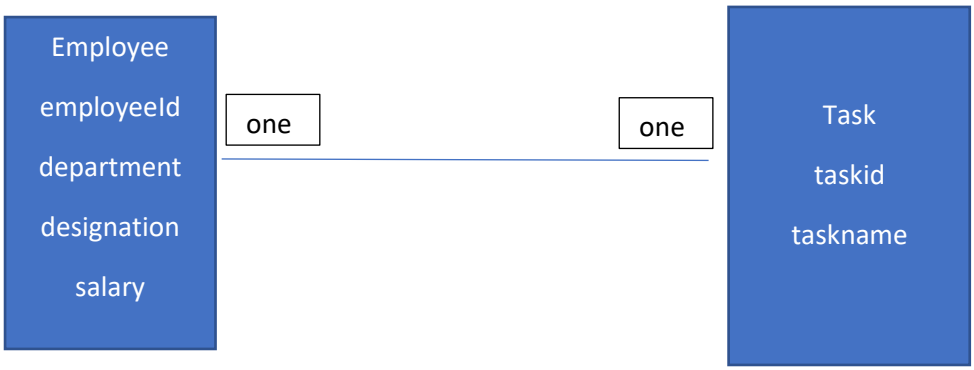
ManyToOne

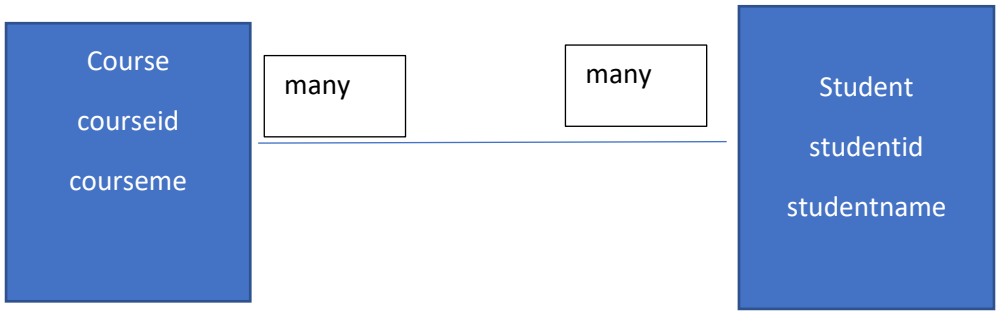
ManyToMany



ManyToOne

OneToMany




Unidirectional

Relationship will be applied on one entity and discovery of entity can be done from one side only

The unidirectional relationship defines relationship only on 1 of 2 entities.and you can navigate only in that direction

Bidirectional relationship

bidirectional mapping models the relationship for both the entities and navigation can be done in both the direction

Bidirectional relationship has an Owning side and inverse side

Cardinality

OneToOne

One To Many

Many To One

Many To Many

Join Column

JPQL

==========

Introduction to web technologies

# JPQL

Executing queries of Entities

eg:-

tradition sql query for a table employee

select * from employee

JPQL over entity

Select e from Employee e

JPQL queries are passed into entitymanager createQuery or createNamedQuery function

Named Parameters

```
String query="select p from Person p where p.personName = :personname ";
```

```
Query jpaquery=emf.createQuery(query);
jpaquery.setParameter("personname","Peter");

List<Person> personList=jpaquery.getResultList();
```

Positional Parameters

```
String query="select p from Person p where p.personName = ?1 ";
```

reference document

https://www.baeldung.com/jpa-query-parameters

Joins in JPA Query

https://www.baeldung.com/jpa-query-parameters

Basics of Web Technology using Servlet and JSP

Types of Web Pages

Static vs Dynamic

#Static Pages do not have interactivity

#Can be built using HTML

Dynamic pages

#Dynamic pages can respond to user action .can contain different responsive elements like

form ,controls ,buttons ,videos

Client Side Script Vs Server Side Script

Client Side Script are programming elements or scripting elements that are written ,embedded and executed on client side

#Validation of form controls

#Error messages

#Information on the client side

Client Side Scripting languages

JavaScript

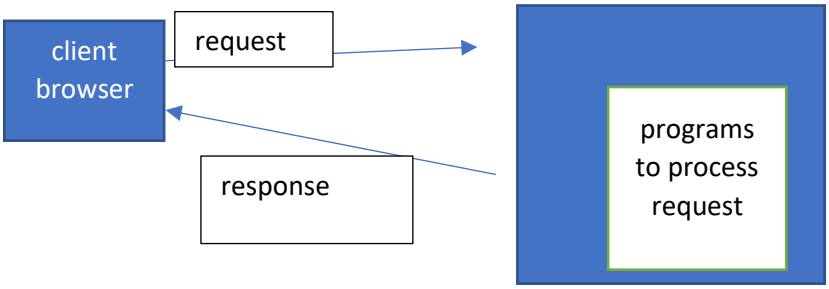
JScript

C#Script

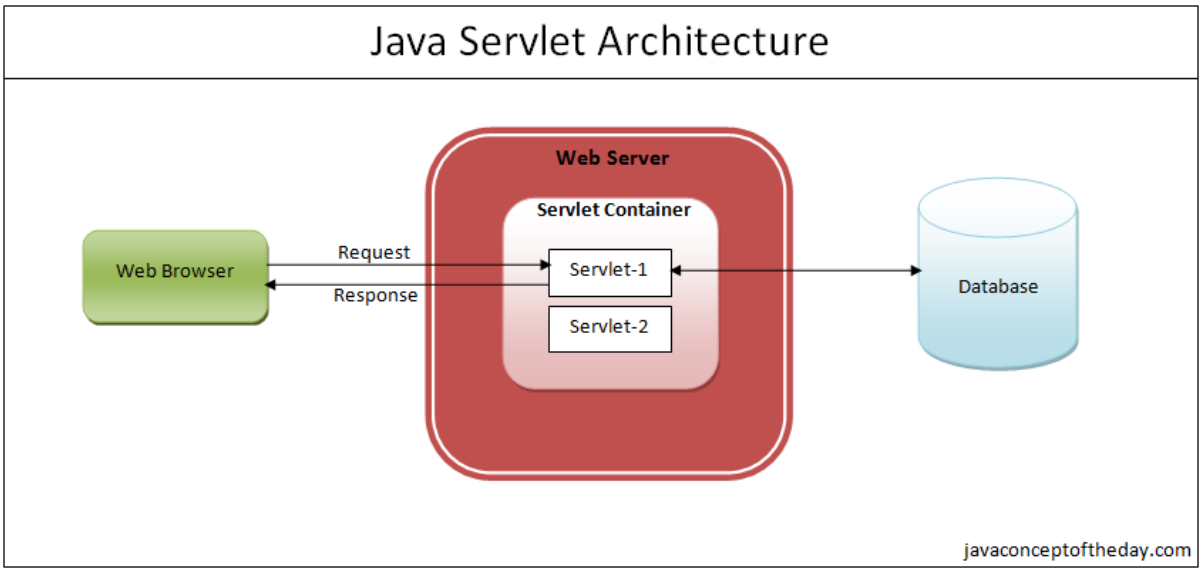
VBScript

Server Side Scripting language

Server Side Scripting languages are used to build programs that are executed from server side



# Servlet




#Servlet is a java program executed from server side

#Servlet resides inside web container

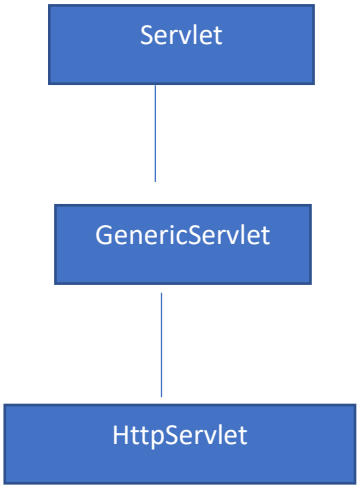
#Servlet has life cycle methods init ,service and destroy

#Each time a request is received service method is executed

#Service has paramters of type ServletRequest and ServletResponse

#ServletRequest handles incoming request and ServletResponse is used to generate Response

#A Web Container invokes servlet methods .We can override Servlet methods but cannot invoke them




HttpServlet is subclass of GenericServlet .HttpServlet contains methods that correspond to http protocol methods

Methods in HTTP protocol

GET

POST

PUT

DELTE

TRACE

ex:-

GET method attaches form values and control parameters to url in key value pair format