SOEN 387
WEB-BASED ENTERPRISE APPLICATIONS DESIGN

TUTORIAL – 12

Java Persistence API

Agenda

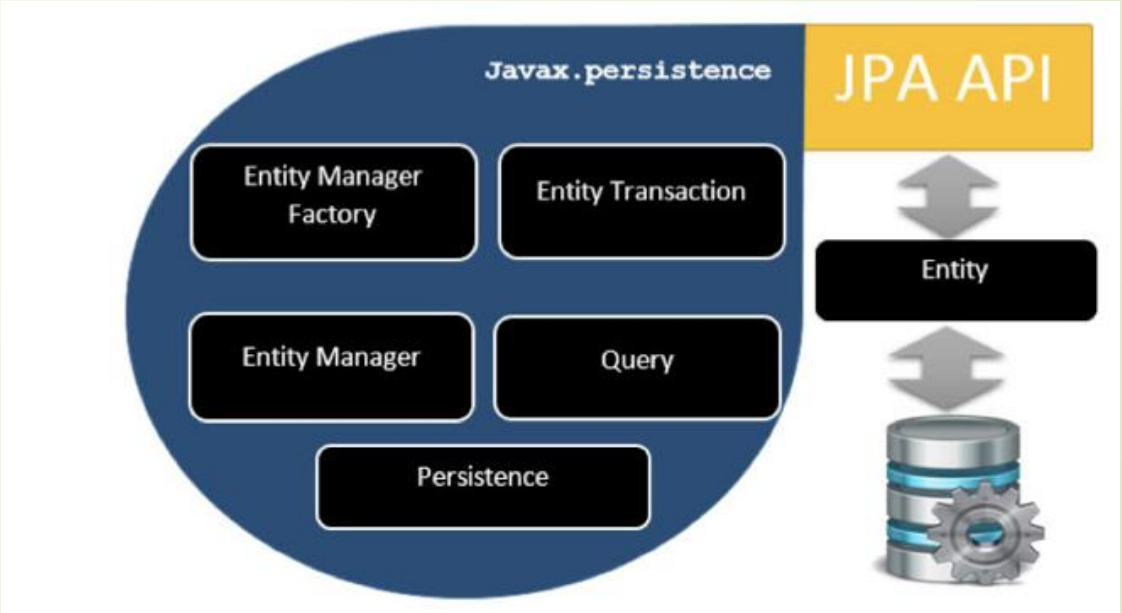
- Introduction & Architecture
- Example: Convert data object to relational type
- Example: Simple JAVA application in NetBeans using ObjectDB and JPA
- JPQL (Java Persistence Query language)
- In class exercise: User Authentication
- In class exercise: Implementing the Repository pattern with JPA

Introduction & Architecture

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database

It forms a bridge between object models (Java program) and relational models (database program).

Java Persistence API is a source to store business entities as relational entities. It shows how to define a PLAIN OLD JAVA OBJECT (POJO) as an entity and how to manage entities with relations.



Example: Convert data object to relational type

ORM is a programming ability to convert data from object type to relational type and vice versa.

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.

```
public class Employee {
   private int eid;
   private String ename;

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

   public Employee( ) {
      super();
   }

   public int getEid( ) {
      return eid;
   }

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

```
   public String getEname( ) {
      return ename;
   }

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

}
```

Mapping.xml

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

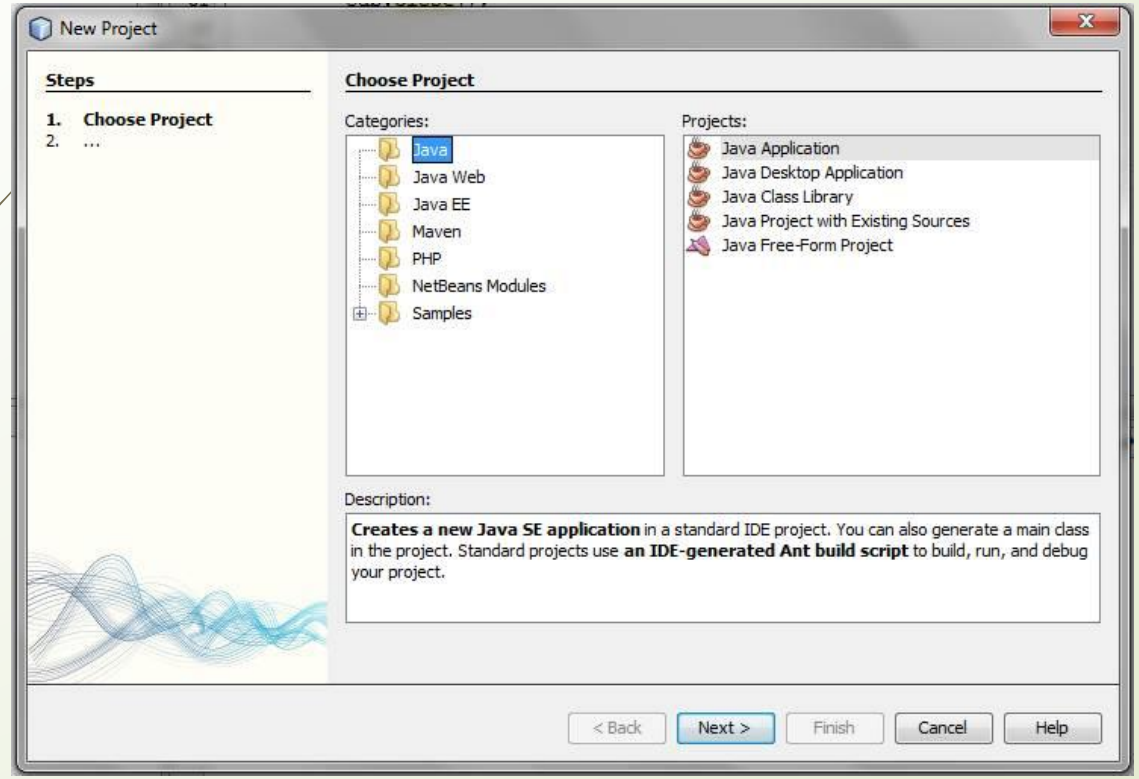
```
<? xml version="1.0" encoding="UTF-8" ?>

<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 class="Employee">
      <table name="EMPLOYEETABLE"/>
      <attributes>
         <id name="eid">
            <generated-value strategy="TABLE"/>
         </id>

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

</entity-mappings>
```

while writing a mapping xml file we need to compare the POJO class attributes with entity tags in mapping.xml file

Example: Simple JAVA application in NetBeans using ObjectDB and JPA

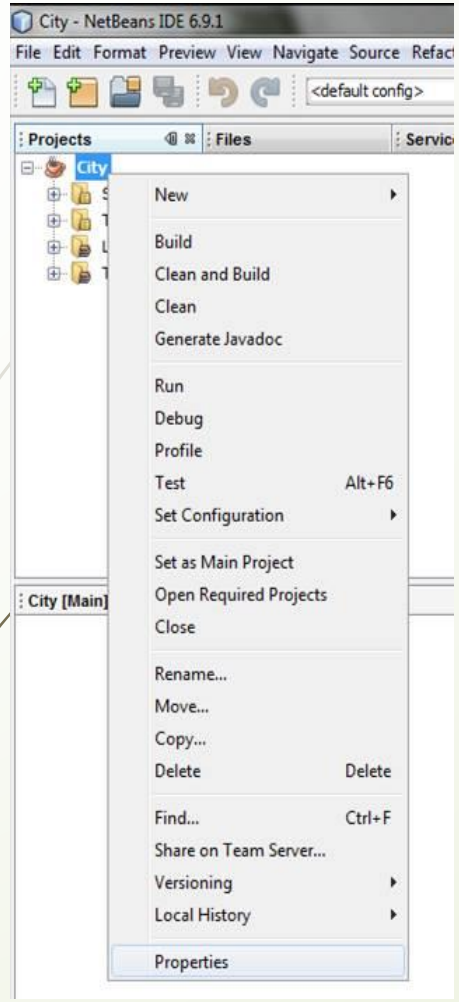
Step 1: Create an ObjectDB Enabled Java SE Project

From the menu, select File > New to get the [New Project] dialog box and finish the project.



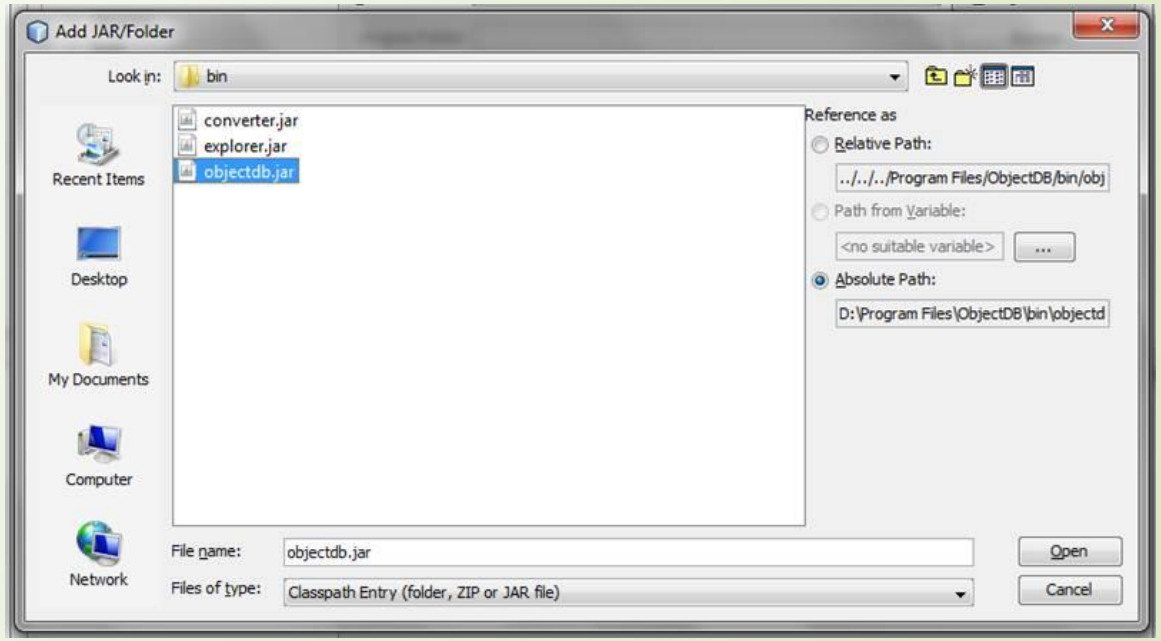
Download the ObjectDB installation files from the download page.

Right click on the project to get the popup menu and select the Properties to get the [Project Properties]window.



Click on Libraries under categories of the Project Properties Window. No libraries are listed at the moment under Compile tab.

Click on Add Jar/Folder and locate the objectdb.jar file inside the bin folder of ObjectDB extracted directory.



This will add the objectdb.jar file to the libraries and now the project is JPA/ObjectDB enabled.

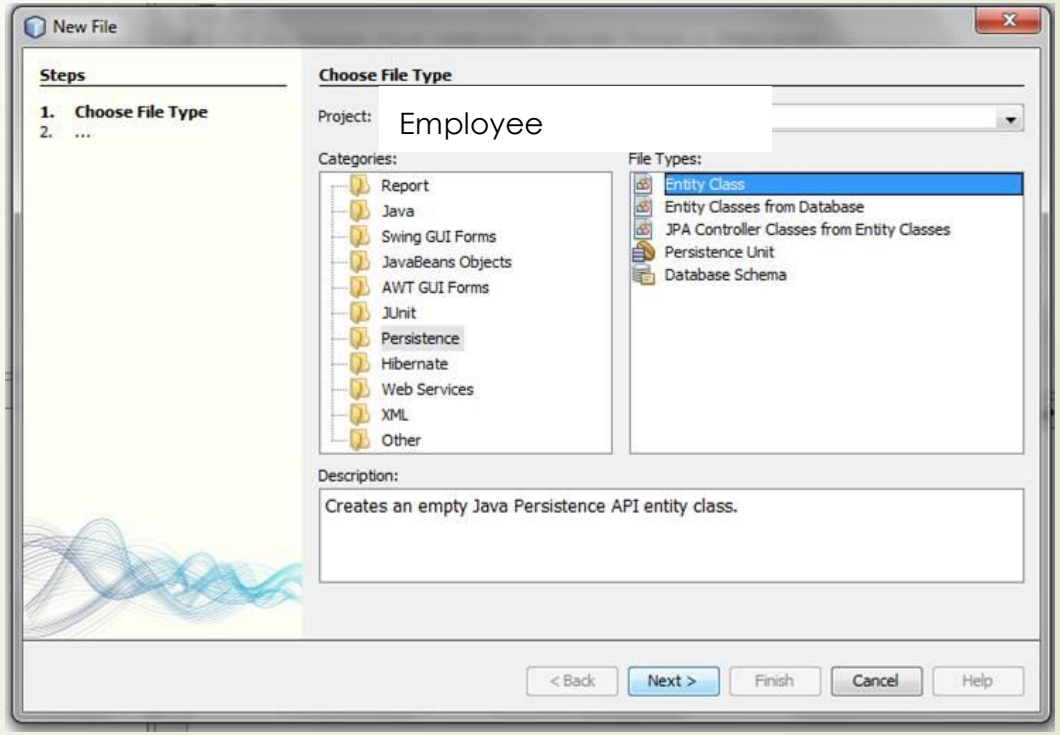
Step 2: Define a JPA Entity Classes

First we will create a new package named 'bean' to include entity classes.

Right click on Employee node or Source Packages node to get the popup menu and select New. The recent selections are listed and find the Java Package.

In the New Java Package window, enter bean as the package name and make sure Source Packages is selected as the location. The click Finish to create the package.

To create a new entity class right click on bean and select New > Entity Class. In the New Entity Class window, type Employee as the Class Name. Then click Finish to generate State Entity Class.



Employee Entity Java:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table
public class Employee {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)

   private int eid;
   private String ename;

  // Construtor details

public Employee( ) {
      super();
    }

  // Setters and getters

  // toString method
}
```

In the above code, we have used @Entity annotation to make this POJO class as entity.

Before going to next module we need to create database for relational entity, which will register the database in persistence.xml file. Open MySQL workbench and type query as follows:

```
create database jpadb
use jpadb
```

Step 3: Define Persistence.xml

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

<persistence version="2.0" 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">

   <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>

      <properties>
         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpadb"/>
         <property name="javax.persistence.jdbc.user" value="root"/>
         <property name="javax.persistence.jdbc.password" value="root"/>
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
         <property name="eclipselink.logging.level" value="FINE"/>
         <property name="eclipselink.ddl-generation" value="create-tables"/>
      </properties>

   </persistence-unit>
</persistence>
```

tag is defined with specific name for JPA persistence

tag defines entity class with package name

tag defines all the properties

<property> tag defines each property such as database registration, URL specification, username, and password

Step 4: Persistence Operations

Persistence operations are used against database and they are load and store operations.

Create a service package under src and all the service classes named as *CreateEmloyee.java, UpdateEmployee.java, FindEmployee.java, and DeleteEmployee.java* are added under this.

CreateEmployee.java

```
public class CreateEmployee {

        public static void main( String[ ] args ) {

          EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Eclipselink_JPA" );

          EntityManager entitymanager = emfactory.createEntityManager( );
          entitymanager.getTransaction( ).begin( );

         // Object creation and assignment

          entitymanager.persist( employee );
          entitymanager.getTransaction( ).commit( );

          entitymanager.close( );
          emfactory.close( );
        }
      }
```

createEntityManagerFactory () creates a persistence unit by providing the same unique name which we provide for persistence-unit in persistent.xml file

For result, open the MySQL workbench and type the following queries.

```
use jpadb
select * from employee
```

The affected database table named employee will be shown in a tabular format as follows:

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager

Similarly you can write JPA for UpdateEmployee.java, FindEmployee.java, and DeleteEmployee.java

Java Persistence Query language

Used to create queries against entities to store in a relational database

JPQL works with Java classes and instances.

```
public class ScalarandAggregateFunctions {
    public static void main( String[ ] args ) {

        EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
        EntityManager entitymanager = emfactory.createEntityManager();

        //Scalar function
        Query query = entitymanager.
        createQuery("Select UPPER(e.ename) from Employee e");
        List<String> list = query.getResultList();

        for(String e:list) {
           System.out.println("Employee NAME :"+e);
        }

        //Aggregate function
        Query query1 = entitymanager.createQuery("Select MAX(e.salary) from Employee e");
        Double result = (Double) query1.getSingleResult();
        System.out.println("Max Employee Salary :" + result);
    }
}
```

It also supports Between, And, Like, Ordering are supported as well.

In class exercise: User Authentication

AIM: To implement user authentication on a login page using JPAs.

- Create a Dynamic Web Project in using JPA for adding the User Entity Class, Login, and Welcome Pages.
- The User class has all the getters/setters for the Users table
- declare the class an Entity by using the @Entity annotation,
- use the @Id annotation to generate an ID for the Id column of the table
- create the persistence.xml file
- create a class LogOnTest to test JPA by adding a record to our table
- create a backing bean named LogonBean.java to authenticate the user.
- create a login.jsp page to enter user information from front end.

Reference

In class exercise: Implementing the Repository pattern with JPA

- Create a repository interface named as BookRepository
- It should have below methods:
  - save a new or changed entity
  - delete an entity,
  - find an entity by its primary key and
  - find an entity by its title.

```
package org.thoughts.on.java.repository;

    import org.thoughts.on.java.model.Book;

    public interface BookRepository {

        Book getBookById(Long id);

        Book getBookByTitle(String title);

        Book saveBook(Book b);

        void deleteBook(Book b);
    }
```

- Now implement the interface with JPA

```
package org.thoughts.on.java.repository;

import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import org.thoughts.on.java.model.Book;

public class BookRepositoryImpl implements BookRepository {

    private EntityManager em;

    public BookRepositoryImpl(EntityManager em) {
        this.em = em;
    }

    @Override
    public Book getBookById(Long id) {
        // Getter
    }
```

```
    @Override
    public Book getBookByTitle(String title) {
        TypedQuery<Book> q = em.createQuery("SELECT b
FROM Book b WHERE b.title = :title", Book.class);
        q.setParameter("title", title);
        return q.getSingleResult();
    }
    @Override
    public Book saveBook(Book b) {
      // Implementation
    }

    @Override
    public void deleteBook(Book b) {
        if (em.contains(b)) {
            em.remove(b);
        } else {
            em.merge(b);
        }
    }
}
```