

Persistence logic

==============

The logic which is written to perform persistence operation is called "Persistence logic".

operations are CRUD/SCUD/CURD.

To write persistence logic in java we have technology and framework

a. technology => JDBC

b. framework => ORM tools like hibernate, jpa, springorm, springdatajpa(hotcake)

When we already have JDBC as persitence logic, what is the need to for ORM?

limitations of JDBC

a. If we use JDBC to develop persistence logic, we need to write sql queries by following the syntax of "Database".

DBQueries are specific to Database, this makes JDBC not portable across multiple databases.

JAVA => WORA

JAVA + JDBC == > WORA (not supported)

b. JDBC technology if we use and write a code, there would be a boiler plate code in our application.

Boiler plate code => A code which would repeat in multiple parts of the project with no change/small change is called boiler plate code.

CRUD

=====

1. Load and register the driver (automatic from JDBC4.X)

2. Establish the connection

3. Create PrepareStatement

4. Execute the Query

5. Process the ResultSet

6. Handle the Exception

7. Closing the Resource

Step1,2,3,6,7 boiler plat code because it is a common logic.

C. JDBC technology throws only one Exception called "SQLException', but it is a CheckedException which means we should have handling logic otherwise code would not compile.

a. try{

}catch(SQLException e){

}

b. public static void main(String ... args) throws SQLException{

}

d. JDBC technology has only Exception class called "SQLException", so we don't have detailed hierarch of Exceptions related to different problems.

e. JDBC ResultSet object is not serializable, so we can't send it over the network, we need to use Bean/POJO to send the data over the network by writing our own logic.

f. While closing the jdbc connection object, we need to analyze the code allot otherwise it would result in "NullPointerException”

eg: Connection con = DriverManager.getConnection(url,user,password)

if(con != null){ ..... }

closing the connection object should take place in "finally" block only.

To make the usage of AutoCloseable, we need to know the syntax of "try with Resource".

g. Java ==== > OOP's based language

Assume we need to send Student object to database, can we write a logic of Database query at Object level if we use JDBC?

No, Not possible becoz DBqueries always expectes the value, but not the object directly.

h. JDBC doesn’t have good support of Transaction Management

a. local

b. global (no support in JDBC)

i. JDBC supports only positional parameters, it is difficult for the user to inject the values

It does not support named parameters.

String sqlInsertQuery = "insert into student(`name', 'email', city', country') values(?,?,?,?)";

String sqlInsertQuery = "insert into student('name', email', city', country') values(:name,:email,:city,:country)";

j. To use JDBC, Strong knowledge of SQL is required.

h. JDBC does not supports versioning, timestamp as inbuilt features

versioning :: keeping track of how many times record got modified.

timestamp :: keep track of when record was inserted and when lastly it was modified.

k.While developing persitence logic using JDBC, we can't enjoy oops features like

a. inheritance

b. polymorphism

c. composition

becoz jdbc does not allow objects as input values in sqlqueries.

Solution: To all the problems mentioned above we develpo persistence logic using "ORM".

ORM tools are hiberante,eclipselink,ibatis,jpa, .....

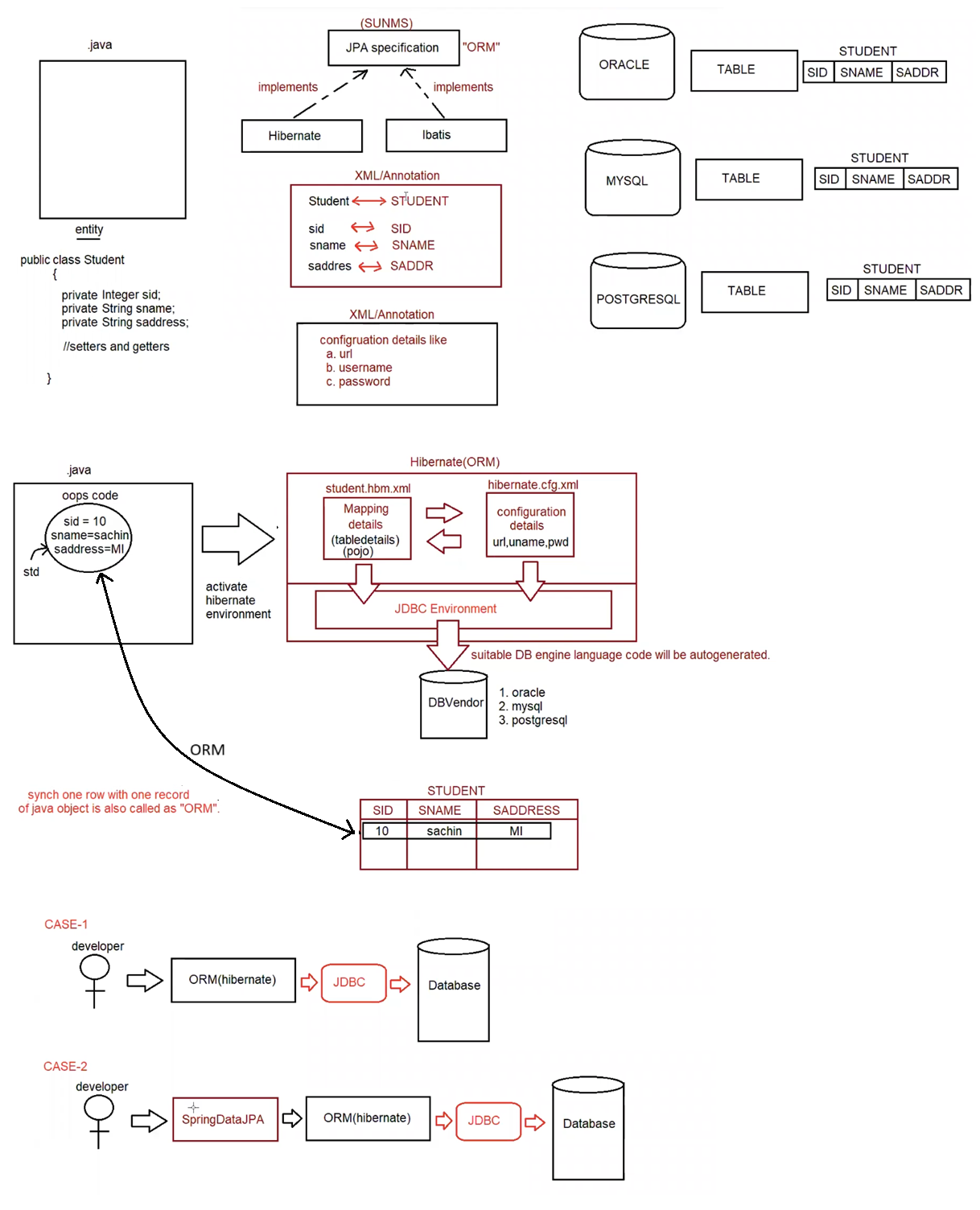
ORM -> Object Relation Mapping

To overcome the limitations of JDBC, we need to opt for ORM.

1. JDBC code is not portable.

2. JDBC we can't deal with Object injection to the database as the query expects values.

ORM => It stands for Object Relational Mapping

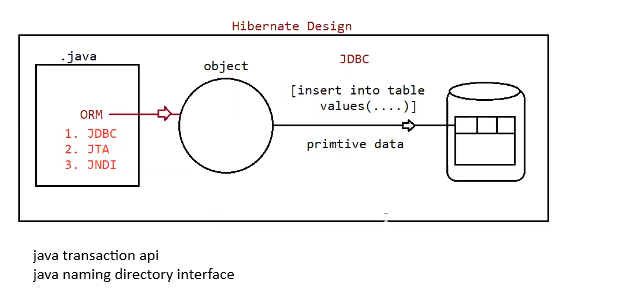


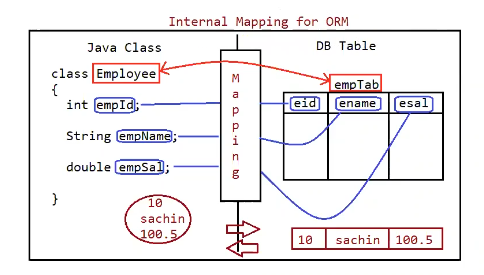
what is ORM?

it stands for Object relational mapping, where the programmer would map the table details with java object details through xml/annotation.

JPA -> Java Persistence API (set of rules and guidelines to implement ORM)

Hibernate is a tool/framework implemented for JPA specification given by SUNMS.





To write one application we need to use 4 files as shown below

a. Model class

b. Mapping code(xml/annotation)

c. Configuration file(xml)

d. Test class

Model class

It represent Model data, it can be also called as Entity/Pojo

It is a class which follows rules given by Hibernate Framework

a. class must be a package statement

b. class must be a public type

c. No of tables = No of classes

d. Class can have variables, must be private

[No of column = No of variables]

e. class should have zero argument constructor and setter-getter methods.

f. class can override toString(), hashcode(), equals() from Object class.

g. class can have annotations given by JPA and also core library annotations.

h. class can inherit (IS-A)[extends/implements] only hibernate api.

Mapping Code

1.annotation

2.xml

Annotation:

==========

1. @Entity

It maps model class with DBTable and Variables with Column Names.

2. @Id

It indicates primary key, Every table must contain primary key column.

3. @Table(optional)

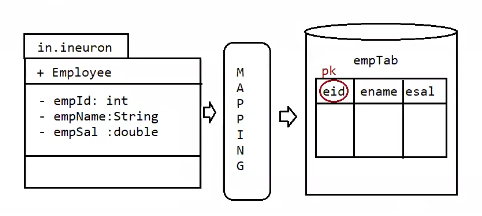
It indicates the tableName which is been mapped with Model class.

4. @Column(optional)

It indicates the columName of table which is been mapped with Model class.

Note:

if @Table, @Column are not provided then by default className is TableName, variableName is ColumnName(taken by hibernate)



eg#1

@Entity

@Table(name="empTab")

public class Employee{

@ld

@Column(name="eid")

private int empld;

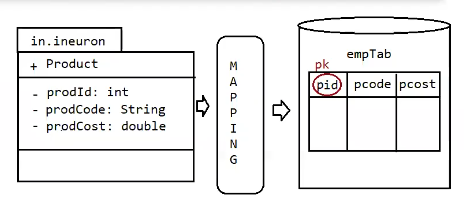
@Column(name="ename")

private String empName;

@Column(name="esal")

private double empSal;

}



eg#2.

@Entity

@Table(name="prodTab")

public class Product{

@ld

@Column(name="pid")

private int prodld;

@Column(name="pcode")

private String prodCode;

@Column(name="pcost")

private double prodCost;

}

Maping w.r.t XML

Eg1:

Employee.hbm.xml

=================

<hibernate-mapping>

<class name="in.ineuron.model.Employee" table="empTab">

<id name="empld" column="eid"/>

<property name="empName" column="ename" />

<property name="empSal" column="esal" />

</class>

</hibernate-mapping>

eg#2

Product.hbm.xml

================

<hibernate-mapping>

<class name="in.ineuron.model.Product" table="prodTab">

<id name="prodld" column="pid"/>

<property name="prodCode" column="pcode" />

<property name="prodCost" column="pcost" />

</class>

</hibernate-mapping>

3. Configuration file

For one application, one configuration file should be given

It is XML format.

\*\*\*\*configuration = Property + mapping class

Property => It represents key-value pair data.

hibernate.cfg.xml

===================

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -- >

<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql:///enterprisejavabatch</property>

<property name="connection.username">root</property>

<property name="connection.password">root123</property>

<!-- JDBC connection pool (use the built-in) -- >

<property name="connection.pool\_size">1</property>

<!-- SQL dialect -- >

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Echo all executed SQL to stdout -- >

<property name="show\_sql">true</property>

<!-- Format SQLOuput to stdOut --- >

<property name="format\_sql">true</property>

<!—Mapping Information>

<mapping resource="Employee.hbm.xml"/>

<mapping class = "in.ineuron.Model.Employee"/>

</session-factory>

</hibernate-configuration>

<!-- SQL dialect -- >

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

dialect => It is a class available inside package called org.hibernate.dialect,it will generate the SQLQuery when the programmer performs operation.

For every database dialect is different.

Oracle => nature of query

MySQL => nature of query

PostgreSQL => nature of query

<!-- Echo all executed SQL to stdout -- >

<property name="show\_sql">true</property>

This property is used to see the Query generated by the dialect based on the datbase environment on the console.

<!-- Echo all executed SQL to stdout -->

<property name="format\_sql">true</property>

This property is used to format the Query generated by the dialect based on the database environment on the console.

<property name = 'hibernate.hbm2ddl.auto">[validate/create/update/create-drop]</property>

validate =>hibernate creates no table, programmer should create or modify tables manually.

this is considered as default value.

create = > hibernate always creates new table, if table exists it will drop the table.

update => hibernate creates new table, if table doesnot exists, otherwise it will reuse the same table.

create-drop=>This option is used for testing purpose not in development

creates a new table and perform operation, at last it will drop the table.

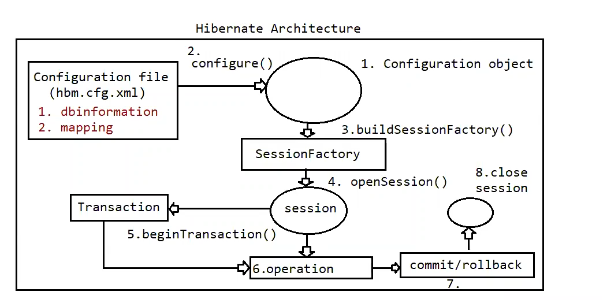
4. Test class

To perform any operation in hibernate we must write Test class.

It is used to perform operation like select/nonselect.

"Transaction" object is requried if we perform non-select operation

"Transaction" object is not required if we perform select operation.



Test class coding and its execution flow

==================================

1. Create a configuration object

2. Load .cfg.xml file into configuration object using configure().

3. Build SessionFactory object using cfg which handles

a. Loading driver class

b. Creating connection

c. Prepare statement objects.

4. use SessionFactory and get Session object to perform Persistence operation.

5. Begin Transacion, if the operation in Non-Select.

6. Now perform operation using Session object.

7. Commit or rollback if transaction has started.

8. close the session at the end.

Note: To specify the configuration details and mapping details we need to write xml file.

if the filename is hibernate.cfg.xml then it promotes automatic loading, otherwise we need to read those data from "FileInputStream".

1. Using hibernate persistence operations can be peformed using methods as shown below

a. insert query

session.save(,)

session.persist(,)

b. select query

session.load(,) => It promotes lazy loading, meaning object will be created and the values will be injected. If record doesn't exists then object won't be created.

if the record is not available it would return "ObjectNotFoundException".

session.get(,) => It promotes eager loading, meaning dummy object will be created whether record exists or not with default values only when we use the Object, it will try to pull the values and keep into the Object.

If the record doesnt exists, it would return null.

c. session.update(,)

session.saveOrUpdate(,)=> first performed selection, record found, so latest values it updated using update query.

=> first performed seelction, record not found, so perform insert operation.

d. deleteQuery

session.delete(,) : Check whether record exists, only if it exists perform deletion.

Eg: HibernateSaveOperation

Eg: SelectOperationUsingLoadMethod

Eg: DeleteOperationUsingDeleteMethod

Eg: SelectOperationUsingGetMethod