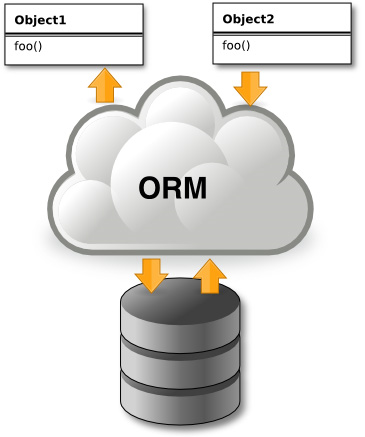
**Hibernate**

Before we can start talking about hibernate let's understand what's object relational mapping first.

**Introduction**

Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the phrase "an ORM".

An ORM library is a completely ordinary library written in your language of choice that encapsulates the code needed to manipulate the data, so you don't use SQL anymore; you interact directly with an object in the same language you're using.

**For example, here is a completely imaginary case with a pseudo language:**

You have a book class, you want to retrieve all the books of which the author is "Linus". Manually, you would do something like that:

book\_list = new List();

sql = "SELECT book FROM library WHERE author = 'John Doe'";

data = query(sql);

while (row = data.next())

{

book = new Book();

book.setAuthor(row.get('author');

book\_list.add(book);

}

With an ORM library, it would look like this:

book\_list = BookTable.query(author="John Doe");

The mechanical part is taken care of automatically via the ORM library.

* **Pros and Cons**

**Using ORM saves a lot of time because:**

1. You write your data model in only one place, and it's easier to update, maintain, and reuse the code.
2. A lot of stuff is done automatically, from database handling to I18N.
3. It forces you to write MVC code, which, in the end, makes your code a little cleaner.
4. You don't have to write poorly-formed SQL (most Web programmers really get bad at it, because SQL is treated like a "sub-language”, when in reality it's a very powerful and complex one).
5. Sanitizing; using prepared statements or transactions are as easy as calling a method.

**Using an ORM library is more flexible because:**

1. It fits in your natural way of coding (it's your language!).
2. It abstracts the DB system, so you can change it whenever you want.
3. The model is weakly bound to the rest of the application, so you can change it or use it anywhere else.
4. It lets you use OOP goodness like data inheritance without a headache.

**But ORM can be a pain:**

1. You have to learn it, and ORM libraries are not lightweight tools;
2. You have to set it up. Same problem.
3. Performance is OK for usual queries, but a SQL master will always do better with his own SQL for big projects.
4. It abstracts the DB. While it's OK if you know what's happening behind the scene, it's a trap for new programmers that can write very greedy statements, like a heavy hit in a for loop.

**How to learn about ORM?**

Whichever ORM library you choose, they all use the same principles. There are a lot of ORM libraries around here:

* Java: Hibernate.
* PHP: Propel or Doctrine (I prefer the last one).
* Python: the Django ORM or SQLAlchemy (My favorite ORM library ever).
* C#: NHibernate or Entity Framework

***Tip:*** Do not try to write your own ORM, unless you are trying to learn something. This is a gigantic piece of work, and the old ones took a lot of time and work before they became reliable.

**What is Hibernate ?**

Hibernate ORM is an object-relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database.

1. **Persistence**

Hibernate ORM is concerned with helping your application to achieve persistence. So what is persistence? Persistence simply means that we would like our application’s data to outlive the applications process. In Java terms, we would like the state of (some of) our objects to live beyond the scope of the JVM so that the same state is available later.

1. **Idiomatic persistence**

Hibernate enables you to develop persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework. Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent.

1. **High Performance**

Hibernate supports lazy initialization, numerous fetching strategies and optimistic locking with automatic versioning and time stamping. Hibernate requires no special database tables or fields and generates much of the SQL at system initialization time instead of at runtime.

Hibernate consistently offers superior performance over straight JDBC code, both in terms of developer productivity and runtime performance.

1. **Scalability**

Hibernate was designed to work in an application server cluster and deliver a highly scalable architecture. Hibernate scales well in any environment: Use it to drive your in-house Intranet that serves hundreds of users or for mission-critical applications that serve hundreds of thousands.

1. **Reliable**

Hibernate is well known for its excellent stability and quality, proven by the acceptance and use by tens of thousands of Java developers.

1. **Extensibility**

Hibernate is highly configurable and extensible.

**The Hibernate architecture is categorized in four layers.**

1. Java application layer
2. Hibernate framework layer
3. Backhand api layer
4. Database layer

Hibernate framework uses many objects such as sessionfactory, session, transaction etc. along with existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

**Elements of Hibernate Architecture**

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

* **SessionFactory**

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

* **Session**

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

* **Transaction**

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

* **ConnectionProvider**

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

* **TransactionFactory**

It is a factory of Transaction. It is optional.

So, let’s get started. Let's look at a simple example where we will create an Employee class which has attributes like ID, name, salary. our goal is to create a console based project which can perform basic CRUD operations.

**Steps for creating our first Hibernate Program**

1. Create a simple maven java project
2. Add pom dependencies for hibernate
3. Create the Persistent class (also known as model or entity class)
4. Create the mapping file for Persistent class (classname.hbm.xml)
5. Create the Configuration file (hibernate.cfg.xml)
6. Create the class that retrieves or stores the persistent object
7. Run the application (main method)

**Pom.xml**

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.3.2.Final</version>

</dependency>

**Employee.java**

public class Employee {

int id;

String name;

float salary;

public Employee() {

}

public Employee(int id, String name, float salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

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 float getSalary() {

return salary;

}

public void setSalary(float salary) {

this.salary = salary;

}

@Override

public String toString() {

return "\n\*\*\*\*\*\*\*\*\*\*Employee Details\*\*\*\*\*\*\*\*\*\*\n" +

"Id: " + id + "\n" +

"Name: " + name + "\n" +

"Salary: " + salary + "\n" ;

}

}

**Employeedetails.hbm.xml**

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="model.Employee" >

<id name="id">

<generator class="increment"></generator>

</id>

<property name="name"></property>

<property name="salary"></property>

</class>

</hibernate-mapping>

**Hibernate.cfg.xml**

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="connection.url">**jdbc:mysql://localhost/test**</property>

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

<property name="hibernate.connection.username">

**root**

</property>

<property name="hibernate.connection.password">

**admin1**

</property>

<property name="hibernate.dialect">

org.hibernate.dialect.**MySQL5Dialect**

</property>

<property name="hbm2ddl.auto">**update**</property>

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

</session-factory>

</hibernate-configuration>

**HibernateConfigurations.java**

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class HibernateConfigurations {

Configuration config;

SessionFactory factory;

Session session;

Transaction transaction;

public HibernateConfigurations() {

config = new Configuration();

config.configure("hibernate.cfg.xml");

factory = config.buildSessionFactory();

session = factory.openSession();

}

public Configuration getConfig() {

return config;

}

public void setConfig(Configuration config) {

this.config = config;

}

public SessionFactory getFactory() {

return factory;

}

public void setFactory(SessionFactory factory) {

this.factory = factory;

}

public Session getSession() {

return session;

}

public void setSession(Session session) {

this.session = session;

}

public Transaction getTransaction() {

return transaction;

}

public void setTransaction(Transaction transaction) {

this.transaction = transaction;

}

}

**EmployeeOperations.java (query implementation class)**

import java.util.Iterator;

import java.util.List;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.Transaction;

import com.HibernateConfigurations;

import model.Employee;

public class EmployeeFacadeImpl implements EmployeeFacade {

HibernateConfigurations hibernateConfig;

Transaction transaction;

Session session;

public Session getSession() {

return session;

}

public void setSession(Session session) {

this.session = session;

}

public Transaction getTransaction() {

return transaction;

}

public void setTransaction(Transaction transaction) {

this.transaction = transaction;

}

public EmployeeFacadeImpl(HibernateConfigurations

hibernateConfig) {

this.hibernateConfig = hibernateConfig;

this.setSession(hibernateConfig.getSession());

this.setTransaction(hibernateConfig.getTransaction());

}

public void insert(Employee e) {

transaction = session.beginTransaction();

session.save(e);

transaction.commit();

System.out.println("Employee Saved");

}

public void update(Employee e) {

transaction = session.beginTransaction();

Employee temp = (Employee) session.get(Employee.class, e.getId());

temp.setName(e.getName());

temp.setSalary(e.getSalary());

session.update(temp);

transaction.commit();

System.out.println("Employee Updated");

}

public void delete(Employee e) {

transaction = session.beginTransaction();

Employee temp = (Employee) session.get(Employee.class, e.getId());

session.delete(temp);

transaction.commit();

System.out.println("Employee Deleted");

}

public void view() {

transaction = session.beginTransaction();

Query query = session.createQuery("from Employee");

List<Employee> list = query.list();

for (Employee employee : list) {

System.out.println(employee);

}

transaction.commit();

}

}

**TestProject.java (main class)**

import java.util.Scanner;

import dao.EmployeeFacadeImpl;

import model.Employee;

public class TestProject {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

HibernateConfigurations configuration = new HibernateConfigurations();

EmployeeFacadeImpl empImpl = new EmployeeFacadeImpl(configuration);

int doChoice=0;

do {

System.out.println("1.Insert\t2.Update\t3.Delete\t4.View");

int choice = scan.nextInt();

Employee e = new Employee();

switch (choice) {

case 1:

System.out.println("Enter id, name and

salary");

e = new Employee(scan.nextInt(),

scan.next(), scan.nextFloat());

empImpl.insert(e);

break;

case 2:

System.out.println("Enter id, name and

salary");

e = new Employee(scan.nextInt(),

scan.next(), scan.nextFloat());

empImpl.update(e);

break;

case 3:

System.out.println("Enter id");

e.setId(scan.nextInt());

empImpl.delete(e);

break;

case 4:

empImpl.view();

break;

default:

break;

}

System.out.println("1.Re Run\t\t2.Exit");

doChoice = scan.nextInt();

} while (doChoice==1);

configuration.getSession().close();

System.out.println("\*\*\*\*\*\*\*\*Good Bye For

Now\*\*\*\*\*\*\*\*");

}

}

**Generator classes in Hibernate**

All the generator classes implements the org.hibernate.id.IdentifierGenerator interface. The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

**Hibernate Inheritance Mapping**

Hibernate has a very powerful feature which allows us to maintain a polymorphic inheritance kind of design while persisting the objects. Usually when we are using relational databases the only way to establish a relation between two tables is with the help of a foreign key or a composite key. However since we are using an object oriented programming paradigm as a business logic layer, it would be great to use parent and child class type of relation even for storing data in the database.

We can map the inheritance hierarchy classes with the tables of the database. There are three inheritance mapping strategies defined in the hibernate:

1. **Table Per Hierarchy**
   1. In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table.
   2. By this inheritance strategy, we can map the whole hierarchy by single table only. Here, an extra column (also known as discriminator column) is created in the table to identify the class.
   3. Mapping File:

<hibernate-mapping>

<class name="model.Employee" table="EmployeeHierarchy" discriminator-value="employee">

<id name="id">

<generator class="assigned">

</generator>

</id>

<discriminator column="type" type="string">

</discriminator>

<property name="name"></property>

<property name="salary"></property>

<subclass name="model.Programmer"

discriminator-value="programmer">

<property name="technology"></property>

</subclass>

<subclass name="model.Tester"

discriminator-value="tester">

<property name="mode"></property>

</subclass>

</class>

</hibernate-mapping>

1. **Table Per Concrete class**
   1. In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.
   2. In case of Table Per Concrete class, there will be multiple tables in the database having no relations to each other. There are two ways to map the table with table per concrete class strategy.
      1. By union-subclass element
      2. By self creating the table for each class
   3. Mapping File:

<hibernate-mapping>

<class name="model.Employee" table="EmployeeBase">

<id name="id">

<generator class="assigned"></generator>

</id>

<property name="name"></property>

<property name="salary"></property>

<union-subclass name="model.Programmer"

table="Programmer">

<property name="technology"></property>

</union-subclass>

<union-subclass name="model.Tester" table="Tester">

<property name="mode"></property>

</union-subclass>

</class>

</hibernate-mapping>

1. **Table Per Subclass**
   1. In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.
   2. In case of Table Per Subclass, subclass mapped tables are related to parent class mapped table by primary key and foreign key relationship. The <joined-subclass> element of class is used to map the child class with parent using the primary key and foreign key relation.
   3. Mapping File:

<hibernate-mapping>

<class name="model.Employee"table="Employee\_Parent">

<id name="id">

<generator class="assigned"></generator>

</id>

<property name="name"></property>

<property name="salary"></property>

<joined-subclass name="model.Programmer" table="Programmer\_Child">

<key column="eid"></key>

<property name="technology"></property>

</joined-subclass>

<joined-subclass name="model.Tester" table="Tester\_Child">

<key column="eid"></key>

<property name="mode"></property>

</joined-subclass>

</class>

</hibernate-mapping>

**Collection Mapping in Hibernate**

There’s the standard problem, A lot of times we need to represent the data stored in the database into representation of a collection item. It becomes very tiring to write the code to retrieve the data and convert it into the collection object manually. We can map collection elements of Persistent class in Hibernate.

You need to declare the type of collection in Persistent class from one of the following types:

1. java.util.List
2. java.util.Set
3. java.util.SortedSet
4. java.util.Map
5. java.util.SortedMap
6. java.util.Collection
7. or write the implementation of org.hibernate.usertype.UserCollectionType

public class Question {

private int id;

private String qname;

private List<String> answers;//List can be of any type

//getters and setters

}

Mapping:

<class name="com.javatpoint.Question" table="q100">

<id name="id">

<generator class="increment"></generator>

</id>

<property name="qname"></property>

<list name="answers" table="ans100">

<key column="qid"></key>

<index column="type"></index>

<element column="answer" type="string"></element>

</list>

</class>

We can assign cardinality that is one to one and one to many to each of the collection Association mapping that we so choose.

**Transaction Management**

A transaction simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling

**session.beginTransaction().**

The methods of Transaction interface are as follows:

1. void begin() starts a new transaction.
2. void commit() ends the unit of work unless we are in FlushMode.NEVER.
3. void rollback() forces this transaction to rollback.
4. void setTimeout(int seconds) it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. boolean isAlive() checks if the transaction is still alive.
6. void registerSynchronization(Synchronization s) registers a user synchronization callback for this transaction.
7. boolean wasCommited() checks if the transaction is commited successfully.
8. boolean wasRolledBack() checks if the transaction is rolledback successfully.

**Hibernate Query Language (HQL)**

Hibernate Query Language (HQL) is same as SQL (Structured Query Language) but it doesn't depend on the table of the database. Instead of table name, we use class name in HQL. So it is database independent query language.

**Advantage of HQL**

1. Database independent
2. Supports polymorphic queries
3. Easy to learn for Java Programmer

**Query Interface**

It is an object oriented representation of Hibernate Query. The object of Query can be obtained by calling the createQuery() method Session interface.

The query interface provides many methods. There is given commonly used methods:

1. public int executeUpdate() is used to execute the update or delete query.
2. public List list() returns the result of the ralation as a list.
3. public Query setFirstResult(int rowno) specifies the row number from where record will be retrieved.
4. public Query setMaxResult(int rowno) specifies the no. of records to be retrieved from the relation (table).
5. public Query setParameter(int position, Object value) it sets the value to the JDBC style query parameter.
6. public Query setParameter(String name, Object value) it sets the value to a named query parameter.

E.g.

Query query=session.createQuery("from Employee");

List list=query.list();

**HCQL (Hibernate Criteria Query Language)**

The Hibernate Criteria Query Language (HCQL) is used to fetch the records based on the specific criteria. The Criteria interface provides methods to apply criteria such as retrieving all the records of table whose salary is greater than 50000 etc.

**Criteria Interface**

The Criteria interface provides many methods to specify criteria. The object of Criteria can be obtained by calling the createCriteria() method of Session interface.

The commonly used methods of Criteria interface are as follows:

1. public Criteria add(Criterion c) is used to add restrictions.
2. public Criteria addOrder(Order o) specifies ordering.
3. public Criteria setFirstResult(int firstResult) specifies the first number of record to be retreived.
4. public Criteria setMaxResult(int totalResult) specifies the total number of records to be retreived.
5. public List list() returns list containing object.
6. public Criteria setProjection(Projection projection) specifies the projection.

**Restrictions class**

Restrictions class provides methods that can be used as Criterion. The commonly used methods of Restrictions class are as follows:

1. public static SimpleExpression lt(String propertyName,Object value) sets the less than constraint to the given property.
2. public static SimpleExpression le(String propertyName,Object value) sets the less than or equal constraint to the given property.
3. public static SimpleExpression gt(String propertyName,Object value) sets the greater than constraint to the given property.
4. public static SimpleExpression ge(String propertyName,Object value) sets the greater than or equal than constraint to the given property.
5. public static SimpleExpression ne(String propertyName,Object value) sets the not equal constraint to the given property.
6. public static SimpleExpression eq(String propertyName,Object value) sets the equal constraint to the given property.
7. public static Criterion between(String propertyName, Object low, Object high) sets the between constraint.
8. public static SimpleExpression like(String propertyName, Object value) sets the like constraint to the given property.

**Order class**

The Order class represents an order. The commonly used methods of Restrictions class are as follows:

1. public static Order asc(String propertyName) applies the ascending order on the basis of given property.
2. public static Order desc(String propertyName) applies the descending order on the basis of given property.

***E.g. Get employees whose salary is more than 10000***

Criteria c=session.createCriteria(Emp.class);

c.add(Restrictions.gt("salary",10000));

List list=c.list();

***E.g. Get employees in ascending order***

Criteria c=session.createCriteria(Emp.class);

c.addOrder(Order.asc("salary"));

List list=c.list();

**Caching in hibernate**

Caching functionality is designed to reduce the amount of necessary database access. When the objects that are cached reside in memory. You have the flexibility to limit the usage of memory and store the items in disk storage. The implementation will depend on the underlying cache manager. There are various flavors of caching available, but it is better to cache non-transactional and read-only data.

Hibernate provides 3 types of caching.

**1. Session Cache**

The session cache caches objects within the current session. It is enabled by default in Hibernate. Read more about Session Cache . Objects in the session cache reside in the same memory location.

**2. Second Level Cache**

The second level cache is responsible for caching objects across sessions. When this is turned on, objects will first be searched in the cache and if they are not found, a database query will be fired. Read here on how to implement Second Level Cache. Second level cache will be used when the objects are loaded using their primary key. This includes fetching of associations. Second level cache objects are constructed and reside in different memory locations.

**3. Query Cache**

Query Cache is used to cache the results of a query. Read here on how to implement query cache. When the query cache is turned on, the results of the query are stored against the combination query and parameters. Every time the query is fired the cache manager checks for the combination of parameters and query. If the results are found in the cache, they are returned, otherwise a database transaction is initiated. As you can see, it is not a good idea to cache a query if it has a number of parameters, because then a single parameter can take a number of values. For each of these combinations the results are stored in the memory. This can lead to extensive memory usage.

Git: <https://github.com/jaydeepresources/Tibco/tree/spring-rest-hibernate>