# Hibernate Tutorial:

hibernate tutorial with example

This hibernate tutorial provides in-depth concepts of Hibernate Framework with simplified examples. It was started in 2001 by Gavin King as an alternative to EJB2 style entity bean. The stable release of Hibernate till July 16, 2014, is hibernate 4.3.6. It is helpful for beginners and experienced persons.

# Hibernate Framework:

Hibernate framework simplifies the development of java application to interact with the database. Hibernate is an open source, lightweight, ORM (Object Relational Mapping) tool.

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

# Advantages of Hibernate Framework

There are many advantages of Hibernate Framework. They are as follows:

1) Opensource and Lightweight: Hibernate framework is opensource under the LGPL license and lightweight.

2) Fast performance: The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled bydefault.

3) Database Independent query: HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, If database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

4) Automatic table creation: Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

5) Simplifies complex join: To fetch data form multiple tables is easy in hibernate framework.

6) Provides query statistics and database status: Hibernate supports Query cache and provide statistics about query and database status.

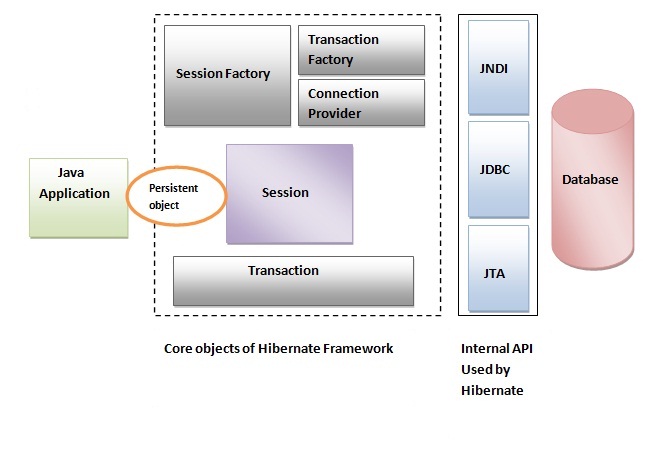
# Hibernate Architecture

The Hibernate architecture includes many objects persistent object, session factory, transaction factory, connection factory, session, transaction etc.

There are 4 layers in hibernate architecture java application layer, hibernate framework layer, backhand api layer and database layer.Let's see the diagram of hibernate architecture:



This is the high level architecture of Hibernate with mapping file and configuration file.



Hibernate framework uses many objects session factory, session, transaction etc. alongwith 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.

# First Hibernate Example without IDE

Here, we are going to create the first hibernate application without IDE. For creating the first hibernate application, we need to follow following steps:

1.Create the Persistent class

2.Create the mapping file for Persistent class

3.Create the Configuration file

4.Create the class that retrieves or stores the persistent object

5.Load the jar file

6.Run the first hibernate application without IDE

# 1) Create the Persistent class

A simple Persistent class should follow some rules:

A no-arg constructor: It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by newInstance() method.

Provide an identifier property (optional): It is mapped to the primary key column of the database.

Declare getter and setter methods (optional): The Hibernate recognizes the method by getter and setter method names by default.

Prefer non-final class: Hibernate uses the concept of proxies, that depends on the persistent class. The application programmer will not be able to use proxies for lazy association fetching.

Let's create the simple Persistent class:

# Employee.java

package com.javatpoint.mypackage;

public class Employee {

private int id;

private String firstName,lastName;

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;

} }

# 2) Create the mapping file for Persistent class

The mapping file name conventionally, should be class\_name.hbm.xml. There are many elements of the mapping file.

hibernate-mapping is the root element in the mapping file.

class It is the sub-element of the hibernate-mapping element. It specifies the Persistent class.

id It is the subelement of class. It specifies the primary key attribute in the class.

generator It is the subelement of id. It is used to generate the primary key. There are many generator classes such as assigned (It is used if id is specified by the user), increment, hilo, sequence, native etc. We will learn all the generator classes later.

property It is the subelement of class that specifies the property name of the Persistent class.

Let's see the mapping file for the Employee class:

# employee.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="com.javatpoint.mypackage.Employee" table="emp1000">

<id name="id">

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

</id>

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

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

</class>

</hibernate-mapping>

# 3) Create the Configuration file

The configuration file contains informations about the database and mapping file. Conventionally, its name should be hibernate.cfg.xml .

# 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="hbm2ddl.auto">update</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

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

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

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

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

</session-factory>

</hibernate-configuration>

# 4) Create the class that retrieves or stores the object

In this class, we are simply storing the employee object to the database.

package com.javatpoint.mypackage;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class StoreData {

public static void main(String[] args) {

//creating configuration object

Configuration cfg=new Configuration();

cfg.configure("hibernate.cfg.xml");//populates the data of the configuration file

//creating seession factory object

SessionFactory factory=cfg.buildSessionFactory();

//creating session object

Session session=factory.openSession();

//creating transaction object

Transaction t=session.beginTransaction();

Employee e1=new Employee();

e1.setId(115);

e1.setFirstName("sonoo");

e1.setLastName("jaiswal");

session.persist(e1);//persisting the object

t.commit();//transaction is commited

session.close();

System.out.println("successfully saved");

}

}

# 5) Load the jar file

For successfully running the hibernate application, you should have the hibernate4.jar file.

download the latest hibernate jar file. Some other jar files or packages are required such as

cglib

log4j

commons

SLF4J

dom4j

xalan

xerces

download the required jar files for hibernate

# 6) How to run the first hibernate application without IDE

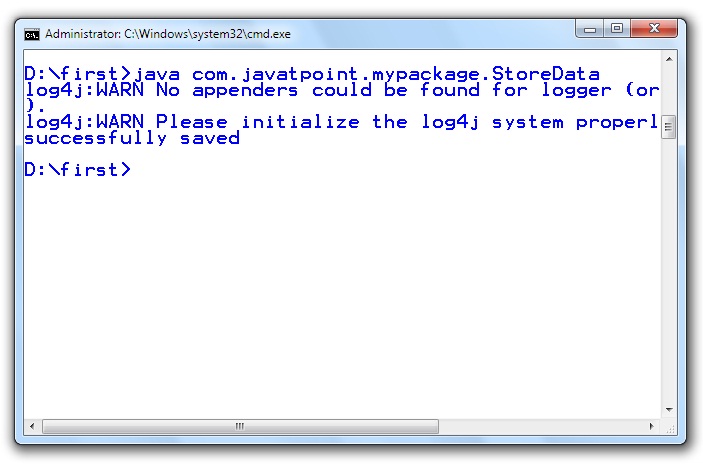
We may run this hibernate application by IDE (e.g. Eclipse, Myeclipse, Netbeans etc.) or without IDE. We will learn about creating hibernate application in Eclipse IDE in next chapter.

To run the hibernate application without IDE:

1.install the oracle10g for this example.

2.load the jar files for hibernate. (One of the way to load the jar file is copy all the jar files under the JRE/lib/ext folder). It is better to put these jar files inside the public and private JRE both.

3.Now Run the StoreData class by java com.javatpoint.mypackage.StoreData



# Hibernate with Annotation

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.

Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the javax.persistence.\* package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

# Example to create the hibernate application with Annotation

There are 4 steps to create the hibernate application with annotation.

1.Add the jar file for oracle (if your database is oracle) and annotation

2.Create the Persistent class

3.Add mapping of Persistent class in configuration file

4.Create the class that retrieves or stores the persistent object

# 1) Add the jar file for oracle and annotation

For oracle you need to add ojdbc14.jar file. For using annotation, you need to add:

hibernate-commons-annotations.jar

ejb3-persistence.jar

hibernate-annotations.jar

# 2) Create the Persistent class

Here, we are creating the same persistent class which we have created in the previous topic. But here, we are using annotation.

@Entity annotation marks this class as an entity.

@Table annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name bydefault.

@Id annotation marks the identifier for this entity.

@Column annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name bydefault.

# Employee.java

package com.javatpoint;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name= "emp500")

public class Employee {

@Id

private int id;

private String firstName,lastName;

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;

}

}

# 3) Add mapping of Persistent class in configuration file

open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

<mapping class="com.javatpoint.Employee"/>

Now the configuration file will look like this:

# 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="hbm2ddl.auto">create</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

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

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

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

<mapping class="com.javatpoint.Employee"/>

</session-factory>

</hibernate-configuration>

# 4) Create the class that retrieves or stores the persistent object

In this class, we are simply storing the employee object to the database. Here, we are using the AnnotationConfiguration class to get the information of mapping from the persistent class.

package com.javatpoint.mypackage;

package com.javatpoint;

import org.hibernate.\*;

import org.hibernate.cfg.\*;

public class Test {

public static void main(String[] args) {

Session session=new AnnotationConfiguration()

.configure().buildSessionFactory().openSession();

Transaction t=session.beginTransaction();

Employee e1=new Employee();

e1.setId(1001);

e1.setFirstName("sonoo");

e1.setLastName("jaiswal");

Employee e2=new Employee();

e2.setId(1002);

e2.setFirstName("vimal");

e2.setLastName("jaiswal");

session.persist(e1);

session.persist(e2);

t.commit();

session.close();

System.out.println("successfully saved");

}

}

# Web Application with Hibernate

Here, we are going to create a web application with hibernate. For creating the web application, we are using JSP for presentation logic, Bean class for representing data and DAO class for database codes.

As we create the simple application in hibernate, we don't need to perform any extra operations in hibernate for creating web application. In such case, we are getting the value from the user using the JSP file.

# Example to create web application using hibernate

In this example, we are going to insert the record of the user in the database. It is simply a registration form.

# index.jsp

This page gets input from the user and sends it to the register.jsp file using post method.

<form action="register.jsp" method="post">

Name:<input type="text" name="name"/><br><br/>

Password:<input type="password" name="password"/><br><br/>

Email ID:<input type="text" name="email"/><br><br/>

<input type="submit" value="register"/>"

</form>

# register.jsp

This file gets all request parameters and stores this information into an object of User class. Further, it calls the register method of UserDao class passing the User class object.

<%@page import="com.javatpoint.mypack.UserDao"%>

<jsp:useBean id="obj" class="com.javatpoint.mypack.User">

</jsp:useBean>

<jsp:setProperty property="\*" name="obj"/>

<%

int i=UserDao.register(obj);

if(i>0)

out.print("You are successfully registered");

%>

# User.java

It is the simple bean class representing the Persistent class in hibernate.

package com.javatpoint.mypack;

public class User {

private int id;

private String name,password,email;

//getters and setters

}

# User.hbm.xml

It maps the User class with the table of the database.

<?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="com.javatpoint.mypack.User" table="u400">

<id name="id">

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

</id>

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

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

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

</class>

</hibernate-mapping>

# UserDao.java

A Dao class, containing method to store the instance of User class.

package com.javatpoint.mypack;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class UserDao {

public static int register(User u){

int i=0;

Session session=new Configuration().

configure().buildSessionFactory().openSession();

Transaction t=session.beginTransaction();

t.begin();

i=(Integer)session.save(u);

t.commit();

session.close();

return i;

}

}

# hibernate.cfg.xml

It is a configuration file, containing informations about the database and mapping file.

<?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="hbm2ddl.auto">create</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

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

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

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

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

</session-factory>

</hibernate-configuration>

# Generator classes in Hibernate

The <generator> subelement of id used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

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:

assigned

increment

sequence

hilo

native

identity

seqhilo

uuid

guid

select

foreign

sequence-identity

# 1) assigned

It is the default generator strategy if there is no <generator> element . In this case, application assigns the id. For example:

....

<hibernate-mapping>

<class ...>

<id ...>

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

</id>

.....

</class>

</hibernate-mapping>

# 2) increment

It generates the unique id only if no other process is inserting data into this table. It generates short, int or long type identifier. The first generated identifier is 1 normally and incremented as 1. Syntax:

....

<hibernate-mapping>

<class ...>

<id ...>

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

</id>

.....

</class>

</hibernate-mapping>

# 3) sequence

It uses the sequence of the database. if there is no sequence defined, it creates a sequence automatically e.g. in case of Oracle database, it creates a sequence named HIBERNATE\_SEQUENCE. In case of Oracle, DB2, SAP DB, Postgre SQL or McKoi, it uses sequence but it uses generator in interbase. Syntax:

.....

<id ...>

<generator class="sequence"></generator>

</id>

.....

For defining your own sequence, use the param subelement of generator.

.....

<id ...>

<generator class="sequence">

<param name="sequence">your\_sequence\_name</param>

</generator>

</id>

.....

# 4) hilo

It uses high and low algorithm to generate the id of type short, int and long. Syntax:

.....

<id ...>

<generator class="hilo"></generator>

</id>

.....

# 5) native

It uses identity, sequence or hilo depending on the database vendor. Syntax:

.....

<id ...>

<generator class="native"></generator>

</id>

.....

# 6) identity

It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column. The returned id is of type short, int or long.

# 7) seqhilo

It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

# 8) uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

# 9) guid

It uses GUID generated by database of type string. It works on MS SQL Server and MySQL.

# 10) select

It uses the primary key returned by the database trigger.

# 11) foreign

It uses the id of another associated object, mostly used with <one-to-one> association.

# 12) sequence-identity

It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only.

# SQL Dialects in Hibernate

For connecting any hibernate application with the database, you must specify the SQL dialects. There are many Dialects classes defined for RDBMS in the org.hibernate.dialect package. They are as follows:

RDBMS Dialect

Oracle (any version) org.hibernate.dialect.OracleDialect

Oracle9i org.hibernate.dialect.Oracle9iDialect

Oracle10g org.hibernate.dialect.Oracle10gDialect

MySQL org.hibernate.dialect.MySQLDialect

MySQL with InnoDB org.hibernate.dialect.MySQLInnoDBDialect

MySQL with MyISAM org.hibernate.dialect.MySQLMyISAMDialect

DB2 org.hibernate.dialect.DB2Dialect

DB2 AS/400 org.hibernate.dialect.DB2400Dialect

DB2 OS390 org.hibernate.dialect.DB2390Dialect

Microsoft SQL Server org.hibernate.dialect.SQLServerDialect

Sybase org.hibernate.dialect.SybaseDialect

Sybase Anywhere org.hibernate.dialect.SybaseAnywhereDialet

PostgreSQL org.hibernate.dialect.PostgreSQLDialect

SAP DB org.hibernate.dialect.SAPDBDialect

Informix org.hibernate.dialect.InformixDialect

HypersonicSQL org.hibernate.dialect.HSQLDialect

Ingres org.hibernate.dialect.IngresDialect

Progress org.hibernate.dialect.ProgressDialect

Mckoi SQL org.hibernate.dialect.MckoiDialect

Interbase org.hibernate.dialect.InterbaseDialect

Pointbase org.hibernate.dialect.PointbaseDialect

FrontBase org.hibernate.dialect.FrontbaseDialect

Firebird org.hibernate.dialect.FirebirdDialect

# Hibernate Logging by Log4j using xml file

Logging enables the programmer to write the log details into a file permanently. Log4j and Logback frameworks can be used in hibernate framework to support logging.

There are two ways to perform logging using log4j:

1.By log4j.xml file (or)

2.By log4j.properties file

# Steps to perform Hibernate Logging by Log4j using xml file

There are two ways to perform logging using log4j using xml file:

1.Load the log4j jar files with hibernate

2.Create the log4j.xml file inside the src folder (parallel with hibernate.cfg.xml file)

# Example of Hibernate Logging by Log4j using xml file

You can enable logging in hibernate by following only two steps in any hibernate example. This is the first example of hibernate application with logging support using log4j.

# Load the required jar files

You need to load the slf4j.jar and log4j.jar files with hibernate jar files.

download all the required jar files

# Create log4j.xml file

Now you need to create log4j.xml file. In this example, all the log details will be written in the C:/javatpointlog.log file.

# log4j.xml

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

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"

debug="false">

<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p %c{2}: %m%n" />

</layout>

</appender>

<appender name="ASYNC" class="org.apache.log4j.AsyncAppender">

<appender-ref ref="CONSOLE" />

<appender-ref ref="FILE" />

</appender>

<appender name="FILE" class="org.apache.log4j.RollingFileAppender">

<param name="File" value="C:/javatpointlog.log" />

<param name="MaxBackupIndex" value="100" />

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p %c{2}: %m%n" />

</layout>

</appender>

<category name="org.hibernate">

<priority value="DEBUG" />

</category>

<category name="java.sql">

<priority value="debug" />

</category>

<root>

<priority value="INFO" />

<appender-ref ref="FILE" />

</root>

</log4j:configuration>

# Hibernate Logging by Log4j using properties file

As we know, Log4j and Logback frameworks are used to support logging in hibernate, there are two ways to perform logging using log4j:

1.By log4j.xml file (or)

2.By log4j.properties file

Here, we are going to enable logging using log4j through properties file.

# Steps to perform Hibernate Logging by Log4j using properties file

There are two ways to perform logging using log4j using properties file:

1.Load the log4j jar files with hibernate

2.Create the log4j.properties file inside the src folder (parallel with hibernate.cfg.xml file)

# Example of Hibernate Logging by Log4j using properties file

You can enable logging in hibernate by following only two steps in any hibernate example. This is the first example of hibernate application with logging support using log4j.

# Load the required jar files

You need to load the slf4j.jar and log4j.jar files with hibernate jar files.

download all the required jar files

# Create log4j.properties file

Now you need to create log4j.properties file. In this example, all the log details will be written in the C:\\javatpointhibernate.log file.

# log4j.properties

# Direct log messages to a log file

log4j.appender.file=org.apache.log4j.RollingFileAppender

log4j.appender.file.File=C:\\javatpointhibernate.log

log4j.appender.file.MaxFileSize=1MB

log4j.appender.file.MaxBackupIndex=1

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

# Direct log messages to stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

# Root logger option

log4j.rootLogger=INFO, file, stdout

# Log everything. Good for troubleshooting

log4j.logger.org.hibernate=INFO

# Log all JDBC parameters

log4j.logger.org.hibernate.type=ALL

# Hibernate Inheritance Mapping Tutorial

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

1.Table Per Hierarchy

2.Table Per Concrete class

3.Table Per Subclass

# Table Per Hierarchy

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 .

Table Per Hierarchy using xml file

Table Per Hierarchy using Annotation

# Table Per Concrete class

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

Table Per Concrete class using xml file

Table Per Concrete class using Annotation

# Table Per Subclass

In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

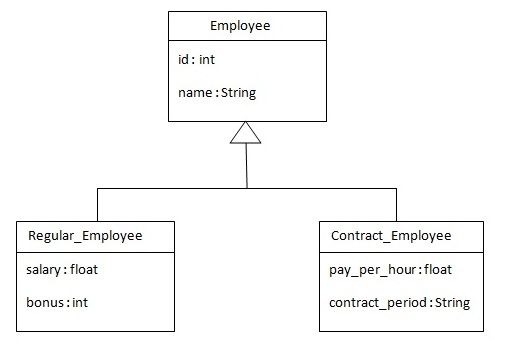
Table Per Subclass using xml file

Table Per Subclass using Annotation

# Hibernate Table Per Hierarchy using xml file

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.

Let's understand the problem first. I want to map the whole hierarchy given below into one table of the database.



There are three classes in this hierarchy. Employee is the super class for Regular\_Employee and Contract\_Employee classes. Let's see the mapping file for this hierarchy.

<?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="com.javatpoint.mypackage.Employee" table="emp121" discriminator-value="emp">

<id name="id">

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

</id>

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

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

<subclass name="com.javatpoint.mypackage.Regular\_Employee" discriminator-value="reg\_emp">

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

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

</subclass>

<subclass name="com.javatpoint.mypackage.Contract\_Employee" discriminator-value="con\_emp">

<property name="pay\_per\_hour"></property>

<property name="contract\_duration"></property>

</subclass>

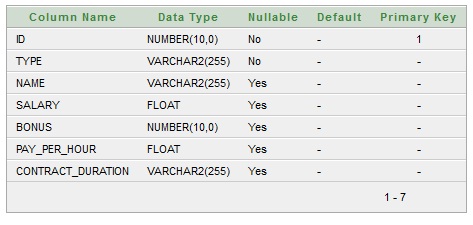
</class>

</hibernate-mapping>

In case of table per class hierarchy an discriminator column is added by the hibernate framework that specifies the type of the record. It is mainly used to distinguish the record. To specify this, discriminator subelement of class must be specified.

The subclass subelement of class, specifies the subclass. In this case, Regular\_Employee and Contract\_Employee are the subclasses of Employee class.

# The table structure for this hierarchy is as shown below:



# Example of Table per class hierarchy

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

# 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

# File: Employee.java

package com.javatpoint.mypackage;

public class Employee {

private int id;

private String name;

//getters and setters

}

# File: Regular\_Employee.java

package com.javatpoint.mypackage;

public class Regular\_Employee extends Employee{

private float salary;

private int bonus;

//getters and setters

}

File: Contract\_Employee.java

package com.javatpoint.mypackage;

public class Contract\_Employee extends Employee{

private float pay\_per\_hour;

private String contract\_duration;

//getters and setters

}

# 2) Create the mapping file for Persistent class

The mapping has been discussed above for the hierarchy.

# File: employee.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="com.javatpoint.mypackage.Employee" table="emp121" discriminator-value="emp">

<id name="id">

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

</id>

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

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

<subclass name="com.javatpoint.mypackage.Regular\_Employee" discriminator-value="reg\_emp">

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

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

</subclass>

<subclass name="com.javatpoint.mypackage.Contract\_Employee" discriminator-value="con\_emp">

<property name="pay\_per\_hour"></property>

<property name="contract\_duration"></property>

</subclass>

</class>

</hibernate-mapping>

# 3) Add mapping of hbm file in configuration file

Open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

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

Now the configuration file will look like this:

# File: 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="hbm2ddl.auto">update</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

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

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

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

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

</session-factory>

</hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

# 4) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

# File: StoreData.java

package com.javatpoint.mypackage;

import org.hibernate.\*;

import org.hibernate.cfg.\*;

public class StoreData {

public static void main(String[] args) {

Session session=new Configuration().configure("hibernate.cfg.xml")

.buildSessionFactory().openSession();

Transaction t=session.beginTransaction();

Employee e1=new Employee();

e1.setName("sonoo");

Regular\_Employee e2=new Regular\_Employee();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee e3=new Contract\_Employee();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

session.close();

System.out.println("success");

}

}

Output:



# Topics in Hibernate Inheritance Mapping

Table Per Hierarchy using xml file

Table Per Hierarchy using Annotation

Table Per Concrete class using xml file

Table Per Concrete class using Annotation

Table Per Subclass using xml file

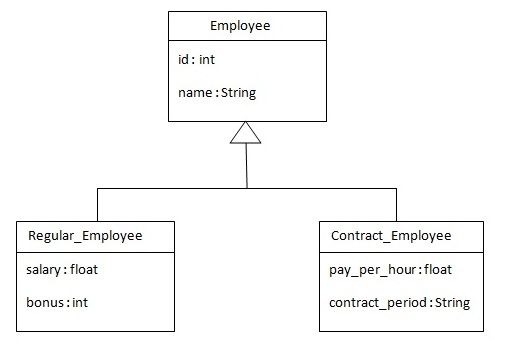
Table Per Subclass using Annotation

# Hibernate Table Per Hierarchy using Annotation

In the previous page, we have mapped the inheritance hierarchy with one table only using xml file. Here, we are going to perform this task using annotation. You need to use @Inheritance(strategy=InheritanceType.SINGLE\_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

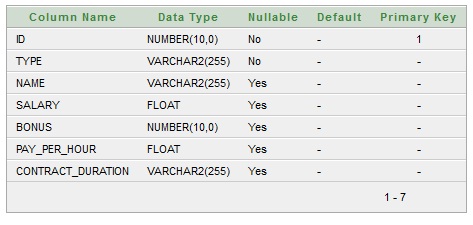
In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as discriminator column) is created in the table to identify the class.

Let's see the inheritance hierarchy:



There are three classes in this hierarchy. Employee is the super class for Regular\_Employee and Contract\_Employee classes.

The table structure for this hierarchy is as shown below:



# Example of Hibernate Table Per Hierarchy using Annotation

You need to follow following steps to create simple example:

1.Create the persistent classes

2.Create the configuration file

3.Create the class to store the fetch the data

# 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

# File: Employee.java

package com.javatpoint.mypackage;

import javax.persistence.\*;

@Entity

@Table(name = "employee101")

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)

@DiscriminatorValue(value="employee")

public class Employee {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name = "id")

private int id;

@Column(name = "name")

private String name;

//setters and getters

}

# File: Regular\_Employee.java

package com.javatpoint.mypackage;

import javax.persistence.\*;

@Entity

@DiscriminatorValue("regularemployee")

public class Regular\_Employee extends Employee{

@Column(name="salary")

private float salary;

@Column(name="bonus")

private int bonus;

//setters and getters

}

# File: Contract\_Employee.java

package com.javatpoint.mypackage;

import javax.persistence.Column;

import javax.persistence.DiscriminatorValue;

import javax.persistence.Entity;

@Entity

@DiscriminatorValue("contractemployee")

public class Contract\_Employee extends Employee{

@Column(name="pay\_per\_hour")

private float pay\_per\_hour;

@Column(name="contract\_duration")

private String contract\_duration;

//setters and getters

}

# 2) Add the persistent classes in configuration file

Open the hibernate.cgf.xml file, and add entries of entity classes like this:

<mapping class="com.javatpoint.mypackage.Employee"/>

<mapping class="com.javatpoint.mypackage.Contract\_Employee"/>

<mapping class="com.javatpoint.mypackage.Regular\_Employee"/>

</pre></div>

<table >

<tr><td>Now the configuration file will look like this:

</td></tr>

</table>

<span id="filename">File: hibernate.cfg.xml</span>

<div class="codeblock"><pre name="code" class="java" >

<?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">

<!-- Generated by MyEclipse Hibernate Tools. -->

<hibernate-configuration>

<session-factory>

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

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

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

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

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

<mapping class="com.javatpoint.mypackage.Employee"/>

<mapping class="com.javatpoint.mypackage.Contract\_Employee"/>

<mapping class="com.javatpoint.mypackage.Regular\_Employee"/>

</session-factory>

</hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

# 3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

# File: StoreTest.java

package com.javatpoint.mypackage;

import org.hibernate.\*;

import org.hibernate.cfg.\*;

public class StoreData {

public static void main(String[] args) {

AnnotationConfiguration cfg=new AnnotationConfiguration();

Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();

Transaction t=session.beginTransaction();

Employee e1=new Employee();

e1.setName("sonoo");

Regular\_Employee e2=new Regular\_Employee();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee e3=new Contract\_Employee();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

session.close();

System.out.println("success");

}

}

Output:



# Topics in Hibernate Inheritance Mapping

Table Per Hierarchy using xml file

Table Per Hierarchy using Annotation

Table Per Concrete class using xml file

Table Per Concrete class using Annotation

Table Per Subclass using xml file

Table Per Subclass using Annotation