Hibernate Intro

28 August 2014 15:07

Intro to ORM

Pros and Cons of JDBC

Pros of JDBC	Cons of JDBC
• Clean and simple SQL processing • Good performance with large data • Very good for small applications • Simple syntax so easy to learn	• Complex if it is used in large projects • Large programming overhead • No encapsulation • Hard to implement MVC concept • Query is DBMS specific

From <http://www.tutorialspoint.com/hibernate/orm_overview.htm>

What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C# etc. An ORM system has following advantages over plain JDBC

S.N.	Advantages
1	Lets business code access objects rather than DB tables.
2	Hides details of SQL queries from OO logic.
3	Based on JDBC 'under the hood'
4	No need to deal with the database implementation.
5	Entities based on business concepts rather than database structure.
6	Transaction management and automatic key generation.
7	Fast development of application.

An ORM solution consists of the following four entities:

S.N.	Solutions
1	An API to perform basic CRUD operations on objects of persistent classes.
2	A language or API to specify queries that refer to classes and properties of classes.
3	A configurable facility for specifying mapping metadata.
4	A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

Java ORM Frameworks:

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate
- And many more

From <http://www.tutorialspoint.com/hibernate/orm_overview.htm>

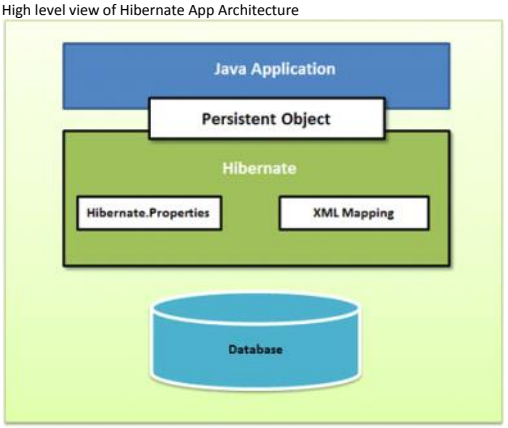
Intro to Hibernate

Hibernate Advantages:

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in Database or in any table then the only need to change XML file properties.
- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimize database access with smart fetching strategies.
- Provides Simple querying of data.

From <http://www.tutorialspoint.com/hibernate/hibernate_overview.htm>

High level view of Hibernate App Architecture



Supported Databases:

Hibernate supports almost all the major RDBMS. Following is list of few of the database engines supported by Hibernate.

- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL
- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- Informix Dynamic Server

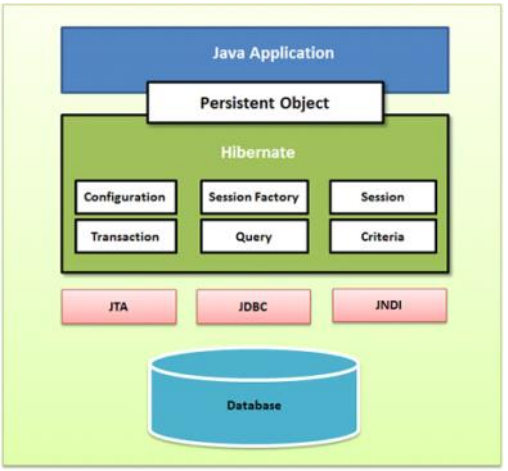
Supported Technologies:

Hibernate supports a variety of other technologies, including the following:

- XDoclet Spring
- J2EE
- Eclipse plug-ins
- Maven

From <http://www.tutorialspoint.com/hibernate/hibernate_overview.htm>

Hibernate Detailed architecture



Jars Reqd: http://www.hibernate.org/

antlr.jar

asm.jar

asm-attrs.jars
c3p0.jar
cglib.jar
commons-collections.jar
commons-logging.jar
dom4j.jar
hibernate3.jar
hsqldb.jar
jta.jar

A simple persistent class

```
package hello;
public class Message {
private Long id;
private String text;
private Message nextMessage;
Message() {}
public Message(String text) {
this.text = text;
}
public Long getId() {
return id;
}
private void setId(Long id) {
this.id = id;
}
public String getText() {
return text;
}
public void setText(String text) {
this.text = text;
}
public Message getNextMessage() {
return nextMessage;
}
public void setNextMessage(Message nextMessage) {
this.nextMessage = nextMessage;
}
}
```

Hibernate mapping for Message class

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class
name="hello.Message"
table="MESSAGES">
<id
name="id"
column="MESSAGE_ID">
<generator class="increment"/>
</id>
<property
name="text"
column="MESSAGE_TEXT"/>
<many-to-one
name="nextMessage"
cascade="all"
column="NEXT_MESSAGE_ID"
foreign-key="FK_NEXT_MESSAGE"/>
</class>
</hibernate-mapping>
```

Hello World Main Application

```
package hello;
import java.util.*;
import org.hibernate.*;
import persistence.*;
public class HelloWorld {
public static void main(String[] args) {
// First unit of work
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
Long msgId = (Long) session.save(message);
tx.commit();
session.close();
// Second unit of work
Session newSession =
HibernateUtil.getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages =
newSession.createQuery("from Message m order by m.text asc").list();
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
Message loadedMsg = (Message) iter.next();
System.out.println( loadedMsg.getText() );
}
newTransaction.commit();
newSession.close();
// Shutting down the application
HibernateUtil.shutdown();
}
}
```

Session—A Hibernate Session is many things in one. It's a single-threaded nonshared object that represents a particular unit of work with the database. It has the persistence manager API you call to load and store objects. (The Session internals consist of a queue of SQL statements that need to be synchronized with the database at some point and a map of managed persistence instances that are monitored by the Session.)

Transaction—This Hibernate API can be used to set transaction boundaries programmatically, but it's optional (transaction boundaries aren't). Other choices are JDBC transaction demarcation, the JTA interface, or container-managed transactions with EJBs.

Query—A database query can be written in Hibernate's own object-oriented query language (HQL) or plain SQL. This interface allows you to create queries, bind arguments to placeholders in the query, and execute the query in various ways.

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

To make the sessionfactory with the configuration provided in the hibernate configuration file. When new Configuration() is called, Hibernate searches for a file named hibernate.properties in the root of the classpath. If it's found, all hibernate.* properties are loaded and added to the Configuration object. When configure() is called, Hibernate searches for a file named hibernate. cfg.xml in the root of the classpath, and an exception is thrown if it can't be found.

The location of the hibernate.properties configuration file is always the root of the classpath, outside of any package.

Hibernate configuration file

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">
org.hsqldb.jdbcDriver
</property>
<property name="hibernate.connection.url">
jdbc:hsqldb:hsql://localhost
</property>
<property name="hibernate.connection.username">
sa
</property>
<property name="hibernate.dialect">
org.hibernate.dialect.HSQLDialect
</property>
<!-- Use the C3P0 connection pool provider -->
<property name="hibernate.c3p0.min_size">5</property>
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.timeout">300</property>
<property name="hibernate.c3p0.max_statements">50</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>
<!-- Show and print nice SQL on stdout -->
```

To load some other hibernate configuration file:

```
SessionFactory sessionFactory = new
Configuration() .configure("/persistence/auction.cfg.xml").buildSessionFactory();
```

Setting show_sql to true will start the logging of the sql queries executed by hibernate framework.

```
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.timeout">300</property>
<property name="hibernate.c3p0.max_statements">50</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>
<!-- Show and print nice SQL on stdout -->
<property name="show_sql">true</property>
<property name="format_sql">true</property>
<!-- List of XML mapping files -->
<mapping resource="hello/Message.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Setting show_sql to true will start the logging of the sql queries executed by hibernate framework. The queries will be printed in the console.
For logging there are other options too other than show_sql, those are: "hibernate.format_sql" , "hibernate.use_sql_comments"

Mapping File configuration
You can either list all your XML mapping files in the Hibernate XML configuration file, or you can set their names and paths programmatically on the Configuration object

Hibernate execute SQL statements asynchronously.
An INSERT statement isn't usually executed when the application calls session.save(), nor is an UPDATE immediately issued when the application calls item.setPrice(). Instead, the SQL statements are usually issued at the end of a transaction.

Logging into log file using log4j
To see output from Log4j, you need a file named log4j.properties in your classpath (right next to hibernate.properties or hibernate.cfg.xml). Also, don't forget to copy the log4j.jar library to your lib directory.

Eg of log4j.properties
```
# 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, stdout
# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=INFO
# Log JDBC bind parameter runtime arguments
log4j.logger.org.hibernate.type=INFO
```

To monitor Hibernate enable live statistics:
```
Statistics stats =
HibernateUtil.getSessionFactory().getStatistics();
stats.setStatisticsEnabled(true);
...
stats.getSessionOpenCount();
stats.logSummary();
EntityStatistics itemStats =
stats.getEntityStatistics("auction.model.Item");
itemStats.getFetchCount();
```

Structure of a project

```
WORKDIR
 build.xml
 +lib
  <all required libraries>
 +src
  +hello
       HelloWorld.java
       Message.java
       Message.hbm.xml
  +persistence
       HibernateUtil.java
  hibernate.cfg.xml (or hibernate.properties)
  log4j.properties
```

Build.xml

```
<project name="HelloWorld" default="compile" basedir=".">
<!-- Name of project and version -->
<property name="proj.name" value="HelloWorld"/>
<property name="proj.version" value="1.0"/>
<!-- Global properties for this build -->
<property name="src.java.dir" value="src"/>
<property name="lib.dir" value="lib"/>
<property name="build.dir" value="bin"/>
<!-- Classpath declaration -->
<path id="project.classpath">
<fileset dir="${lib.dir}">
<include name="**/*.jar"/>
<include name="**/*.zip"/>
</fileset>
</path>
<!-- Useful shortcuts -->
<patternset id="meta.files">
<include name="**/*.xml"/>
<include name="**/*.properties"/>
</patternset>
<!-- Clean up -->
<target name="clean">
<delete dir="${build.dir}"/>
<mkdir dir="${build.dir}"/>
</target>
<!-- Compile Java source -->
<target name="compile" depends="clean">
<mkdir dir="${build.dir}"/>
<javac
srcdir="${src.java.dir}"
destdir="${build.dir}"
nowarn="on">
<classpath refid="project.classpath"/>
</javac>
</target>
<!-- Copy metadata to build classpath -->
<target name="copymetafiles">
<copy todir="${build.dir}">
<fileset dir="${src.java.dir}">
<patternset refid="meta.files"/>
</fileset>
</copy>
</target>
<!-- Run HelloWorld -->
<target name="run" depends="compile, copymetafiles"
description="Build and run HelloWorld">
<java fork="true"
classname="hello.HelloWorld"
classpathref="project.classpath">
<classpath path="${build.dir}"/>
</java>
</target>
</project>
```

Using Annotation in place of Mapping file
-jars reqd: hibernate-annotations.jar, ejb3-persistence.jar
-remove the mapping file, and remove the entry of the mapping file with annotated class or its list

```
package hello;
import javax.persistence.*;
@Entity
@Table(name = "MESSAGES")
public class Message {
@Id @GeneratedValue
@Column(name = "MESSAGE_ID")
private Long id;
@Column(name = "MESSAGE_TEXT")
private String text;
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "NEXT_MESSAGE_ID")
private Message nextMessage;
private Message() {}
public Message(String text) {
this.text = text;
}
public Long getId() {
return id;
}
private void setId(Long id) {
this.id = id;
}
public String getText() {
return text;
}
public void setText(String text) {
this.text = text;
}
public Message getNextMessage() {
return nextMessage;
}
public void setNextMessage(Message nextMessage) {
this.nextMessage = nextMessage;
}
}
```

Changes done in the hibernate configuration file:
```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
```

Using Hibernate EntityManager:
Advantage: You no longer have to list all annotated classes (or XML mapping files) in your configuration file.

To use jpa in this project :
-Use EntityManagerFactory in place of session factory.
To use EMF a configuration file will be needed, named persistence.xml:

```
<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_1_0.xsd"
version="1.0">
<persistence-unit name="helloworld">
<properties>
<property name="hibernate.ejb.cfgfile"
value="/hibernate.cfg.xml"/>
</properties>
</persistence-unit>
</persistence>
```

Full persistence unit configuration:

```
<persistence-unit name="helloworld">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<!-- Not needed, Hibernate supports auto-detection in JSE
<class>hello.Message</class>
-->
<properties>
<property name="hibernate.archive.autodetection"
value="class, hbm"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.connection.driver_class"
value="org.hsqldb.jdbcDriver"/>
<property name="hibernate.connection.url"
value="jdbc:hsqldb:hsql://localhost"/>
<property name="hibernate.connection.username"
value="sa"/>
<property name="hibernate.c3p0.min_size"
value="5"/>
<property name="hibernate.c3p0.max_size"
value="20"/>
<property name="hibernate.c3p0.timeout"
value="300"/>
<property name="hibernate.c3p0.max_statements"
value="50"/>
<property name="hibernate.c3p0.idle_test_period"
value="3000"/>
<property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
<property name="hibernate.hbm2ddl.auto" value="create"/>
</properties>
</persistence-unit>
```

```
<!-- ... Many property settings ... -->
<!-- List of annotated classes-->
<mapping class="hello.Message"/>
</session-factory>
</hibernate-configuration>
```

Changes in the main class:

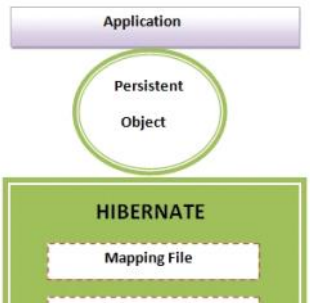
```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically
cfg.addAnnotatedClass(hello.Message.class);
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

The "Hello World" main application code with JPA

```
package hello;
import java.util.*;
import javax.persistence.*;
public class HelloWorld {
public static void main(String[] args) {
// Start EntityManagerFactory
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("helloworld");
// First unit of work
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message message = new Message("Hello World");
em.persist(message);
tx.commit();
em.close();
// Second unit of work
EntityManager newEm = emf.createEntityManager();
EntityTransaction newTx = newEm.getTransaction();
newTx.begin();
List messages = newEm
.createQuery("select m from Message m
➥ order by m.text asc")
.getResultList();
System.out.println( messages.size() + " message(s) found" );
for (Object m : messages) {
Message loadedMsg = (Message) m;
System.out.println(loadedMsg.getText());
}
newTx.commit();
newEm.close();
// Shutting down the application
emf.close();
}
}
```

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.

Hibernate Architecture



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.

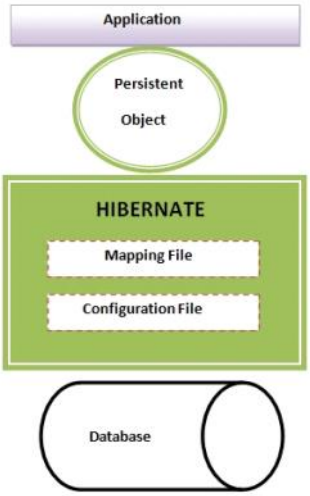
2) Fast performance: The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled bydefault.

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.



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.

Object States:

transient,
persistent, detached, or removed.

Some Imp Annotations:

-@Entity :
=name
-@Id
-@GeneratedValue(strategy=GenerationType.IDENTITY)
=strategy
=generator
Different strategies: identity, sequence, table, auto, and none.
-@SequenceGenerator

-@Column
-@OneToOne
= mappedBy

Session:

Load : for loading entities from your database.
Save
Persist
Merge: We update the detached object and merge()
it–which should update the value in the database
Refresh: will reload the properties of the object from the database, overwriting them; thus, as stated, refresh() is the inverse of merge().
saveOrUpdate
Delete: public void delete(Object object) throws HibernateException
This method takes a persistent object as an argument. The argument can also be a transient object with the
identifier set to the ID of the object that needs to be erased.
createQuery: