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.

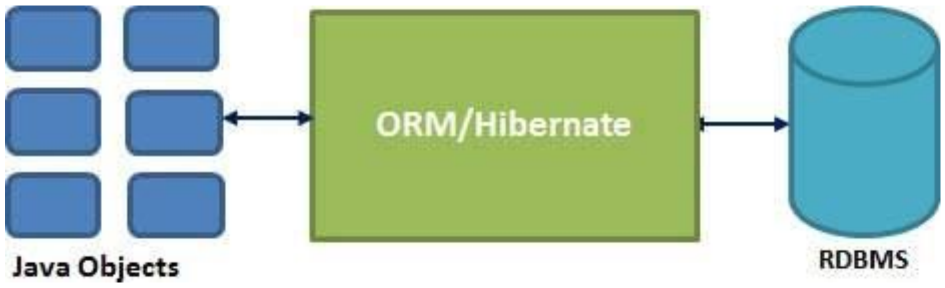
- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate
- And many more

Hibernate

Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA. It is an open source persistent framework created by **Gavin King in 2001**. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from most of the common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.



Advantages

- It 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 the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

Supported Databases

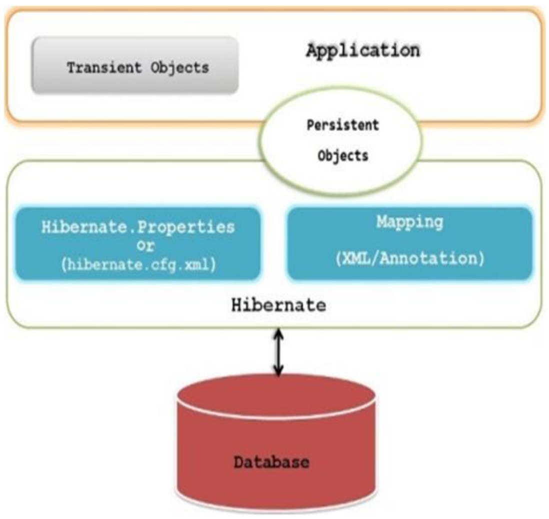
Hibernate supports almost all the major RDBMS. Following is a list of few of the database engines supported by Hibernate –

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

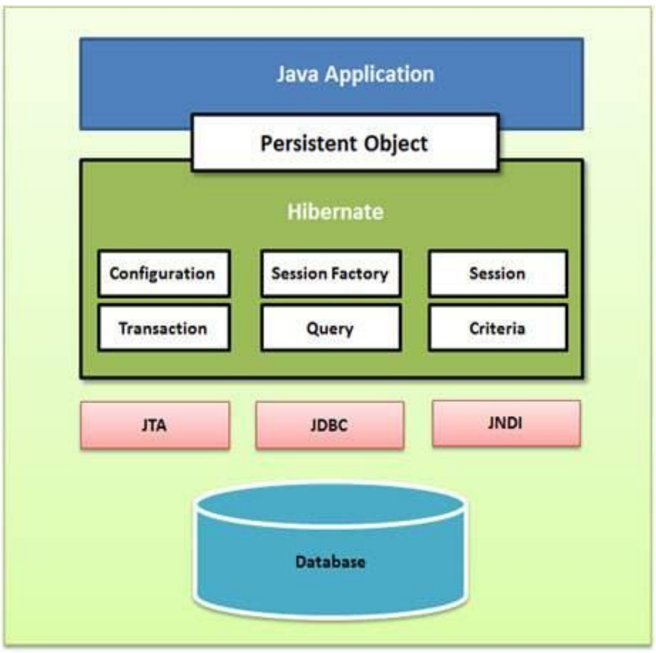
Architecture

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a high level and detailed view of the Hibernate Application Architecture.



High Level View



Detailed View with Components

__Components__

1. **Configuration Object**

   The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

The Configuration object provides two key components −

- **Database Connection** − this is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
- **Class Mapping Setup** − this component creates the mapping between the Java classes and database tables.

2. **SessionFactory Object**

   Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

   The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

3. **Session Object**

   A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

   The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

4. **Transaction Object**

   A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

   This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

5. **Query Object**

   Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

6. **Criteria Object**

   Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

Hibernate - Configuration:

Hibernate requires to know in advance — where to find the mapping information that defines how Java classes relates to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

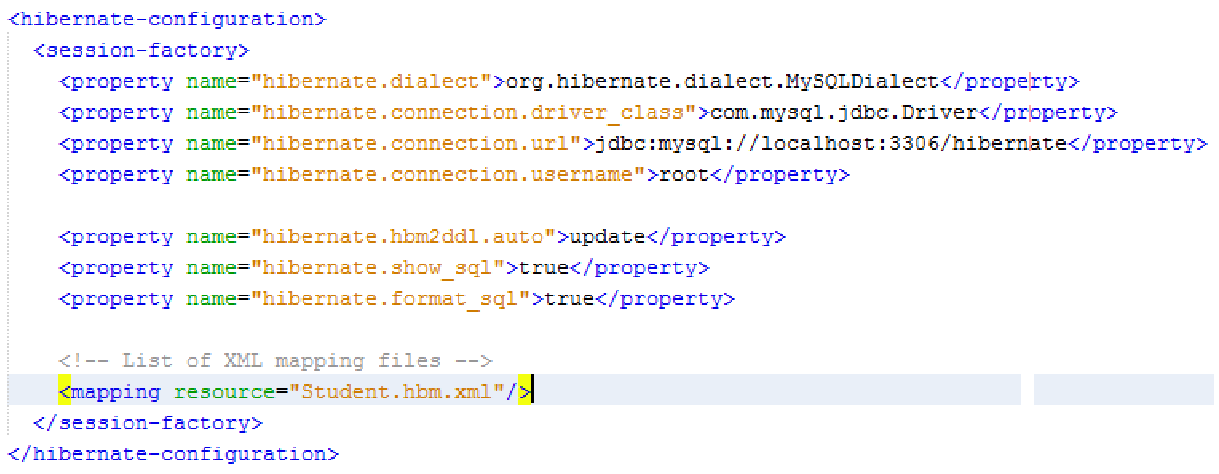
Here XML formatted file **hibernate.cfg.xml** is considered to specify required Hibernate properties. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file should be kept in the root directory of application's classpath.

Hibernate Properties

1. **hibernate.dialect -** This property makes Hibernate generate the appropriate SQL for the chosen database.
2. **hibernate.connection.driver_class -** The JDBC driver class.
3. **hibernate.connection.url -** The JDBC URL to the database instance.
4. **hibernate.connection.username -** The database username.
5. **hibernate.connection.password -** The database password.
6. **hibernate.connection.pool_size -** Limits the number of connections waiting in the Hibernate database connection pool.
7. **hibernate.connection.autocommit -** Allows auto commit mode to be used for the JDBC connection.

Example:

hibernate.cfg.xml

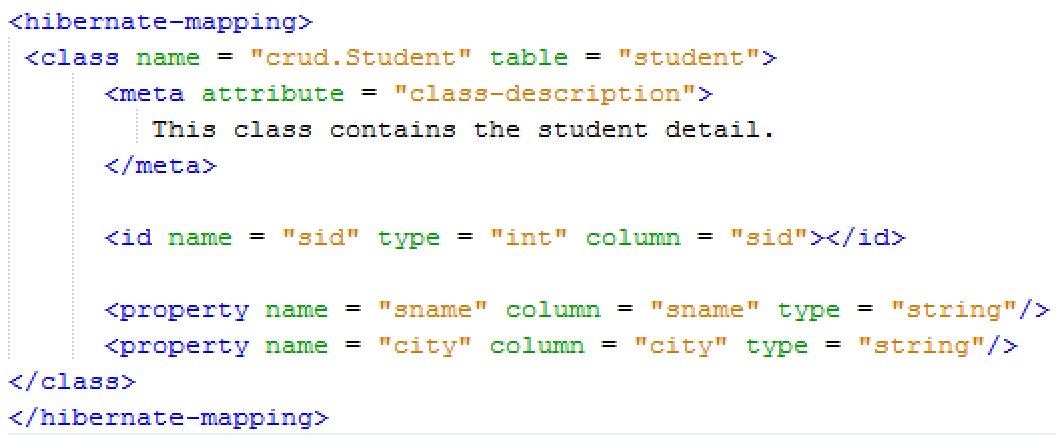


```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate</property>
    <property name="hibernate.connection.username">root</property>

    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>

    <!-- List of XML mapping files -->
    <mapping resource="Student.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

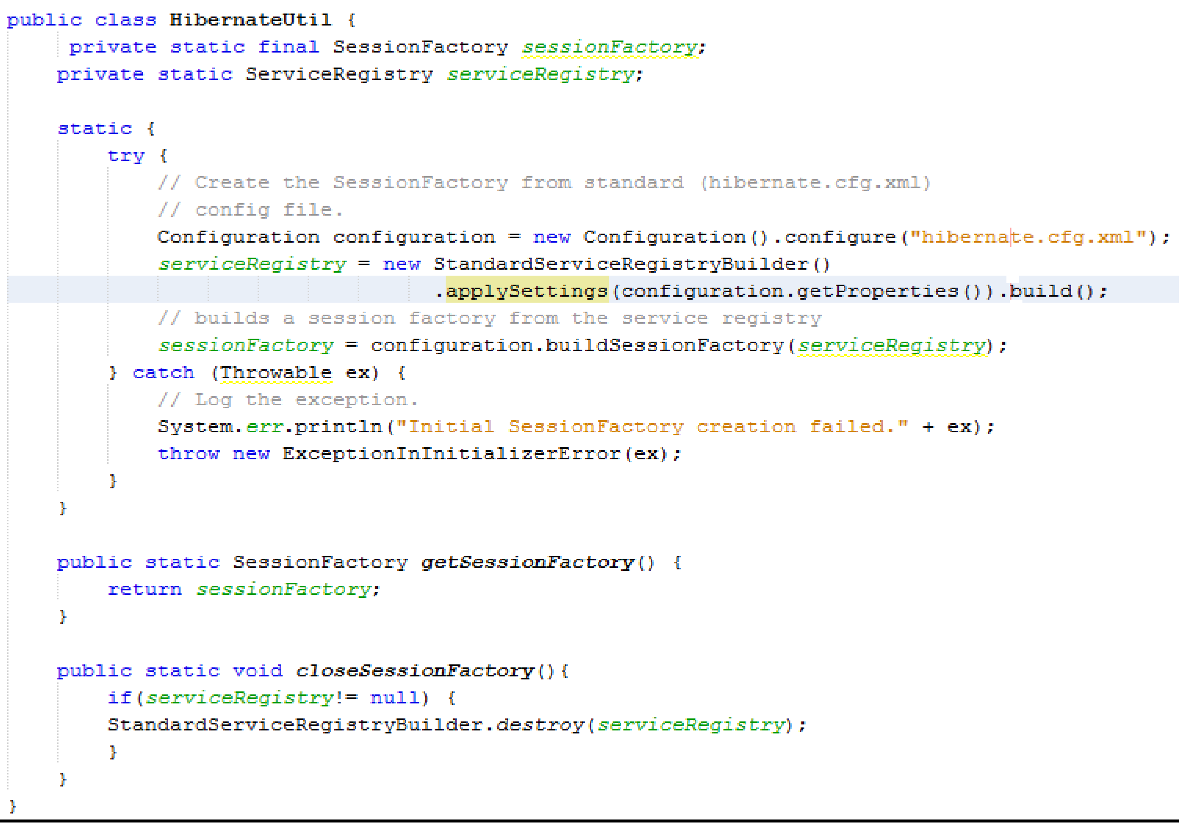
Student.hbm.xml



```
<hibernate-mapping>
 <class name = "crud.Student" table = "student">
      <meta attribute = "class-description">
         This class contains the student detail.
      </meta>

      <id name = "sid" type = "int" column = "sid"></id>

      <property name = "sname" column = "sname" type = "string"/>
      <property name = "city" column = "city" type = "string"/>
</class>
</hibernate-mapping>
```



```java
public class HibernateUtil {
     private static final SessionFactory sessionFactory;
    private static ServiceRegistry serviceRegistry;

    static {
        try {
            // Create the SessionFactory from standard (hibernate.cfg.xml)
            // config file.
            Configuration configuration = new Configuration().configure("hibernate.cfg.xml");
            serviceRegistry = new StandardServiceRegistryBuilder()
                                    .applySettings(configuration.getProperties()).build();
            // builds a session factory from the service registry
            sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void closeSessionFactory(){
        if(serviceRegistry!= null) {
        StandardServiceRegistryBuilder.destroy(serviceRegistry);
        }
    }
}
```

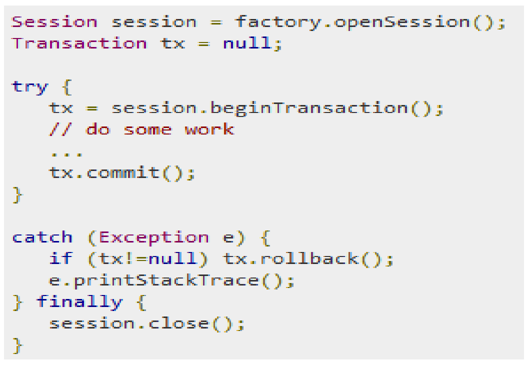
<u>Hibernate - Session:</u>

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed as needed. The main function of the Session is to offer, create, read, update and delete operations for instances of mapped entity classes.

Instances may exist in one of the following three states at a given point in time –

- **transient** – A new instance of a persistent class, which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
- **persistent** – You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
- **detached** – Once we close the Hibernate Session, the persistent instance will become a detached instance.



```java
Session session = factory.openSession();
Transaction tx = null;

try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}

catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
} finally {
   session.close();
}
```