Hibernate

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.

ORM overview

ORM Advantages

1. ORM allows the 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.

ORM Frameworks in Java

1. Castor
2. Torque
3. iBatis
4. Spring DAO
5. Java Data Objects (JDO)
6. TopLink
7. Hibernate

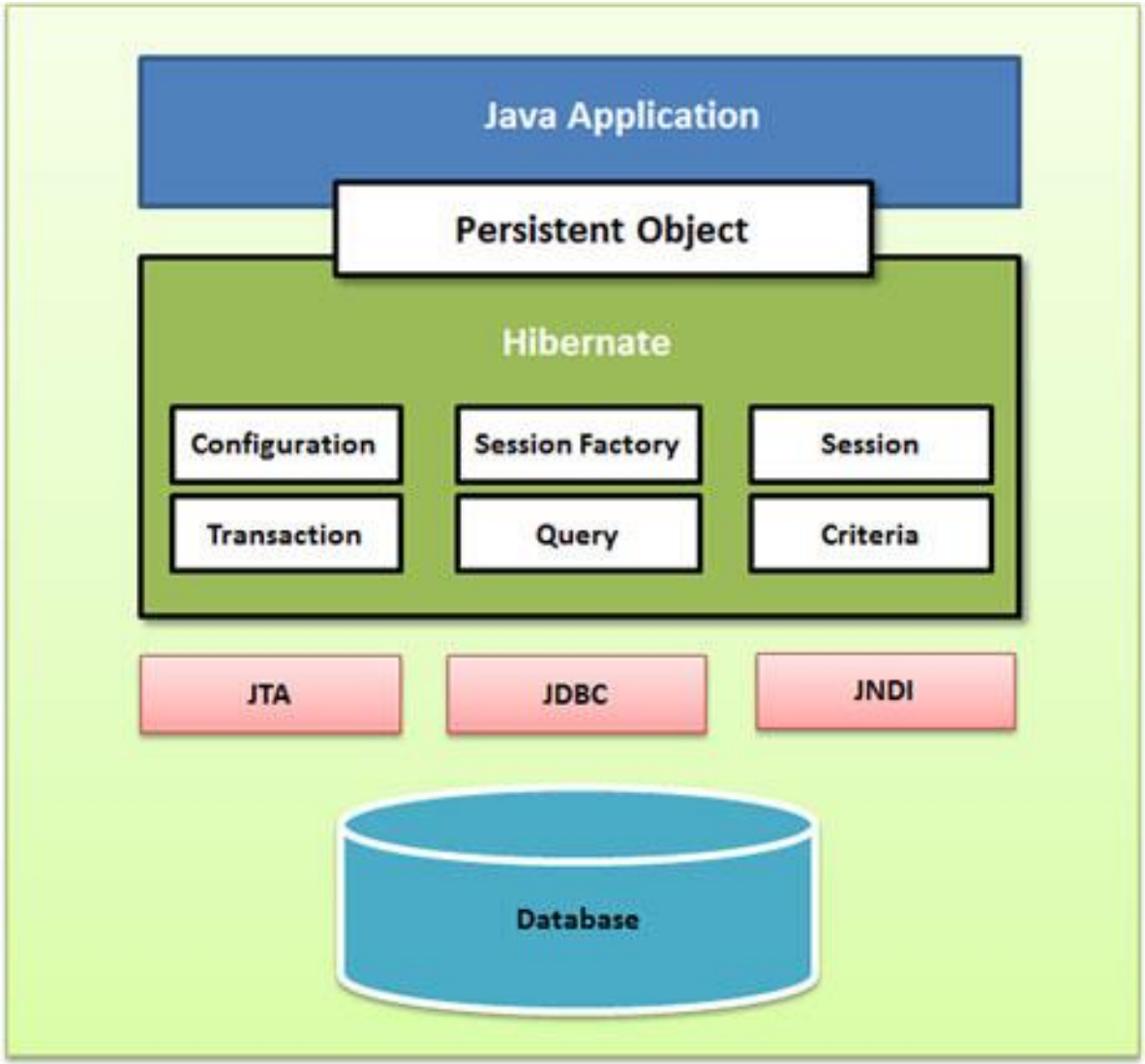
What is Hibernate?

- Hibernate is a high-performance Object/Relational persistence and query service which is licensed under the open source Lesser General Public License (LGPL) and is free to download. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.

Supported databases

1. HSQL Database Engine
2. DB2/NT
3. MySQL
4. PostgreSQL
5. FrontBase
6. Oracle
7. Microsoft SQL Server
8. Sybase SQL Server
9. Informix Dynamic Server

Hibernate Architecture



Hibernate Architecture continued..

- **The Configuration object:**

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

1. **Database Connection-**

   This is handled through one or more configuration files supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml.

2. **Class Mapping Setup-**

   This component creates the connection between the Java classes and database tables.

Hibernate Architecture continued..

- **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 heavyweight object so usually it is 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.

Hibernate Architecture continued..

- **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.

Hibernate Architecture continued..

- **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.

Hibernate Architecture continued..

- **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.

Hibernate Architecture continued..

- **Criteria Object:**

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

HQL (Hibernate Query Language)

- **FROM Clause**

  You will use FROM clause if you want to load a complete persistent objects into memory.

  HQL: *"FROM Employee"*

- **AS Clause**

  The AS clause can be used to assign aliases to the classes in your HQL queries.

  HQL: *"FROM Employee AS E"*

- **SELECT Clause**

  If you want to obtain few properties of objects instead of the complete object, use the SELECT clause.

  HQL: *"SELECT E.firstName FROM Employee E"*

HQL continued…

- **WHERE Clause**

  HQL: *“FROM Employee E WHERE E.id = 10”*

- **ORDER BY Clause**

  To sort your HQL query's results, you will need to use the ORDER BY clause.

  HQL: *“FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC”*

- **GROUP BY Clause**

  This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value.

  HQL: *“SELECT SUM(E.salary), E.firtName FROM Employee E GROUP BY E.firstName”*

HQL continued…

- **Using Named Paramters**
  This makes writing HQL queries that accept input from the user.
  String hql = "*FROM Employee E WHERE E.id = :employee_id*";
  Query query = session.createQuery(hql);
  query.setParameter("employee_id",10);
  List results = query.list();

- **UPDATE Clause**
  The UPDATE clause can be used to update one or more properties of an one or more objects.
  HQL: *“UPDATE Employee set salary = :salary WHERE id = :employee_id”*

- **DELETE Clause**
  The DELETE clause can be used to delete one or more objects.
  HQL: *“DELETE FROM Employee”*

HQL continued…

- **INSERT Clause**

  HQL supports INSERT INTO clause only where records can be inserted from one object to another object.

  HQL: *"INSERT INTO Employee(firstName, lastName, salary) SELECT firstName, lastName, salary FROM old_employee"*

- **Aggregate Methods**

  avg*(property name)* The average of a property's value

  count*(property name or *)* The number of times a property occurs in the results

  max*(property name)* The maximum value of the property values

  min*(property name)* The minimum value of the property values

  sum*(property name)* The sum total of the property values

- HQL: *"SELECT count(distinct E.firstName) FROM Employee E"*

Criteria

- Criteria API allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));
```

Criteria continued…

```
// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));
```

Criteria AND/OR condition

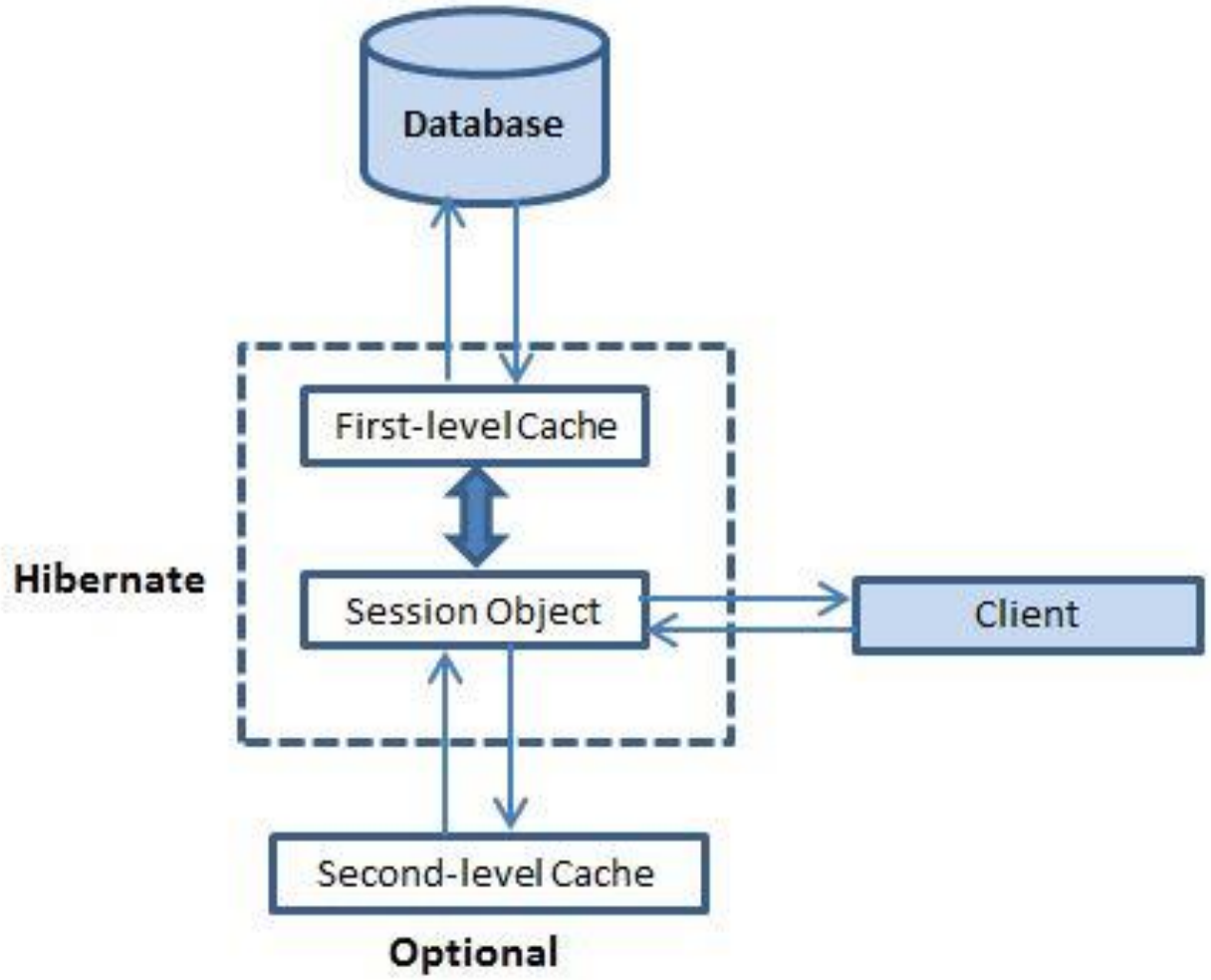
```
Criteria cr = session.createCriteria(Employee.class);
Criterion salary = Restrictions.gt("salary", 2000);
Criterion name = Restrictions.ilike("firstNname","zara%");
// To get records matching with OR condistions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );

// To get records matching with AND condistions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );
List results = cr.list();
```

Criteria Sorting

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));
// To sort records in descening order
cr.addOrder(Order.desc("salary"));
// To sort records in ascending order
cr.addOrder(Order.asc("salary"));
List results = cr.list();
```

Hibernate Caching



Hibernate Caching continued…

- **First-level cache:**

  The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

  If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Hibernate Caching continued…

- **Second-level cache:**

  Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

  Any third-party cache can be used with Hibernate. An org.hibernate.cache.CacheProvider interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation

Concurrency strategies

- **Transactional:** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update. Full Transaction Isolation, even repetable-read.
- **Read-write:** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update. Maintains read-committed isolation level.
- **Nonstrict-read-write:** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.
- **Read-only:** A concurrency strategy suitable for data which never changes. Use it for reference data only.

Concurrency strategies continued…

```
<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <cache usage="read-write"/>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```