ORM

Amit Kumar

JDBC

JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

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

JDBC

Example:

```
public class Employee {
   private int id;
   private String first_name;
   private String last_name;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.first_name = fname;
      this.last_name = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public String getFirstName() {
      return first_name;
   }
   public String getLastName() {
      return last_name;
   }
   public int getSalary() {
      return salary;
   }
}
```

JDBC

Table:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

JDBC

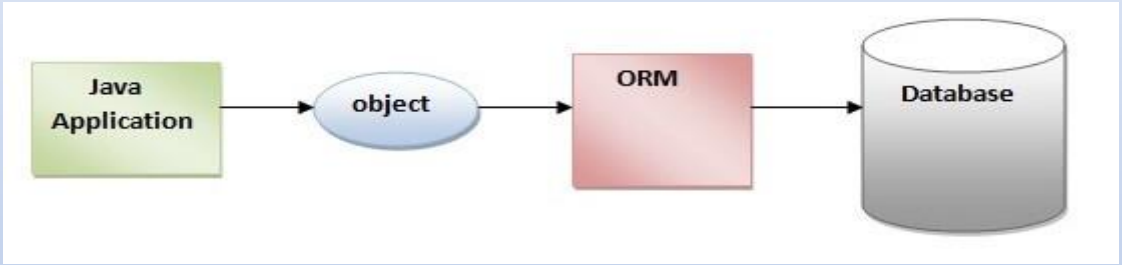
First problem, what if we need to modify the design of our database after having developed a few pages or our application? Second, loading and storing objects in a relational database exposes us to the following five mismatch problems −

Sr.No.	Mismatch & Description
1	**Granularity** Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database.
2	**Inheritance** RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages.
3	**Identity** An RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).
4	**Associations** Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.
5	**Navigation** The ways you access objects in Java and in RDBMS are fundamentally different.

The **O**bject-**R**elational **M**apping (ORM) is the solution to handle all the above impedance mismatches.

ORM

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.



Object-Relational Mapping (ORM) is the process of converting Java objects to database tables. In other words, this allows us to interact with a relational database without any SQL. **The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications.** The primary focus of JPA is the ORM layer.

ORM Frameworks

Following are the various frameworks that function on ORM mechanism: -

- Hibernate
- TopLink
- ORMLite
- iBATISs
- JPOX

ORM

Advantage:

Sr.No.	Advantages
1	Let’s 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

Benefits:

Improved performance : Lazy loading, Sophisticated caching, Eager loading.

Improved productivity : High-level object-oriented API, Less Java code to write, No SQL to write.

Improved maintainability : A lot less code to write.

Improved portability : ORM framework generates database-specific SQL for you.

ORM

An ORM solution consists of the following four entities −

Sr.No.	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.

ORM

Mapping Directions

Mapping Directions are divided into two parts: -

- **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
- **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

Following are the various ORM mappings: -

- **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
- **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
- **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
- **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

Hibernate

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

Advantages of Hibernate Framework

Following are the advantages of hibernate framework:

1) Open Source and Lightweight

Hibernate framework is open source 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 by default.

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.

Hibernate

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

Fetching data from multiple tables is easy in hibernate framework.

6) Provides Query Statistics and Database Status

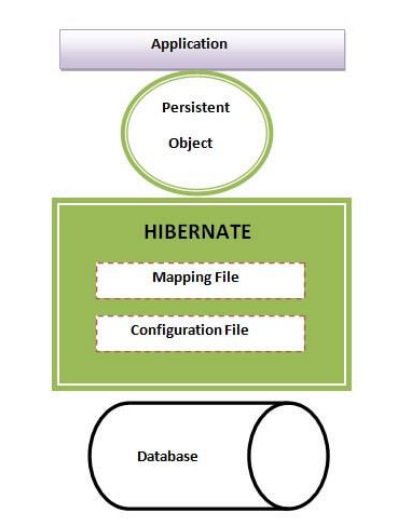
Hibernate supports Query cache and provide statistics about query and database status.

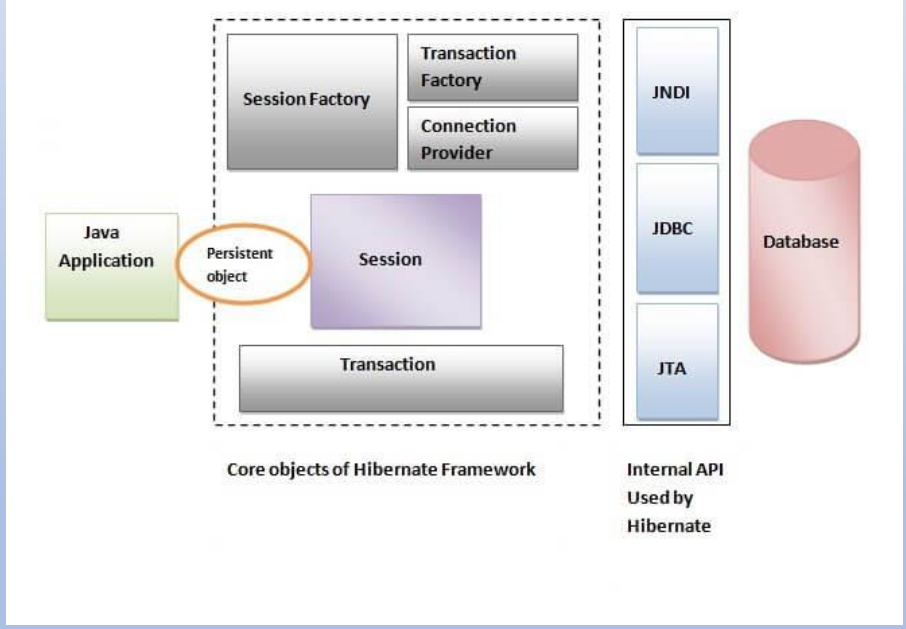
Hibernate

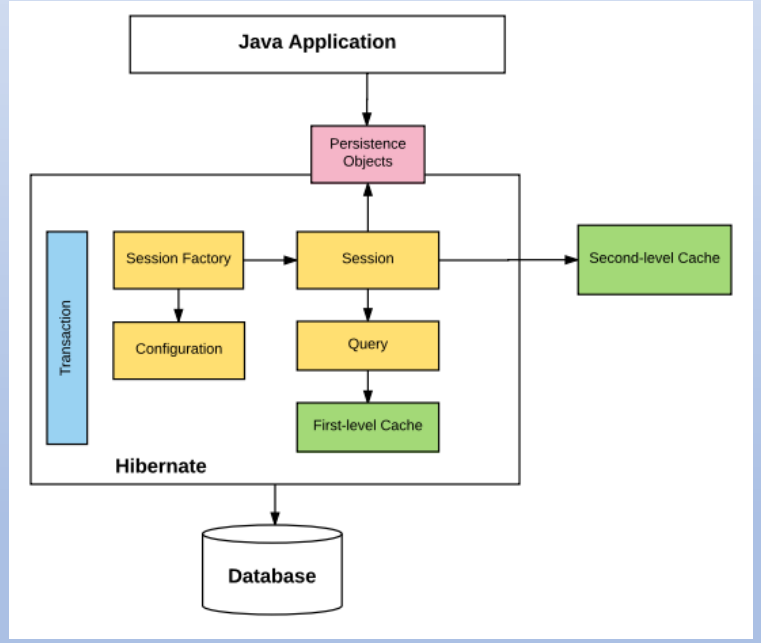
The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.

The Hibernate architecture is categorized in four layers.

- Java application layer
- Hibernate framework layer
- Backend api layer
- Database layer







Hibernate

Configuration:

Configuration: Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find the class annotated with @Configuration. It is used by SessionFactory to work with Java applications and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.

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.

Hibernate

TransactionFactory: It is a factory of Transaction. It is optional.

Query: It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query databases, including NamedQuery and Criteria API.

First-level cache: It represents the default cache used by Hibernate Session object while interacting with the database. It is also called a session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.

Transaction: enables you to achieve data consistency, and rollback in case something goes unexpected.

Persistent objects: These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate. They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.

Second-level cache: It is used to store objects across sessions. This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is EhCache.

Hibernate Entity

In general, entity is a group of states associated together in a single unit. On adding behaviour, an entity behaves as an object and becomes a major constituent of object-oriented paradigm. So, an entity is an application-defined object in Java Persistence Library.

Entity Properties

These are the properties of an entity that an object must have: -

- **Persistability -** An object is called persistent if it is stored in the database and can be accessed anytime.
- **Persistent Identity -** In Java, each entity is unique and represents as an object identity. Similarly, when the object identity is stored in a database then it is represented as persistence identity. This object identity is equivalent to primary key in database.
- **Transactionality -** Entity can perform various operations such as create, delete, update. Each operation makes some changes in the database. It ensures that whatever changes made in the database either be succeed or failed atomically.
- **Granuality -** Entities should not be primitives, primitive wrappers or built-in objects with single dimensional state.

Entity Metadata

Each entity is associated with some metadata that represents the information of it. Instead of database, this metadata is exist either inside or outside the class. This metadata can be in following forms: -

- **Annotation -** In Java, annotations are the form of tags that represents metadata. This metadata persist inside the class.
- **XML -** In this form, metadata persist outside the class in XML file.

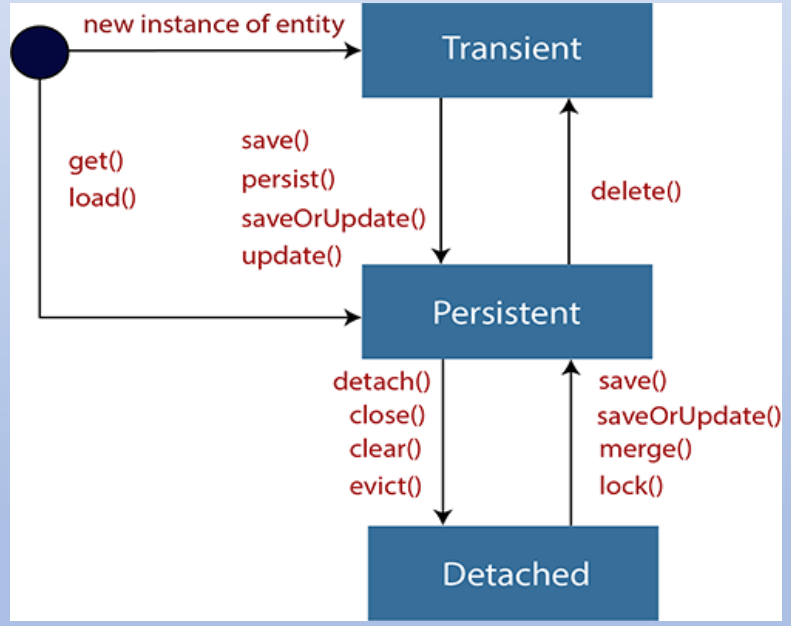
Hibernate

Hibernate Lifecycle

In Hibernate, either we create an object of an entity and save it into the database, or we fetch the data of an entity from the database. Here, each entity is associated with the lifecycle. The entity object passes through the different stages of the lifecycle.

The Hibernate lifecycle contains the following states: -

- Transient state
- Persistent state
- Detached state



Transient state

- The transient state is the initial state of an object.
- Once we create an instance of POJO class, then the object entered in the transient state.
- Here, an object is not associated with the Session. So, the transient state is not related to any database.
- Hence, modifications in the data don't affect any changes in the database.
- The transient objects exist in the heap memory. They are independent of Hibernate.

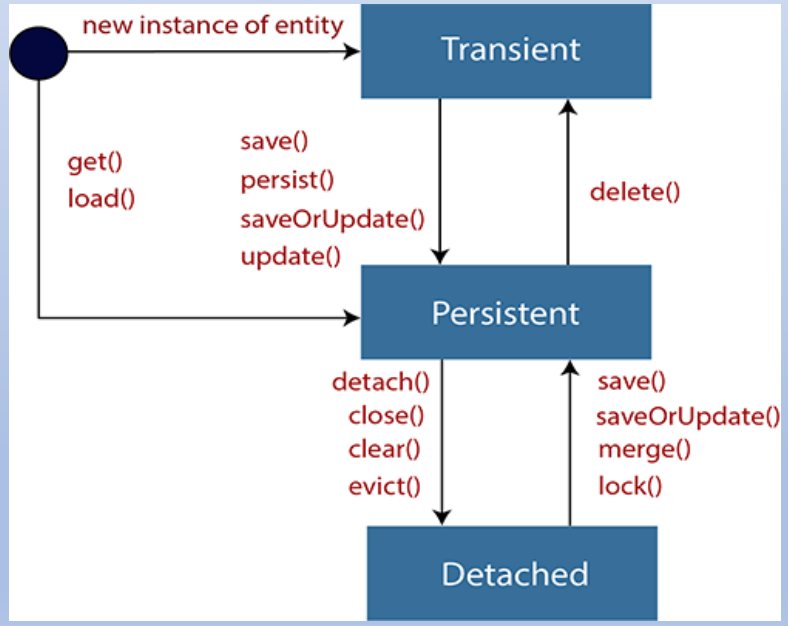
Ex: Employee e=**new** Employee(); //Here, object enters in the transient state.

Hibernate

Persistent state

- As soon as the object associated with the Session, it entered in the persistent state.
- Hence, we can say that an object is in the persistence state when we save or persist it.
- Here, each object represents the row of the database table.
- So, modifications in the data make changes in the database.

```
session.save(e);
session.persist(e);
session.update(e);
session.saveOrUpdate(e);
session.lock(e);
session.merge(e);
```

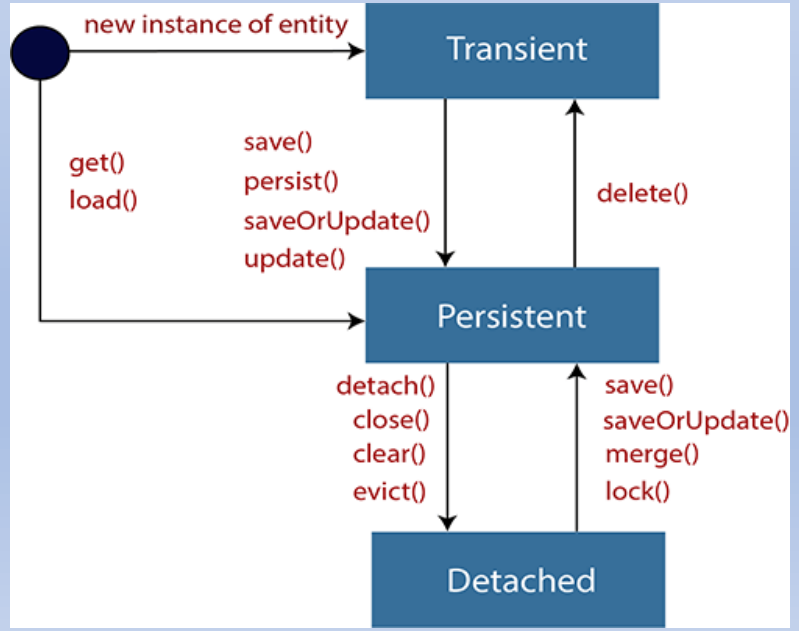


Hibernate

Detached State

- Once we either close the session or clear its cache, then the object entered into the detached state.
- As an object is no more associated with the Session, modifications in the data don't affect any changes in the database.
- However, the detached object still has a representation in the database.
- If we want to persist the changes made to a detached object, it is required to reattach the application to a valid Hibernate session.
- To associate the detached object with the new hibernate session, use any of these methods - load(), merge(), refresh(), update() or save() on a new session with the reference of the detached object.

```
session.close();
session.clear();
session.detach(e);
session.evict(e);
```



Hibernate

Dialect Classes:

The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements. For connecting any hibernate application with the database, it is required to provide the configuration of SQL dialect.

There are many Dialects classes defined for RDBMS in the **org.hibernate.dialect** package.

RDBMS	Dialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle10g	org.hibernate.dialect.Oracle10gDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
DB2	org.hibernate.dialect.DB2Dialect

Hibernate Configuration

- As Hibernate can operate in different environments, it requires a wide range of configuration parameters. These configurations contain the mapping information that provides different functionalities to Java classes. Generally, we provide database related mappings in the configuration file. Hibernate facilitates to provide the configurations either in an XML file (like hibernate.cfg.xml) or properties file (like hibernate.properties).
- An instance of Configuration class allows specifying properties and mappings to applications. This class also builds an immutable **SessionFactory**.

- Hibernate JDBC Properties:

Property	Description
hibernate.connection.driver_class	It represents the JDBC driver class.
hibernate.connection.url	It represents the JDBC URL.
hibernate.connection.username	It represents the database username.
hibernate.connection.password	It represents the database password.
Hibernate.connection.pool_size	It represents the maximum number of connections available in the connection pool.

Hibernate Configuration

- Hibernate DataSource Properties:

Property	Description
hibernate.connection.datasource	It represents datasource JNDI name which is used by Hibernate for database properties.
hibernate.jndi.url	It is optional. It represents the URL of the JNDI provider.
hibernate.jndi.class	It is optional. It represents the class of the JNDI InitialContextFactory.

Hibernate Configuration

- Hibernate Configuration Properties:

Property	Description
hibernate.dialect	It represents the type of database used in hibernate to generate SQL statements for a particular relational database.
hibernate.show_sql	It is used to display the executed SQL statements to console.
hibernate.format_sql	It is used to print the SQL in the log and console.
hibernate.default_catalog	It qualifies unqualified table names with the given catalog in generated SQL.
hibernate.default_schema	It qualifies unqualified table names with the given schema in generated SQL.
hibernate.session_factory_name	The SessionFactory interface automatically bound to this name in JNDI after it has been created.
hibernate.default_entity_mode	It sets a default mode for entity representation for all sessions opened from this SessionFactory
hibernate.order_updates	It orders SQL updates on the basis of the updated primary key.
hibernate.use_identifier_rollback	If enabled, the generated identifier properties will be reset to default values when objects are deleted.
hibernate.generate_statistics	If enabled, the Hibernate will collect statistics useful for performance tuning.
hibernate.use_sql_comments	If enabled, the Hibernate generate comments inside the SQL. It is used to make debugging easier.

Hibernate Configuration

- Hibernate Cache Properties:

Property	Description
hibernate.cache.provider_class	It represents the classname of a custom CacheProvider.
hibernate.cache.use_minimal_puts	It is used to optimize the second-level cache. It minimizes writes, at the cost of more frequent reads.
hibernate.cache.use_query_cache	It is used to enable the query cache.
hibernate.cache.use_second_level_cache	It is used to disable the second-level cache, which is enabled by default for classes which specify a mapping.
hibernate.cache.query_cache_factory	It represents the classname of a custom QueryCache interface.
hibernate.cache.region_prefix	It specifies the prefix which is used for second-level cache region names.
hibernate.cache.use_structured_entries	It facilitates Hibernate to store data in the second-level cache in a more human-friendly format.

Hibernate Configuration

- Hibernate Cache Properties:

Property	Description
hibernate.transaction.factory_class	It represents the classname of a TransactionFactory which is used with Hibernate Transaction API.
hibernate.transaction.manager_lookup_class	It represents the classname of a TransactionManagerLookup. It is required when JVM-level caching is enabled.
hibernate.transaction.flush_before_completion	If it is enabled, the session will be automatically flushed during the before completion phase of the transaction.
hibernate.transaction.auto_close_session	If it is enabled, the session will be automatically closed during the after completion phase of the transaction.

Hibernate Configuration

- Hibernate Other Properties:

Property	Description
hibernate.connection.provider_class	It represents the classname of a custom ConnectionProvider which provides JDBC connections to Hibernate.
hibernate.connection.isolation	It is used to set the JDBC transaction isolation level.
hibernate.connection.autocommit	It enables auto-commit for JDBC pooled connections. However, it is not recommended.
hibernate.connection.release_mode	It specifies when Hibernate should release JDBC connections.
hibernate.current_session_context_class	It provides a custom strategy for the scoping of the "current" Session.
hibernate.hbm2ddl.auto	It automatically generates a schema in the database with the creation of SessionFactory.

Hibernate Named Query

The hibernate named query is way to use any query by some meaningful name. It is like using alias names. The Hibernate framework provides the concept of named queries so that application programmer need not to scatter queries to all the java code.

- **@NamedQueries** annotation is used to define the multiple named queries.
- **@NamedQuery** annotation is used to define the single named query.

Example:

```
@NamedQueries(
    {
        @NamedQuery(
        name = "findEmployeeByMobileNo",
        query = "from Employee e where e.mobileNo = :mobileNo"
        )
    }
)
```

Hibernate Native Query

Hibernate provide option to execute native SQL queries through the use of SQLQuery object. Hibernate SQL Query is very handy when we have to execute database vendor specific queries that are not supported by Hibernate API. For example query hints or the CONNECT keyword in Oracle Database.

For Hibernate Native SQL Query, we use **Session.createSQLQuery(String query)** to create the SQLQuery object and execute it. For example, if you want to read all the records from Employee table, we can do it through below code.

```
// Example
SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
Session session = sessionFactory.getCurrentSession();
// Get All Employees
Transaction tx = session.beginTransaction();
SQLQuery query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee"); List<Object[]>
rows = query.list();
for(Object[] row : rows){
 Employee emp = new Employee();
 emp.setId(Long.parseLong(row[0].toString()));
 emp.setName(row[1].toString());
 emp.setSalary(Double.parseDouble(row[2].toString()));
 System.out.println(emp); }
```

Hibernate Native Query

Hibernate uses **ResultSetMetadata** to deduce the type of the columns returned by the query, from performance point of view we can use **addScalar()** method to define the data type of the column. However we would still get the data in form of Object array.

```
//Get All Employees - addScalar example
query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee")
 .addScalar("emp_id", new LongType()) .addScalar("emp_name", new StringType())
 .addScalar("emp_salary", new DoubleType());
 rows = query.list();
 for(Object[] row : rows){
 Employee emp = new Employee();
 emp.setId(Long.parseLong(row[0].toString()));
 emp.setName(row[1].toString());
 emp.setSalary(Double.parseDouble(row[2].toString()));
 System.out.println(emp);
}
```

Hibernate Native Query

Hibernate Native SQL Multiple Tables

If we would like to get data from both Employee and Address tables, we can simply write the SQL query for that and parse the result set.

```
query = session.createSQLQuery("select e.emp_id, emp_name, emp_salary,address_line1, city,
        zipcode from Employee e, Address a where a.emp_id=e.emp_id");
rows = query.list();
for(Object[] row : rows){
        Employee emp = new Employee();
        emp.setId(Long.parseLong(row[0].toString()));
        emp.setName(row[1].toString());
        emp.setSalary(Double.parseDouble(row[2].toString()));
        Address address = new Address();
        address.setAddressLine1(row[3].toString());
        address.setCity(row[4].toString());
        address.setZipcode(row[5].toString());
        emp.setAddress(address);
        System.out.println(emp);
}
```

Hibernate Native Query

Hibernate Native SQL Entity and Join

We can also use addEntity() and addJoin() methods to fetch the data from associated table using tables join. For example, above data can also be retrieved as below.

```
//Join example with addEntity and addJoin
query = session.createSQLQuery("select {e.*}, {a.*} from Employee e join Address a ON
e.emp_id=a.emp_id")
                .addEntity("e",Employee.class)
                .addJoin("a","e.address");
rows = query.list();
for (Object[] row : rows) {
    for(Object obj : row) {
        System.out.print(obj + "::");
    }
    System.out.println("\n");
}
//Above join returns both Employee and Address Objects in the array
for (Object[] row : rows) {
        Employee e = (Employee) row[0];
        System.out.println("Employee Info::"+e);
        Address a = (Address) row[1];
        System.out.println("Address Info::"+a);
}
```

Hibernate Native Query

Hibernate Native SQL Query with Parameters

We can also pass parameters to the Hibernate SQL queries, just like JDBC PreparedStatement. The parameters can be set using the name as well as index, as shown in below example.

```
query = session
                .createSQLQuery("select  emp_id,  emp_name,  emp_salary  from  Employee  where
emp_id = ?");
List<Object[]> empData = query.setLong(0, 1L).list();
for (Object[] row : empData) {
        Employee emp = new Employee();
        emp.setId(Long.parseLong(row[0].toString()));
        emp.setName(row[1].toString());
        emp.setSalary(Double.parseDouble(row[2].toString()));
        System.out.println(emp);
}
query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee where emp_id =
:id");
empData = query.setLong("id", 2L).list();
for (Object[] row : empData) {
        Employee emp = new Employee();
        emp.setId(Long.parseLong(row[0].toString()));
        emp.setName(row[1].toString());
        emp.setSalary(Double.parseDouble(row[2].toString()));
        System.out.println(emp);
```

Hibernate Get and Load

Hibernate Session provide different methods to fetch data from database. Two of them are – **get()** and **load()**. There are also a lot of overloaded methods for these, that we can use in different circumstances**.**

get() returns the object by fetching it from database or from hibernate cache whereas load() just returns the reference of an object that might not actually exists, it loads the data from database or cache only when you access other properties of the object.

Differences:

- get() loads the data as soon as it's called whereas load() returns a proxy object and loads data only when it's actually required, so load() is better because it support lazy loading.
- Since load() throws exception when data is not found, we should use it only when we know data exists.
- We should use get() when we want to make sure data exists in the database

Hibernate

One to One mapping:

```
@Entity
public class Library {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;

@OneToOne
private Student stud;
```

One to Many mapping:

```
@Entity
public class Student {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private int s_id;
   private String s_name;

   @OneToMany(targetEntity=Library.class)
   private List books_issued;
```

Hibernate

Many to One mapping:

```
@Entity
public class Student {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private int s_id;
   private String s_name;

   @ManyToOne
   private Library lib;
```

Many to Many mapping:

```
@Entity
public class Student {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private int s_id;
   private String s_name;

   @ManyToMany(targetEntity=Library.class)
   private List lib;
```

Hibernate Criteria Api

There are three way to pulling data from the database in the Hibernate:

1. **Using session methods**(get() and load() methods) -limited control to accessing data
2. **Using HQL** – Slightly more control using where clause and other clauses but there are some problems here… is more complicated to maintain in case of bigger queries with lots of clauses.
3. **Using Criteria API**

Criteria Api:

- The API (Application Programming Interface) of Hibernate Criteria provides an elegant way of building dynamic query on the persistence database.
- The hibernate criteria API is very Simplified API for fetching data from Criterion objects. The criteria API is an alternative of HQL (Hibernate Query Language) queries. It is more powerful and flexible for writing tricky criteria functions and dynamic queries

Example:
```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Long> criteriaQuery = builder.createQuery(Long.class);
```

Hibernate Criteria Api

The Criteria API Example:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
        // Count number of employees
        CriteriaQuery<Long> criteriaQuery = builder.createQuery(Long.class);
        Root<Employee> root = criteriaQuery.from(Employee.class);
        criteriaQuery.select(builder.count(root));
        Query<Long> query = session.createQuery(criteriaQuery);
        long count = query.getSingleResult();
        System.out.println("Count = " + count);
```

Criteria ordering another example:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
// mobileNo employees
        CriteriaQuery<Employee> criteriaQuery2 = builder.createQuery(Employee.class);
        Root<Employee> root2 = criteriaQuery2.from(Employee.class);
        Predicate predicateForMobile = builder.equal(root2.get("mobileNo"), "234243534");
        criteriaQuery2.where(predicateForMobile);
        List<Employee> employees = session.createQuery(criteriaQuery2).getResultList();
        for(Employee emp: employees) {
        System.out.println(emp);
        }
```

Hibernate Criteria Api

Problem with Criteria API:

1. Performance issue

You have no way to control the SQL query generated by Hibernate, if the generated query is slow, you are very hard to tune the query, and your database administrator may not like it.

2. Maintenance issue

All the SQL queries are scattered through the Java code, when a query went wrong, you may spend time to find the problem query in your application. On the others hand, named queries stored in the Hibernate mapping files are much more easier to maintain.