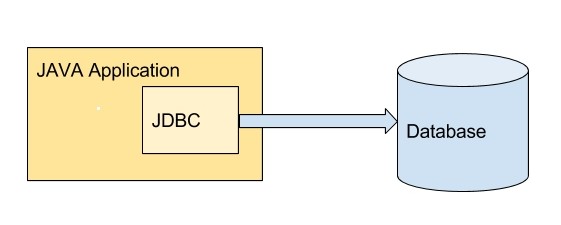
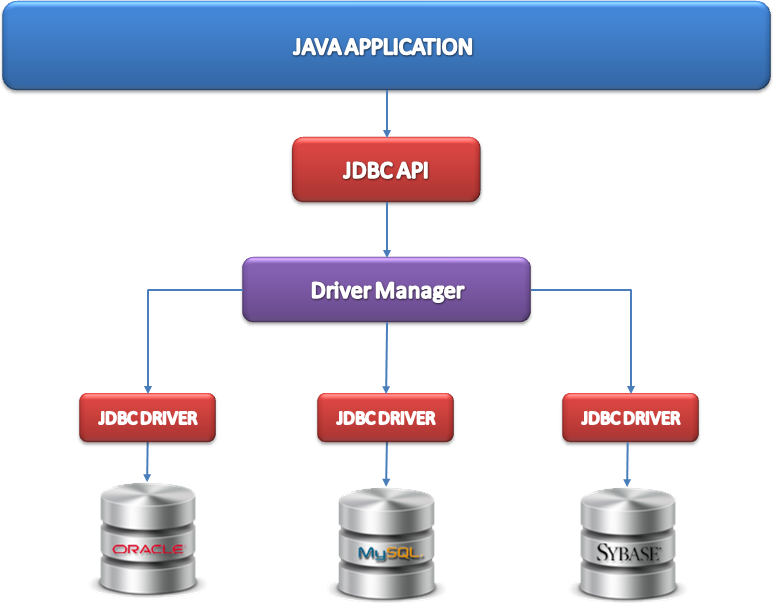
**What Is JDBC?**



JDBC stands for **Java Database Connectivity**. It is a free open source application programming interface for Java that enables applications to access databases. It enables developers to create queries and update data to a relational database using the **Structured Query Language** (SQL).

**JDBC Workflow**

****

1. Open a database connection
2. Send SQL queries to database using a JDBC driver
3. JDBC driver connects to the database
4. Execute the queries to get the result set
5. Send the data to the application via the driver manager
6. When results are returned, it processes the data
7. Finally, the connection is closed

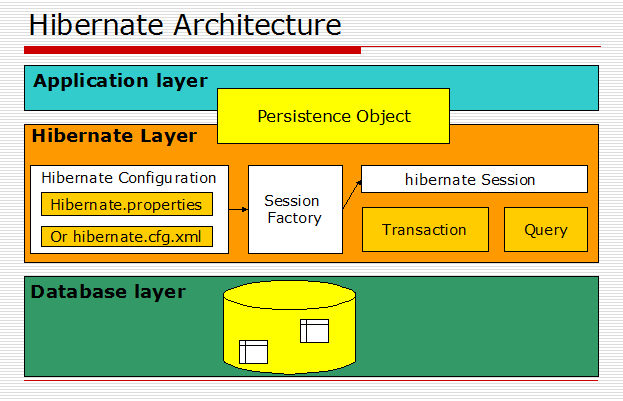
**What Is Hibernate?**

****

*Source:*[*J2eebrain*](http://www.j2eebrain.com/java-J2ee-hibernate-object-relational-mapping.html)

Hibernate is a free, open source object-relational mapping library for Java designed to map objects to an RDBMS and to implement the object-oriented programming concepts in a relational database.

**Hibernate Workflow**

****

1. Unlike JDBC, Hibernate connects with the database itself and uses HQL (Hibernate Query Language) to execute the queries, then maps the results to Java objects.
2. The results are mapped to objects based on the properties given in the Hibernate configuration XML file.
3. The database connection from an application is created using the session, which also helps in saving and retrieving the persistent object.
4. Session factory is an interface that helps to create an instance of a session. There must be only one session factory per database. For example, if you are using MySQL and Oracle in your application, one session factory for MySQL and one session factory for Oracle is maintained. There will not be more than one session factory for MySQL alone.

So, without further ado, let's take a look at all the reasons why you should consider Hibernate over JDBC.

**Impedance Mismatch**

Hibernate mainly solves the **object-relational impedance mismatch** problems that arise when a relational database is connected by an application written in an object-oriented programming language style. Object-relational impedance mismatches arise due to data type differences, manipulative differences, transactional differences, and structural and integrity differences. Click [here](http://hibernate.org/orm/what-is-an-orm/) to see more details.

**Object Mapping**

In JDBC, you need to write code to map the object model’s data representation to a relational model and its corresponding schema. Hibernate itself **maps Java classes to database tables** using XML or by using annotations. The following example will help you understand how **object mapping** is done in Hibernate

//JDBC

List<User> users=new ArrayList<User>();

while(rs.next()) {

User user = new User();

user.setUserId(rs.getString("UserId"));

user.setName(rs.getString("FirstName"));

user.setEmail(rs.getString(“Email”));

users.add(user);

}

//Hibernate

@Entity

@Table(name = "user")

public class UserModel {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private BigInteger id;

@NotEmpty

@Column(name = "email", unique = true)

private String email;

@NotEmpty

@Column(name = "name")

private String name;

public BigInteger getId() {

return this.id;

}

public void setId(BigInteger id) {

this.id = id;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

}

In the above example, you can see that by using JDBC, you need to set every property of an object upon fetching the data each and every time. But in Hibernate, we need to map the table with the Java class as mentioned above.

**HQL**

Hibernate uses HQL (Hibernate Query Language), which is similar to SQL, but Hibernate’s HQL provides full support for **polymorphic queries**. HQL understands object-oriented concepts like **inheritance, polymorphism, and association**. For a detailed understanding of polymorphic HQL queries, refer to this [link.](http://www.roseindia.net/hibernate/hibernate4/hibernatePolymorphicQueries.shtml)

**Database Independent**

Hibernate’s code is **database independent** because you do not need to change the HQL queries (with a few exceptions) when you change databases like MySQL, Oracle, etc. Hence, it is **easy to migrate** to a new database. It is achieved by using a friendly dialect to communicate with the database. The database can be specified using a dialect in the Hibernate configuration XML as follows.

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

For example, consider the following scenario. You need to fetch the first 10 entries of a table. How this is implemented in different databases is explained below:

#MySQL

SELECT column\_name FROM table\_name ORDER BY column\_name ASC LIMIT 10;

#SQL Server

SELECT TOP 10 column\_name FROM table\_name ORDER BY column\_name ASC;

In Hibernate, this can be done as follows

Session.CreateQuery("SELECT E.id FROM Employee E ORDER BY E.id ASC").SetMaxResults(10).List();

Thus your query statement need not change, irrespective of the database you are using.

**Minimize Code Changes**

Hibernate **minimizes code changes** when we add a new column to a database table.

**JDBC**

* You have to add a new field in your POJO class.
* Change your JDBC method that performs the “select” to include the new column.
* Change your JDBC method that performs the “insert” to add a new value into the new column.
* Change your JDBC method that performs the “update” to update an existing value in your new column.

**Hibernate**

* Add the new field into your POJO class.
* Modify the Hibernate XML mapping file to include the new column.

Thus, database table changes can be easily implemented using Hibernate with less effort.

**Reduce Repeat Code**

Hibernate **reduces the amount of** repeating lines of **code**, which you can often find with JDBC. For your understanding, I have left a simple scenario below.

Class.forName("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection( "jdbc:mysql://localhost:3306/sonoo",

"root","root");

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");

stmt.setInt(1,101);

stmt.setString(2,"Ratan");

stmt.executeUpdate();

con.close();

In JDBC, as mentioned above for inserting a record, you will be creating a prepared statement and will be setting each column of a table. If the number of columns increases, the statements will also increase. But in Hibernate, we just need to save the object with persist(Object);.

**Lazy Loading**

We can achieve **lazy loading** using Hibernate. Consider an example where there is a list of users in the user table. The identity proof documents uploaded by the users are stored in the identity\_proof table. The user has a ‘one to many’ relationships with the identity\_proof. In this case, the user is the parent class and identity\_proof is the child class. If you fetch the parent class, i.e the user, all the documents associated with the user will also be fetched. Imagine the size of each document. As the number of documents increases, the size of data to be processed also increases, and hence it will slow up the application.

With Hibernate, you can specify the fetch type for data as LAZY. If you do so when you fetch a user, documents will not be fetched. You can fetch the documents where you want using Hibernate’s initialize() method.

// Declaring fetch type for one to many association in your POJO

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private Set<Proof> ProofList = new HashSet<Proof>();

// To fetch user with document use initialize() method as follows

getHibernateTemplate().initialize(user.getProofList());

**Avoiding Try-Catch Blocks**

JDBC will throw SQLexception, which is a checked exception. So you will be writing “try-catch” blocks in your code. Hibernate handles this by converting all **JDBC exceptions** to **unchecked** exceptions. Therefore, you need not waste your time implementing try-catch blocks.

**Transaction Management**

A [transaction](http://www.sitesbay.com/jdbc/jdbc-transaction-management) is a group of operations to be performed under one task. If all the operations in a group succeed, then the task is finished and the transaction is successfully completed. If one of the operations fails, then the whole task fails, and thus the transaction also fails.**.**In JDBC, if a transaction is a success you need to commit. Otherwise, you need to perform a rollback with the following commands:

* con.commit();
* con.rollback();

In Hibernate, you don’t have to commit and roll back these transactions, as it is implicitly provided.

**Associations**

It is easy to create an **association** between tables using Hibernate. Associations like one-to-one, one-to-many, many-to-one, and many-to-many can be achieved easily in Hibernate by using annotations, mapping the object entity of the required table. To read more about these associations, kindly go through this [link](https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/associations.html). An example for creating association is given below:

@Entity

@Table(name = "college")

public class College {

@Id

@GeneratedValue

private Integer id;

@Column(name="college\_name")

private String name;

@OneToMany(mappedBy="college", cascade=CascadeType.ALL)

private Set<Department> departments;

}

@Entity

@Table(name = "department")

public class Department {

@Id

@GeneratedValue

private Integer id;

@Column(name="department\_name")

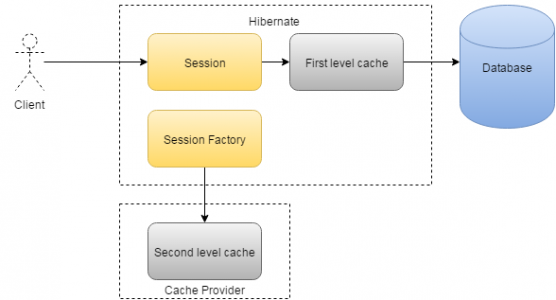
private String departmentname;

@ManyToOne

@JoinColumn(name = "college\_id")

private College college;

}

**Hibernate Caching**

Hibernate provides a **caching mechanism**, which helps reduce the number of hits, as much as possible, that your application makes to the database server. This will have a considerable effect regarding the performance of your application. There is no such caching mechanism available in JDBC. This is because Hibernate stores the object in session, which is available until the transaction is active. When a particular query is executed repeatedly, the value stored in the session is used. When a new transaction begins, the object is fetched again from the database and is stored in the session. In fact, two levels of caches are provided by Hibernate. To learn more about Hibernate caching, refer to this [link](http://way2java.com/hibernate/hibernate-first-level-and-second-level-cache-examples/).

**Versioning**

[Database versioning](http://www.java4s.com/hibernate/hibernate-versioning-example-hibernate-versioning-of-objects/) is an important feature of Hibernate. Hibernate enables developers to define version type fields in an application, which is updated when data is updated every time. The advantage is if two different users retrieve the same data and then modify it, and one user saved their modified data to the database before the other user, the version is updated. Now when the other user tries to saves their data, it will not allow it because this data is not up to date. In JDBC, this check is performed by the developer.

**Audit Functionality**

Hibernate provides a library called [**“Envers”**](http://hibernate.org/orm/envers/),which enables us to easily gain audit functionality. It creates tables and stores the audited data with version number from where we can track the changes made to the entity. Let me explain how auditing is done using Envers.

@Entity

@Table(name = "User")

@Audited

public class User {

// ... the body of the entity

}

When we add the @Audited annotation in our entity class, Hibernate will create a table called User\_AUD to store the audited entries. The table will contain two additional fields — REVTYPE and REV. REVTYPE is the type of the revision. It can store the values as add for inserting, mod for modifying, and del for deleting. The REV field contains the revision number of the stored entry. Hibernate creates an extra table called REVINFO in which the REV number is mapped with a timestamp when the change of the entity happened. You can also customize the revision table using @RevisionEntity.

**JPA Annotation Support**

Hibernate provides **support to JPA annotations** like @Entity, @Table, @Column, etc. These annotations make the code portable to other ORM frameworks. You can find some of the annotations explained with an example in this [link](http://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/).

**Connection Pooling**

As explained on [Wikipedia](https://en.wikipedia.org/wiki/Connection_pool), connection pooling is a mechanism in which the database connections, when created are stored in the cache by an external tool so that these connections can be reused from the cache when the application tries to connect to the same database in future. Connection pooling helps increase performance. We can achieve this connection pooling in Hibernate. Hibernate supports the following pools:

* [C3p0](http://sourceforge.net/projects/c3p0)
* [Apache DBCP](http://jakarta.apache.org/commons/dbcp/)
* [Proxool](http://proxool.sourceforge.net/)

You need to set the properties in Hibernate.cfg.xml as required to enable connection pooling. A sample property is shown below.

<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>