OBJECT™ TECHNOLOGIES

Using Hibernate

What Will Be Covered

OBJECT

- Using hibernate framework in application
- Writing configuration file
- Specifying mapping information
- Introduction to JPA
- Mapping annotations
- Bootstrapping the application
- Simple save opearation

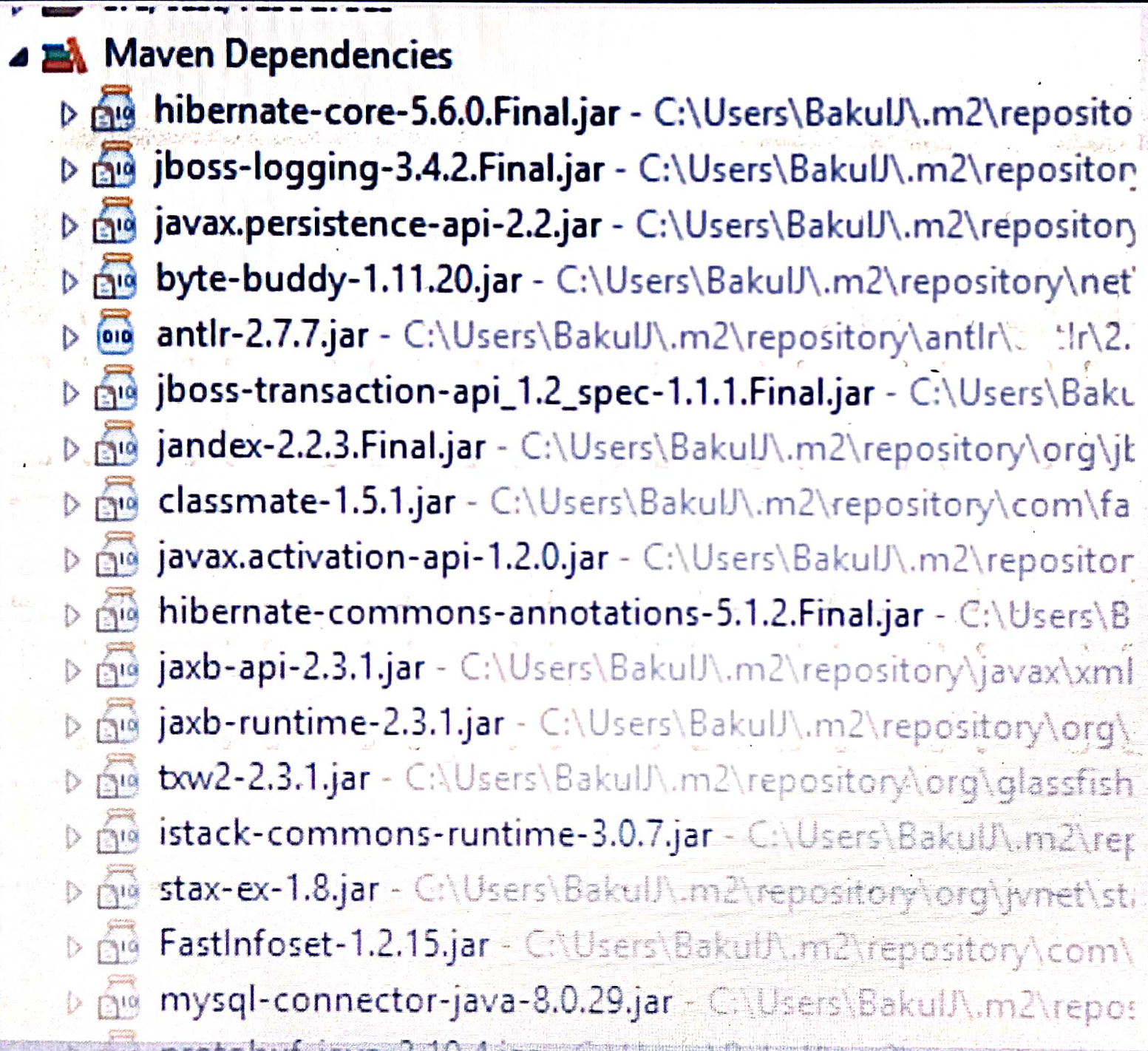
Using hibernate framework in application

OBJECT TECHNOLOGIES

- Framework software will be in the form of a set of jar files, where one jar file acts as main (We can call this file as core) and remaining will acts as dependent jar files.
- Each Framework software contain at least one configuration xml file, but multiple configuration files also allowed.
- In this case, in order to setup the Hibernate framework environment into a java application, set of jar files are required because of transitive dependenies
- Along with the hibernate jars we must include one more jar file, which is nothing but related to our database, (this is depending on your database) is needed

Using hibernate framework in application

OBJECT TECHNOLOGIES



Using hibernate framework in application

OBJECT TECHNOLOGIES

- These jar files can be physically downloaded
- And they can be made available to java application using classpath for that application.
- Alternative way is to use some build tool like maven which allows to add jar files using pom.xml file

```
<dependencies>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.0.Final</version>

    </dependency>

    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
</dependencies>
```

Writing configuration file

OBJECT TECHNOLOGIES

```
<hibernate-configuration>
    <session-factory>
        <!-- SQL Dialect - type of database -->
        <property name="hibernate.dialect">
org.hibernate.dialect.MySQL5Dialect</property>

        <!-- Database Connection Settings -->
        <property name="hibernate.connection.driver_class">
com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">
jdbc:mysql://localhost:3306/test?serverTimezone=UTC</property>
        <property name="hibernate.connection.username">
root</property>
        <property name="hibernate.connection.password">
root</property>

        <!--  optionally specified -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
update</property>

        <!-- Specifying Session Context -->
        <property
name="hibernate.current_session_context_class">
org.hibernate.context.internal.ThreadLocalSessionContext</proper
ty>

        <!-- Mapping With Model Class Containing Annotations -->
```

Writing configuration file

OBJECT TECHNOLOGIES

- Hibernate 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.
- Conventionally, name of xml file should be hibernate.cfg.xml
- But it can be different one and this file should be available in application's classpath.

Specifying mapping information

OBJECT TECHNOLOGIES

- The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.
- All these persistent classes follow the Plain Old Java Object (POJO) / JavaBean programming model.
- A simple persistent class should follow some rules:
  - **A no-arg constructor**: It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by the newInstance() method.
  - **Provide an identifier property**: It is better to assign an attribute as id. This attribute behaves as a primary key in a database.
  - **Declare getter and setter methods**: The Hibernate recognizes the method by getter and setter method names by default.
  - **Prefer non-final class**: Hibernate uses the concept of proxies, which depends on the persistent class. The application programmer will not be able to use proxies for lazy association fetching.

Specifying mapping information

OBJECT

- *Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose. With the coming of Jakarta Persistence, most of this information is now defined in a way that is portable across ORM/Jakarta Persistence providers using annotations (and/or standardized XML format).*
- *So initially for every POJO class, one mapping XML file is required for specifying relation of class with database table and attribute with table columns.*
- *Now a days mapping an*notations are preferred which are specified in Java Persistent API

Introduction to JPA

OBJECT

- As a specification, the Jakarta Persistence API (**formerly Java Persistence API**) is concerned with persistence, which loosely means any mechanism by which Java objects outlive the application process that created them.
- The JPA specification lets you define which objects should be persisted, and how they are persisted in your Java applications.
- By itself, JPA is not a tool or framework; rather, it defines a set of concepts that guide implementers.
- Some JPA implementations have been extended for use with NoSQL datastores.
- JPA's object-relational mapping (ORM) model was originally based on Hibernate.
- JPA lets you avoid the need to “think relationally." In JPA, you define your persistence rules in the realm of Java code and objects
- Because of their intertwined history, Hibernate and JPA are frequently conflated.

Mapping annotations in JPA

OBJECT TECHNOLOGIES

- JPA entities don't need to implement any interface or extend a superclass. They are simple POJOs.
- But you still need to identify a class as an entity class, and you might want to adapt the default table mapping.
- **@Entity**
- The JPA specification requires the @Entity annotation. It identifies a class as an entity class.

```
@Entity
public class Author { ... }
```

- **@Table**
- By default, each entity class maps a database table with the same name in the default schema of your database. You can customize this mapping using the name, schema, and catalog attributes of the @Table annotation.

```
@Entity
@Table(name = "AUTHORS", schema = "STORE")
public class Author {
```

Mapping annotations in JPA

- **Basic Column Mappings**
- By default, all JPA implementations map each entity attribute to a database column with the same name and a compatible type.
- Following annotations enable you to perform basic customizations like change the name of the column, adapt the type mapping, identify primary key attributes, and generate unique values for them.
- **@Column**
- It is an optional annotation that enables you to customize the mapping between the entity attribute and the database column.

```
@Entity
public class Book {

    @Column(name = "title", updatable = false, insertable = true)
    private String title;
```

Mapping annotations in JPA

OBJECT TECHNOLOGIES

- **Basic Column Mappings**
- By default, all JPA implementations map each entity attribute to a database column with the same name and a compatible type.
- Following annotations enable you to perform basic customizations like change the name of the column, adapt the type mapping, identify primary key attributes, and generate unique values for them.
- **@Column**
- It is an optional annotation that enables you to customize the mapping between the entity attribute and the database column.

```
@Entity
public class Book {

    @Column(name = "title", updatable = false, insertable = true)
    private String title;
```

Mapping annotations in JPA

OBJECT TECHNOLOGIES

- **@Id**
- JPA and Hibernate require you to specify at least one primary key attribute for each entity with @Id annotation

```
@Entity
public class Author {

    @Id
    private Long id;
```

- **@GeneratedValue**
- This annotation enables you to reference a custom generator. Database sequence can be used by setting the strategy attribute to GenerationType.SEQUENCE. For auto-incremented database column to generate primary key values, use the strategy to GenerationType.IDENTITY.

```
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
```

Mapping annotations in JPA

OBJECT TECHNOLOGIES

- **@Enumerated**
- The @Enumerated annotation enables you to define how an enum attribute gets persisted in the database.
- **@Temporal**
- Using this annotation, you can define if the attribute shall be mapped as an SQL DATE, TIME, or TIMESTAMP.
- **@Lob**
- Using JPA's @Lob annotation, you can map a BLOB to a byte[] and a CLOB to a String.

Bootstrapping the application

OBJECT TECHNOLOGIES

- The term bootstrapping refers to initializing and starting a software component. In Hibernate, we are specifically talking about the process of building a fully functional SessionFactory instance
- Generally, the following classes are used as per the requirements for building SessionFactory.
  - **ServiceRegistry**: defines service registry contracts that applications are likely to want to utilize for configuring Hibernate behavior. The two popular implementations are BootstrapServiceRegistry, StandardServiceRegistry and SessionFactoryServiceRegistry.
  - **StandardServiceRegistry**: hosts and manages services in runtime.
  - **MetaData**: an object containing the parsed representations of an application domain model and its mapping to a database.
  - **MetadataSources**: provides the source information to be parsed to form MetaData.

Bootstrapping the application

OBJECT TECHNOLOGIES

```
public class HibernateUtil {
  private static SessionFactory sessionFactory = buildSessionFactory();

  private static SessionFactory buildSessionFactory() {
    try {
      if (sessionFactory == null) {
        StandardServiceRegistry standardRegistry
            = new StandardServiceRegistryBuilder()
              .configure()
              .build();

        Metadata metadata = new MetadataSources(standardRegistry)
            .getMetadataBuilder()
            .build();

        sessionFactory = metadata.getSessionFactoryBuilder().build();
      }
      return sessionFactory;
    } catch (Throwable ex) {
      throw new ExceptionInInitializerError(ex);
    }
  }
```

Simple save opearation

OBJECT TECHNOLOGIES

- Sessionfactory will create and manage the sessions.
- Sessionfactory is a heavy weight object, because it maintains datasources, mappings, hibernate configuration information's etc.
- Sessionfactory is an immutable object and it will be created as singleton.
- Sessions will be opened using sessionfactory.openSession() and some database operations will be done finally
- Session is one instance per client/thread/one transaction and it is light weight and not thread safe
- A transaction is associated with a Session and is usually instantiated by a call to Session.beginTransaction(). A single session might span multiple transactions and transaction provides methods for **commit and rollback.**

Simple save opearation

OBJECT TECHNOLOGIES

```
Emp e = new Emp(120,"Raju",34500.00f,"Admin");

Session session = sf.openSession();

Transaction tr = session.beginTransaction();
//object e to be saved as record
session.save(e);

tr.commit();

System.out.println("Emp saved");

session.close();
```

```
@Entity
@Table(name="emp")
public class Emp
{
    @Id
    int empid;
    @Column(name="empname")
    String ename;
    @Column
    float salary;
    @Column
    String dept;
```

Using hibernate

In this chpater we will learn about what are different steps that should be taken to make hibernate tool to perform database operations for java application.

Using hibernate framework

Framework software will be in the form of a set of jar files, where one jar file acts as main (We can call this file as core) and remaining will acts as dependent jar files.

Each Framework software contain at least one configuration xml file, but multiple configuration files also allowed.

In this case, in order to setup the Hibernate framework environment into a java application, set of jar files are required because of transitive dependenies

Along with the hibernate jars we must include one more jar file, which is nothing but related to our database, (this is depending on your database) is needed

These jar files can be downloaded and linked in the existing application environment in the classpath.

This even can be done with the help of any build tool like Maven. For Maven, add following entry in pom.xml

```
<dependencies>
 <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.0.Final</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
  </dependency>
</dependencies>
```

Note that dependency for database jar file should be separately added.

Hibernate configuration file

One of the most required configuration file in Hibernate is hibernate.cfg.xml file. By default, it is placed under src/main/resource folder. hibernate.cfg.xml file contains database related configurations and session related configurations.

Database configuration includes jdbc connection url, DB user credentials, driver class and hibernate dialect

Hibernate configuration file

One of the most required configuration file in Hibernate is hibernate.cfg.xml file. By default, it is placed under src/main/resource folder. hibernate.cfg.xml file contains database related configurations and session related configurations. Database configuration includes jdbc connection url, DB user credentials, driver class and hibernate dialect.

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
       <property name="hbm2ddl.auto">update</property>
       <property name="dialect"> org.hibernate.dialect.MySQL5Dialect</property>
       <property name="connection.url"> jdbc:mysql://localhost:3306/test</property>
       <property name="connection.username">system</property>
       <property name="connection.password">jtp</property>
       <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
    </session-factory>
</hibernate-configuration>
```

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 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 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 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 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.

Specifying mapping information

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

All these persistent classes follow the Plain Old Java Object (POJO) / JavaBean programming model.

A simple persistent class should follow some rules:

Specifying mapping information

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

All these persistent classes follow the Plain Old Java Object (POJO) / JavaBean programming model.

A simple persistent class should follow some rules:

1. A no-arg constructor: It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by the newInstance() method.
2. Provide an identifier property: It is better to assign an attribute as id. This attribute behaves as a primary key in a database.
3. Declare getter and setter methods: The Hibernate recognizes the method by getter and setter method names by default.
4. Prefer non-final class: Hibernate uses the concept of proxies, which depends on the persistent class. The application programmer will not be able to use proxies for lazy association fetching.

Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose. With the coming of Jakarta Persistence, most of this information is now defined in a way that is portable across ORM/Jakarta Persistence providers using annotations (and/or standardized XML format).

So initially for every POJO class, one mapping XML file is required for specifying relation of class with database table and attribute with table columns.

Now a days mapping annotations are preferred which are specified in Jakarta Persistent API formerly known as Java Persistent API

Introduction to JPA

- By itself, JPA is not a tool or framework; rather, it defines a set of concepts that guide implementers.
- JPA is a specification, the Jakarta Persistence API (formerly Java Persistence API) is concerned with persistence, which loosely means any mechanism by which Java objects outlive the application process that created them.
- Basically it provides standardization for all ORMapping tools

Following is the list very basic annotations required for mapping an entity with table and attributes with columns.

Annotation	Purpose
@Entity	It identifies a class as an entity class.
@Table	Specifies mapping class with a table in mentioned schema
@Id	Specify at least one primary key attribute for each entity

Introduction to JPA

- By itself, JPA is not a tool or framework; rather, it defines a set of concepts that guide implementers.
- JPA is a specification, the Jakarta Persistence API (formerly Java Persistence API) is concerned with persistence, which loosely means any mechanism by which Java objects outlive the application process that created them.
- Basically it provides standardization for all ORMapping tools

Following is the list very basic annotations required for mapping an entity with table and attributes with columns.

Annotation	Purpose
@Entity	It identifies a class as an entity class.
@Table	Specifies mapping class with a table in mentioned schema
@Id	Specify at least one primary key attribute for each entity
@GeneratedValue	Enables you to reference a custom generator.
@Column	Customize the mapping between the entity attribute and the database column.

Hibernate components

SessionFactory (org.hibernate.SessionFactory)

A thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for org.hibernate.Session instances. The EntityManagerFactory is the JPA equivalent of a SessionFactory and basically, those two converge into the same SessionFactory implementation.

A SessionFactory is very expensive to create, so, for any given database, the application should have only one associated SessionFactory. The SessionFactory maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations, etc.

Session (org.hibernate.Session)

Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries.

Session (org.hibernate.Session)

Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the ··· 'erlying transaction system in use (JDBC or JTA).

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.

Save operation

Sessionfactory will create and manage the sessions. It is a heavy weight object, because it maintains datasources, mappings, hibernate configuration information's etc. Sessionfactory is an immutable object and it will be created as singleton.
Sessions will be opened using sessionfactory.openSession() and some database operations will be done finally
Session is one instance per client/thread/one transaction and it is light weight and not thread safe.
A transaction is associated with a Session and is usually instantiated by a call to Session.beginTransaction(). A single session might span multiple transactions and transaction provides methods for commit and rollback.

```
//creating session factory
//opening some services
StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure().build();
//read info from xml file
```

Save operation

Sessionfactory will create and manage the sessions. It is a heavy weight object, because it maintains datasources, mappings, hibernate configuration information's etc. Sessionfactory is an immutable object and it will be created as singleton.

Sessions will be opened using sessionfactory.openSession() and some database operations will be done finally

Session is one instance per client/thread/one transaction and it is light weight and not thread safe.

A transaction is associated with a Session and is usually instantiated by a call to Session.beginTransaction(). A single session might span multiple transactions and transaction provides methods for commit and rollback.

```
//creating session factory
//opening some services
StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure().build();
//read info from xml file
Metadata metadata = new MetadataSources(registry).getMetadataBuilder().build();
//establish connection to database
SessionFactory sf = metadata.getSessionFactoryBuilder().build();
```

Following code creates session and saves the Emp object as a record in the database table using transaction.

```
Emp e = new Emp(120,"Raju",34500.00f,"Admin");
Session session = sf.openSession();
Transaction tr = session.beginTransaction();
//object e to be saved as record
session.save(e);
tr.commit();
System.out.println("Emp saved");
session.close();
sf.close();
```

Update and delete operation

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. These methods work on the basis of object identifire. These methods help in getting the persistent instances from the database which can be updated and deleted and by using appropriate methods in hibernate session it can be reflected in the database accordingly.

Updating the entities :

Update and delete operation

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. These methods work on the basis of object identifire. These methods help in getting the persistent instances from the database which can be updated and deleted and by using appropriate methods in hibernate session it can be reflected in the database accordingly.

Updating the entities :

```
Session session = sf.openSession();
Emp e = session.get(Emp.class, 102);
e.setSalary(55900.00f);
Transaction tr = session.beginTransaction();
session.update(e);
tr.commit();
System.out.println("Emp updated");
session.close();
sf.close();
```

Deleting the entities :

```
Session session = sf.openSession();
Emp e = session.get(Emp.class, 110);
Transaction tr = session.beginTransaction();
session.delete(e);
tr.commit();
System.out.println("Emp deleted");
session.close();
sf.close();
```

Assignments

1. Create Maven project by adding dependency for hibernate and mysql
2. Check the maven dependencies folder for jar files added.
3. Create mapping class for Emp which is mapped for emp table
4. Create new employee instance and save the instance

Deleting the entities :

```
Session session = sf.openSession();
Emp e = session.get(Emp.class, 110);
Transaction tr = session.beginTransaction();
session.delete(e);
tr.commit();
System.out.println("Emp deleted");
session.close();
sf.close();
```

Assignments

1. Create Maven project by adding dependency for hibernate and mysql
2. Check the maven dependencies folder for jar files added.
3. Create mapping class for Emp which is mapped for emp table
4. Create new employee instance and save the instance