Hibernate WorkFlow Reference Doc

1. **Creating a Java Project**
2. **Adding Hibernate Jars to classpath**
3. **Creating Hibernate Configuration xml file**
4. **Creating Sample Project components and Classes**

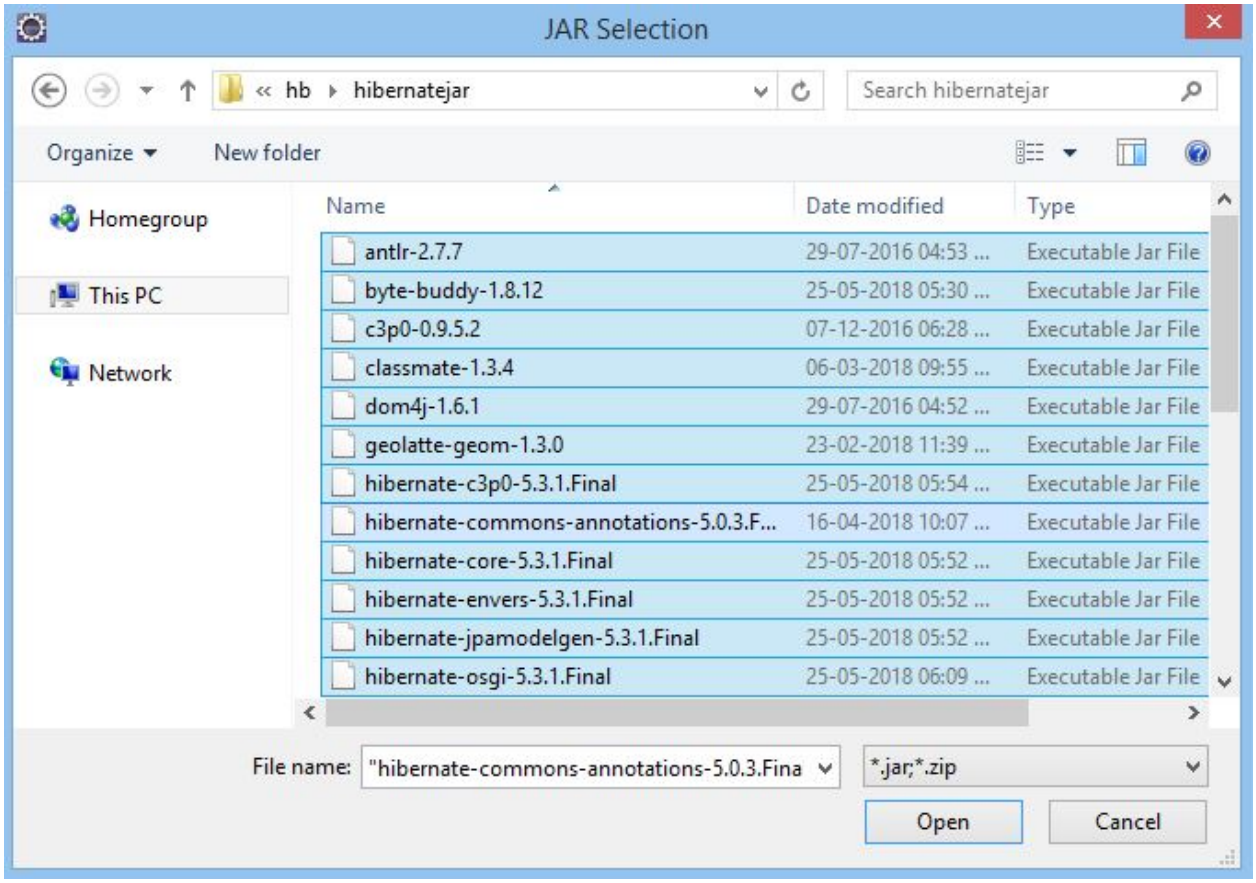
1.Creating Java Project

Create the java project by **File Menu** - **New** - **project** - **java project** . Now specify the project name e.g. HibernateProject then **next** - **finish** .

2.Adding Hibernate Jars to ClassPath

In order to use Hibernate classes and interfaces ,we have to add the dependencies for those classes and interfaces,and we will do that by adding the Jars to the classPath of our project.
We will require a number of jars because of their inter dependency on each other.

To add the jar files **Right click on your project** - **Build path** - **Add external archives**. Now select all the jar files as shown in the image given below then click open.



3.Creating Hibernate Configuration xml file

Hibernate is a ORM framework which basically does Object relational mapping ,which maps a Java object to a table in Database,and enables us to do operations on databases through our Java code efficiently and effectively.
So in order to communicate with the database, hibernate should know some basic configuration about the database and the persistent entities in Java which are mapped to tables in the database.

So In order to communicate with the database we have to provide some configuration about it in the form of tags inside a xml file.
We can give whatever name we want to that file ,but that file should be present as a resource at classpath at location src/ inside our project.
We will be using this xml file inside our java code in order to communicate with database through hibernate libraries-

So the xml file looks like this-

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property
name="connection.url">jdbc:mysql://localhost:3306/hibernate_one-to-one_uni?useSSL=false&a
mp;serverTimezone=UTC</property>
        <property name="connection.username">root</property>
        <property name="connection.password">070898</property>

        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>

        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
```

```
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>

                <!-- Set the current session context -->
                <property name="current_session_context_class">thread</property>
                <property name="hibernate.hbm2ddl.auto">update</property>

    </session-factory>

</hibernate-configuration>
```

In the above hibernate.cfg.xml configuration file we are defining the basic configuration for our databases inside the <session-factory> tag along with some other hibernate configuration.

```
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property
name="connection.url">jdbc:mysql://localhost:3306/hibernate_one-to-one_uni?useSSL=false&a
mp;serverTimezone=UTC</property>
        <property name="connection.username">root</property>
        <property name="connection.password">070898</property>
```

These properties contains the url,username,password for our db.

4.Creating Student Course Management Project

In this project we will be storing some student data along with the student registration in different courses in the database.
We will be using hibernate for this purpose to interact with the database and map the database tables to Java Objects and vice versa.
We will also define mapping between student and courses table in terms of Java classes using hibernate annotations to fetch all courses for a particular student record.

For first of all we will define two classes in two different files -

1. **Student.java-**

```
package com.greatlearning.entity;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name="student")
public class Student {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="id")
	private int id;

	@Column(name="first_name")
	private String firstName;
```

```
	@Column(name="last_name")
	private String lastName;

	@Column(name="email")
	private String email;

	//Set up mapping between Student and Courses table
	@OneToMany(mappedBy="student",
				cascade= {CascadeType.PERSIST,
CascadeType.MERGE,CascadeType.DETACH, CascadeType.REFRESH})
	private List<Course> courses;



	@Override
	public String toString() {
		return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" +
lastName + ", email=" + email
					+ "]";
	}

	public Student() {

	}

	public Student(String firstName, String lastName, String email) {
		super();

		this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getFirstName() {
```

```
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public List<Course> getCourses() {
		return courses;
	}

	public void setCourses(List<Course> courses) {
		this.courses = courses;
	}

	//adding a convenience method for bi-directional relationship
	public void add(Course tempCourse) {
		if(courses==null) {
			courses = new ArrayList<>();
		}
		courses.add(tempCourse);
		tempCourse.setStudent(this);
	}
}
```

2. **Course.java-**

```
package com.greatlearning.entity;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="course")
public class Course {


	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="id")
	private int id;

	@Column(name="name")
	private String name;

	//We should not use CascadeType.Delete here
	@ManyToOne(cascade= {CascadeType.PERSIST, CascadeType.MERGE,
				CascadeType.DETACH, CascadeType.REFRESH})
	@JoinColumn(name="student_id")
	private Student student;

	public Course() {

	}

	public Course(String name) {
		this.name = name;
	}
```

```
	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}



	public Student getStudent() {
		return student;
	}

	public void setStudent(Student student) {
		this.student = student;
	}

	@Override
	public String toString() {
		return "Course [id=" + id + ", name=" + name + "]";
	}

}
```

Student.java and Course.java contains many hibernate annotations like-

@Entity Annotation

The EJB 3 standard annotations are contained in the javax.persistence package, so we import this package as the first step. Second, we used the @Entity annotation to the Student and Course class, which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now, we are using just table name, which is EMPLOYEE.

@Id and @GeneratedValue Annotations

Each entity bean will have a primary key, which you annotate on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation, which takes two parameters strategy and generator that I'm not going to discuss here, so let us use only the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes –

- name attribute permits the name of the column to be explicitly specified.
- length attribute permits the size of the column used to map a value particularly for a String value.
- nullable attribute permits the column to be marked NOT NULL when the schema is generated.
- unique attribute permits the column to be marked as containing only unique values.

@OneToMany Annotation

one-to-many mapping means that one row in a table is mapped to multiple rows in another table. In our case each Student record can have multiple records in course table.

The ***@JoinColumn*** annotation helps us specify the column we'll use for joining an entity association or element collection. On the other hand, the ***mappedBy*** attribute is used to define the referencing side (non-owning side) of the relationship.

3. Driver Class-

```
package com.greatlearning.Driver;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;


import com.greatlearning.entity.Course;
import com.greatlearning.entity.Student;


public class InsertStudentAndCourses {

	public static void main(String[] args) {

		// create session factory
		SessionFactory factory = new
Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Student.class)
				.addAnnotatedClass(Course.class).buildSessionFactory();

		// create session
		Session session = factory.getCurrentSession();

		try {

			// start transaction
			session.beginTransaction();

			// create the objects
			Student tempStudent = new Student("Harshit", "Choudhary",
"HarshitChoudhary@greatlearning.com");

			Course course1 = new Course("Python");
			Course course2 = new Course("Java");
```

```
                        tempStudent.add(course1);
                        tempStudent.add(course2);




                        // save the student
                        session.save(tempStudent);

                        //save the course
                        session.save(course1);
                        session.save(course2);


                        // commit transaction
                        session.getTransaction().commit();

                        System.out.println("Completed Successfully");

                } finally {
                        factory.close();
                }
        }
}
```

This is our main driver class which is creating a session-factory object from hibernate.cfg.xml file which has configuration details about database and session factory bean.

We can then get multiple session objects from a session-factory instance. And can create transactions which can do insertion, updation and deletion in the database.