Java – Hibernate Framework

1. Introduction to Hibernate Architecture

Q. What is Hibernate?

Ans: Hibernate is an **Object-Relational Mapping (ORM)** tool in Java that simplifies database interactions. It allows you to map Java objects to database tables and vice versa, automating CRUD (Create, Read, Update, Delete) operations.

Comparison between Hibernate and JDBC:

- **JDBC**: A low-level API for connecting to and interacting with databases, requiring explicit handling of SQL queries, result sets, and connections.
- **Hibernate**: An abstraction layer over JDBC that reduces the need for manual SQL and allows you to work with Java objects directly, providing automatic handling of database interactions.

Why Use Hibernate? (Advantages):

- **Database Independence**: Hibernate abstracts database-specific details, making it easier to switch databases.
- **Automatic Table Creation**: It can automatically generate database tables based on Java classes.
- **HQL (Hibernate Query Language)**: A powerful query language similar to SQL but designed to work with Java objects.
- **Caching**: Hibernate supports caching, improving performance by reducing database access.
- **Less Boilerplate Code**: It reduces the need for repetitive SQL statements and error-prone database handling code.

Q. Hibernate Architecture:

1) **SessionFactory**:

- It is the central interface for configuring Hibernate and creating **Session** objects.
- A **SessionFactory** is typically created once and used throughout the application. It reads configuration files (hibernate.cfg.xml) to establish database connection settings and mappings.

2) **Session**:

- The **Session** is the primary interface for interacting with the database.
- It allows for CRUD operations on Java objects and acts as a bridge between the application and the database.
- Sessions are used to load, save, update, or delete persistent objects.

3) **Transaction**:

- **Transaction** is used to manage the database transactions (e.g., commit, rollback).
- It ensures data consistency and atomicity during operations, like saving or updating multiple entities in one transaction.

4) **Query**:

- **Query** represents an HQL (Hibernate Query Language) statement, which is a database-independent query language.
- It allows for querying and retrieving data in an object-oriented manner, such as session.createQuery("FROM Employee WHERE salary > 1000").

5) **Criteria**:

- **Criteria API** allows for building dynamic, object-oriented queries in a type-safe manner.

- It helps in constructing queries programmatically (especially when queries need to be built based on dynamic conditions), avoiding direct string-based HQL queries.

2. Hibernate Relationships (One-to-One, One-to-Many, Many-to-One, Many-to-Many)

Q. Object Relationships in Hibernate:

Hibernate maps relationships between Java objects and database tables using annotations or XML configurations. These relationships help represent how entities (Java objects) are associated with each other in the database. Hibernate provides mechanisms to manage these relationships in an efficient way.

1).One-to-One Relationship:

- **Definition**: In a **One-to-One** relationship, a single instance of one entity is associated with a single instance of another entity.
- **Example**: A Person entity has a one-to-one relationship with a Passport entity, where each person has one passport.
- **Hibernate Mapping**:
    - Use @OneToOne annotation in the entity classes.

In the database, this is usually represented by adding a foreign key to one of the tables.

2).One-to-Many Relationship:

- **Definition**: In a **One-to-Many** relationship, one entity can be associated with multiple related entities.
- **Example**: A Department entity can have multiple Employee entities.
- **Hibernate Mapping**:
    - Use @OneToMany annotation in the "one" side (parent) and @ManyToOne in the "many" side (child).

    - The Employee table will contain a foreign key reference to the Department table.

3). Many-to-One Relationship:

- **Definition**: In a **Many-to-One** relationship, multiple entities can be associated with a single instance of another entity.
- **Example**: Many Employee entities are associated with a single Department.
- **Hibernate Mapping**:
    - Use @ManyToOne annotation in the child entity (many side) and @OneToMany in the parent entity (one side).

    - The Employee table will contain a foreign key column (dept_id) referencing the Department table.

4). Many-to-Many Relationship:

- **Definition**: In a **Many-to-Many** relationship, multiple instances of one entity are associated with multiple instances of another entity.
- **Example**: A Student entity can be associated with multiple Course entities, and a Course can have multiple Students.
- **Hibernate Mapping**:
    - Use @ManyToMany annotation in both entities and optionally use a **join table** to store the relationship.

The student_course join table stores the association between students and courses.

Mapping Relationships in Hibernate Using Annotations

Hibernate provides several annotations to map different types of relationships between Java objects (entities) and database tables. Below is how you can map these relationships using annotations:

1).@OneToOne (One-to-One Relationship):

  - **Purpose**: Maps a one-to-one relationship between two entities.
  - **Example**: A Person entity is associated with one Passport entity.

2). @OneToMany (One-to-Many Relationship):

- **Purpose**: Maps a one-to-many relationship where one entity is associated with multiple instances of another entity.
- **Example**: A Department has many Employees.

3). @ManyToOne (Many-to-One Relationship):

- **Purpose**: Maps a many-to-one relationship where multiple entities are associated with one entity.
- **Example**: Many Employee entities belong to one Department.

4). @ManyToMany (Many-to-Many Relationship):

- **Purpose**: Maps a many-to-many relationship between two entities, where multiple instances of one entity are associated with multiple instances of another entity.
- **Example**: A Student can enroll in multiple Courses, and a Course can have multiple Students.

3. Hibernate CRUD Example

CRUD Operations in Hibernate

In Hibernate, CRUD operations refer to the basic operations (Create, Read, Update, and Delete) that you perform on the database using Hibernate's ORM (Object-Relational Mapping) framework. Here's a simple explanation of each operation:

1. Create (Insert):

To insert a new record into the database, Hibernate provides the save() or persist() method. These methods allow you to create a new entity and store it in the database.

- session.save(student) inserts the student object into the database.

- session.get(Student.class, 1) retrieves the student with ID 1.
- session.createQuery("FROM Student", Student.class).list() retrieves all students.

3. Update:

To update an existing record, Hibernate provides the update() method. First, you fetch the entity, modify its attributes, and then commit the changes.

2. Read (Select):

To fetch data from the database, Hibernate provides methods like get(), load(), and createQuery() to retrieve records.

- session.update(student) updates the student record in the database.

4. Delete:

To delete a record, Hibernate provides the delete() method. You first fetch the entity and then delete it.

- session.delete(student) removes the student record from the database.

Writing HQL (Hibernate Query Language)

- **HQL Basics**: HQL is similar to SQL but operates on Java objects (entities) instead of database tables. It's object-oriented, querying entity classes and their properties.
- **Difference Between HQL and SQL**:
  - **SQL**: Works with tables and columns.
  - **HQL**: Works with entities and their fields (e.g., Student class and age property).

CRUD Operations Using HQL:

- **Create**: Use session.save() or session.persist() to insert records.
- **Read**: Use createQuery() to fetch data, e.g., session.createQuery("FROM Student WHERE age > 20").
- **Update**: Use session.createQuery("UPDATE Student SET name = :name WHERE id = :id") for updates.

- **Delete**: Use session.createQuery("DELETE FROM Student WHERE id = :id") to delete records