Hibernate Notes

Name: Raj Prajapati

Institute: Tops Technologies

Subject: Hibernate

Hibernate Notes

Index

Hibernate Notes

1. Introduction to Hibernate

- **What is Hibernate?**
    - Hibernate is an Object-Relational Mapping (ORM) framework for Java applications.
    - It allows you to map Java objects to database tables and vice versa.
- **Why Hibernate?**
    - Reduces boilerplate code compared to JDBC.
    - Provides database independence.
    - Offers caching mechanisms for performance optimization.

Hibernate Architecture

Hibernate's architecture consists of several components that work together to provide ORM functionality. Here's a detailed breakdown:

1. **Core Components**:
    - **Configuration**:
        - The Configuration object is used to configure Hibernate.
        - It loads settings from configuration files like hibernate.cfg.xml or hibernate.properties.
        - **Example:**

```
Configuration configuration = new Configuration().configure("hibernate.cfg.xml");
```

    - **SessionFactory**:
        - A factory for creating Session objects.
        - It is a heavyweight object and is thread-safe, so it is created only once per application.
    - **Session**:
        - Represents a connection to the database.
        - It is used to perform CRUD operations and queries.
        - Not thread-safe; each thread should have its own session.
    - **Transaction**:
        - Used to manage database transactions.

- Ensures data consistency during operations.
- Example:

```
Transaction tx = session.beginTransaction();
```

- **Query**:
  - Allows you to execute HQL (Hibernate Query Language) or SQL queries.
  - Can be used for dynamic or static queries.

2. **Layers of Hibernate Architecture**:
   - **Java Application Layer**:
     - The application interacts with Hibernate through APIs (Session, Transaction, etc.).
   - **Hibernate Framework Layer**:
     - Contains the ORM logic to map objects to database tables.
   - **Database Layer**:
     - Directly interacts with the database using SQL.
3. **Lifecycle of an Entity**: (Different pdf is Created For this Lifecycle)
   - Transient → Persistent → Detached → Removed.
   - These states define how an entity is managed during its interaction with Hibernate.

❖

Hibernate Notes

2. Hibernate Basics and Setup

- **Prerequisites**:
  - JDK installed.
  - Maven/Gradle for dependency management.
  - Database (e.g., MySQL, PostgreSQL).
- **Steps to Set Up Hibernate**:
  1. Add Hibernate dependency to your Maven pom.xml:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.15.Final</version>
</dependency>
```

  2. Create the hibernate.cfg.xml file:

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your_database</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
  </session-factory>
</hibernate-configuration>
```

  3. Create the first Hibernate entity class (e.g., Employee).
  4. Write a test application to save and retrieve data.

❖

Hibernate Notes

Hibernate Core Methods

1. save()

- **Description**:
  - Used to save an object into the database.
  - Assigns a generated identifier to the entity.
  - Returns the generated identifier as a Serializable.
- **Entity State**:
  - Moves an entity from the **Transient** state to the **Persistent** state.
- **Example**:

  Employee emp = new Employee();

  emp.setName("John Doe");

  session.save(emp); // emp is now Persistent
- **Effect in Database**:
  - An INSERT statement is immediately executed.

2. persist()

- **Description**:
  - Similar to save(), but doesn't return the generated identifier.
  - Works only within an active transaction.
  - Follows JPA specifications.
- **Entity State**:
  - Moves an entity from the **Transient** state to the **Persistent** state.
- **Example**:

  Employee emp = new Employee();

  emp.setName("Jane Doe");

  session.persist(emp); // emp is now Persistent
- **Effect in Database**:
  - The INSERT statement is executed when the transaction is committed.

3. update()

- **Description**:
  - Reattaches a **Detached** entity to a Hibernate session, so Hibernate starts managing it again.
  - Cannot be used if the entity already exists in the session (throws NonUniqueObjectException).
- **Entity State**:
  - Moves an entity from the **Detached** state back to the **Persistent** state.
- **Example**:

```
Employee emp = new Employee();
emp.setId(1);
emp.setName("Updated Name");
session.update(emp);
```

- **Effect in Database**:
  - An UPDATE statement is executed.

4. merge()

- **Description**:
  - Similar to update(), but can be used when there is an existing instance of the entity in the session.
  - Returns a new, managed instance of the entity.
- **Entity State**:
  - Moves an entity from the **Detached** state to the **Persistent** state.
- **Example**:

```
Employee detachedEmp = session.get(Employee.class, 1);
detachedEmp.setName("Updated Name");
Employee managedEmp = session.merge(detachedEmp);
```

- **Effect in Database**:
  - An UPDATE statement is executed.

5. delete()

- **Description**:

    - Removes an entity from the database.
    - The entity enters the **Removed** state.
- **Entity State**:
    - Moves an entity from the **Persistent** state to the **Removed** state.
- **Example**:

```
Employee emp = session.get(Employee.class, 1);
session.delete(emp);
```

- **Effect in Database**:
    - A DELETE statement is executed.

6. get()

- **Description**:
    - Fetches an entity by its identifier.
    - Returns null if the entity is not found.
    - Fetches the entity immediately (eager loading).
- **Entity State**:
    - The entity is in the **Persistent** state after retrieval.
- **Example**:

```
Employee emp = session.get(Employee.class, 1); // Returns Employee with ID 1
```

- **Effect in Database**:
    - Executes a SELECT query immediately.

7. load()

- **Description**:
    - Fetches a proxy of the entity (lazy loading).
    - Throws ObjectNotFoundException if the entity is not found and accessed later.
- **Entity State**:
    - Proxy object is Persistent after initialization.
- **Example**:

```
Employee emp = session.load(Employee.class, 1); // Returns a proxy
System.out.println(emp.getName()); // Executes SELECT if accessed
```

- **Effect in Database**:
  - Executes a SELECT query only when the entity is accessed.

8. evict()

- **Description**:
  - Removes an entity from the session's persistence context.
  - The entity becomes **Detached**.
- **Entity State**:
  - Moves an entity from the **Persistent** state to the **Detached** state.
- **Example**:

```
session.evict(emp);
```

- **Effect in Database**:
  - No immediate effect, as the object is no longer managed.

9. refresh()

- **Description**:
  - Reloads the state of a Persistent entity from the database, overwriting any changes in the current session.
- **Entity State**:
  - Keeps the entity in the **Persistent** state.
- **Example**:

```
session.refresh(emp);
```

- **Effect in Database**:
  - Executes a SELECT query to reload the entity from the database.

10. clear()

- **Description**:
  - Clears the session's persistence context, detaching all managed entities.
- **Entity State**:

- Moves all entities from the **Persistent** state to the **Detached** state.

- **Example**:

  session.clear();

- **Effect in Database**:
  - No immediate effect on the database.

11. saveOrUpdate()

- **Description**:
  - Saves the entity if it is in the **Transient** state or updates it if it is in the **Detached** state.
- **Entity State**:
  - Transient → Persistent or Detached → Persistent.
- **Example**:

  session.saveOrUpdate(emp);

- **Effect in Database**:
  - Executes an INSERT or UPDATE depending on the state of the entity.

State Transition Table for Hibernate Methods:

Method	Initial State	Final State	Database Action
`save()`	Transient	Persistent	`INSERT` immediately
`persist()`	Transient	Persistent	`INSERT` on transaction commit
`update()`	Detached	Persistent	`UPDATE` immediately
`merge()`	Detached	Persistent	`UPDATE` immediately
`delete()`	Persistent	Removed	`DELETE` on transaction commit
`get()`	-	Persistent	`SELECT` immediately
`load()`	-	Proxy (Persistent)	`SELECT` on access
`evict()`	Persistent	Detached	No database action
`refresh()`	Persistent	Persistent	`SELECT` immediately
`clear()`	Persistent	Detached	No database action
`saveOrUpdate()`	Transient/Detached	Persistent	`INSERT` or `UPDATE`

3. Hibernate Mapping

Mapping is the process of associating Java objects with database tables. It determines how properties in a class correspond to columns in a table.

1. **Entity Mapping**:
    - Annotate a class with @Entity to mark it as a Hibernate entity.
    - Use @Table to specify the table name in the database (optional if the name matches the class name).
2. **Field Mapping**:
    - Map class fields to table columns using @Column.
    - You can define column-specific properties like name, nullable, length, etc.

```
@Column(name = "employee_name", nullable = false, length = 100)
private String name;
```

3. **Primary Key Mapping**:
    - Define a primary key field with @Id.
    - Use @GeneratedValue to generate values automatically (e.g., IDENTITY, SEQUENCE).

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

4. **Mapping Types**:
    - Hibernate supports primitive types (e.g., int, String) and custom types.
    - **Example:**

```
@Column(name = "salary")
private double salary;
```

- **Example:**

```java
@Entity
@Table(name = "Employee")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    // Getters and Setters
}
```

❖

Hibernate Notes

4. Relationships in Hibernate

- **Types of Relationships**:
    - **One-to-One**: @OneToOne.
    - **One-to-Many**: @OneToMany.
    - **Many-to-One**: @ManyToOne.
    - **Many-to-Many**: @ManyToMany.

1. One-to-One Relationship

- **Definition**: A @OneToOne relationship maps one entity to exactly one related entity. This is common in relationships like a User and Profile.
- **Hibernate Mapping**:

    @OneToOne

    @JoinColumn(name = "profile_id") // Foreign Key in User table

    private Profile profile;

- **Database Result**:
    - In the User table, a foreign key column (profile_id) will reference the primary key of the Profile table.
- **Action Example**: When saving a User object, the associated Profile object is also saved (if CascadeType.PERSIST is enabled).

    User user = new User();

    Profile profile = new Profile();

    user.setProfile(profile);

    session.save(user);

Effect in Database:

- Two rows are created:
    - One in the User table with the profile_id as a foreign key.
    - One in the Profile table.

2. One-to-Many Relationship

- **Definition**: A @OneToMany relationship maps one entity to a list/collection of another entity. Example: Department having many Employee entities.

- **Hibernate Mapping**:

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

  private List<Employee> employees;

- **Database Result**:
  - In the Employee table, a foreign key column (department_id) will reference the primary key of the Department table.
- **Action Example**: When adding employees to a department and saving the department:

  Department department = new Department();

  Employee emp1 = new Employee();

  Employee emp2 = new Employee();

  emp1.setDepartment(department);

  emp2.setDepartment(department);

  department.setEmployees(Arrays.asList(emp1, emp2));

  session.save(department);

Effect in Database:

- One row is created in the Department table.
- Two rows are created in the Employee table, with department_id set to the primary key of the Department row.

3. Many-to-One Relationship

- **Definition**: A @ManyToOne relationship maps multiple entities to a single related entity. Example: Many Employee objects belonging to one Department.
- **Hibernate Mapping**:

  @ManyToOne

  @JoinColumn(name = "department_id") // Foreign Key in Employee table

  private Department department;

- **Database Result**:
  - In the Employee table, a department_id foreign key column references the primary key of the Department table.

- **Action Example**: When saving an employee with a department:

```
Department department = new Department();
Employee emp = new Employee();
emp.setDepartment(department);
session.save(department);
session.save(emp);
```

Effect in Database:

- One row is created in the Department table.
- One row is created in the Employee table with department_id pointing to the department's primary key.

4. Many-to-Many Relationship

- **Definition**: A @ManyToMany relationship maps multiple entities to multiple related entities. Example: Student and Course.
- **Hibernate Mapping**:

```
@ManyToMany
@JoinTable(
  name = "student_course", // Join table
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id")
)
private List<Course> courses;
```

- **Database Result**:
  - A join table (student_course) is created with two columns:
    - student_id: Foreign key referencing the Student table.
    - course_id: Foreign key referencing the Course table.
- **Action Example**: When enrolling a student in courses:

```
Student student = new Student();
Course course1 = new Course();
Course course2 = new Course();
```

student.setCourses(Arrays.asList(course1, course2));

session.save(student);

Effect in Database:

- Rows are created in the Student and Course tables.
- Two rows are added to the student_course join table to map the relationships.

5. Cascading and FetchType Impacts

- **Cascade**:
  - Cascade operations (PERSIST, REMOVE, ALL) determine whether related entities are automatically persisted or deleted.
  - For example:
    - If **CascadeType.ALL** is enabled on a parent entity, saving or deleting the parent automatically saves or deletes all associated child entities.
- **FetchType**:
  - **FetchType.LAZY**: Associated entities are loaded only when accessed.
  - **FetchType.EAGER**: Associated entities are loaded immediately with the parent entity.

6.Inheritance Mapping:

- Hibernate supports inheritance between entities.
- Use annotations like **@Inheritance(strategy = InheritanceType.SINGLE_TABLE).**

Summary Table of Actions and Database Changes:

Relationship Type	Action	Database Effect
One-to-One	Save parent object (`User`)	Creates a row in parent table and associated row in child table (`Profile`)
One-to-Many	Save parent with children (`Department` & `Employees`)	Creates a row in parent table and multiple rows in child table with foreign key
Many-to-One	Save child object (`Employee` with `Department`)	Creates a row in child table with foreign key to parent table
Many-to-Many	Save entity with associated list (`Student` & `Courses`)	Rows in both entity tables and corresponding rows in the join table

5. Hibernate Query Language (HQL)

Hibernate Query Language (HQL) - Comprehensive Notes

1. What is HQL?

- Hibernate Query Language (HQL) is an object-oriented query language similar to SQL.
- It operates on Hibernate entities and attributes, rather than directly on database tables and columns.
- HQL is case-insensitive for keywords but case-sensitive for entity names and attributes.

2. Key Features of HQL

- Object-oriented: Queries are written based on entity classes.
- Supports joins: Works with entities that have relationships (e.g., One-to-Many, Many-to-One).
- Portable: Database-independent, as Hibernate translates HQL into native SQL for the underlying database.
- Dynamic: HQL queries can be written dynamically during runtime.

3. Basic Syntax of HQL

- **Selecting Data**:

  ```
  FROM EntityName

  SELECT e FROM EntityName e
  ```

  Example:

  ```
  SELECT e FROM Employee e
  ```

- **Filtering Data (**WHERE **Clause)**:

  ```
  SELECT e FROM Employee e WHERE e.name = 'John'
  ```

  Example with a parameterized query:

  ```
  Query query = session.createQuery("FROM Employee e WHERE e.name =
  :name");

  query.setParameter("name", "John");

  List<Employee> employees = query.list();
  ```

- **Sorting Data (**ORDER BY **Clause)**:

```
SELECT e FROM Employee e ORDER BY e.salary DESC
```

- **Pagination**: Set the maximum number of results (maxResults) and the starting position (firstResult):

```
Query query = session.createQuery("FROM Employee");

query.setFirstResult(10);

query.setMaxResults(5);

List<Employee> employees = query.list();
```

4. Common Operations in HQL

- **Retrieve All Records**:

  FROM Employee

- **Retrieve Specific Attributes**:

  SELECT e.name, e.salary FROM Employee e

- **Update Records**:

  UPDATE Employee e SET e.salary = e.salary + 1000 WHERE e.department = 'IT'

Executing the query:

```
Query query = session.createQuery("UPDATE Employee e SET e.salary = e.salary + 1000 WHERE e.department = :dept");

query.setParameter("dept", "IT");

int result = query.executeUpdate();
```

- **Delete Records**:

```
DELETE FROM Employee e WHERE e.id = 5
```

Executing the query:

```
Query query = session.createQuery("DELETE FROM Employee e WHERE e.id = :id");

query.setParameter("id", 5);

int result = query.executeUpdate();
```

5. Advanced HQL Features

- **Joins in HQL**:
  - Inner Join:

SELECT e.name, d.name FROM Employee e INNER JOIN e.department d

- Left Join:

  SELECT e.name, d.name FROM Employee e LEFT JOIN e.department d

- **Aggregations (**GROUP BY **and Aggregate Functions)**:
  - Example:

    SELECT d.name, AVG(e.salary) FROM Employee e INNER JOIN e.department d GROUP BY d.name

  - Aggregate Functions:
    - AVG(): Average.
    - SUM(): Sum.
    - COUNT(): Count rows.
    - MIN(), MAX(): Minimum/Maximum.
- **Named Queries**:
  - Defining a named query:

    @NamedQuery(name = "findEmployeesByDepartment", query = "FROM Employee e WHERE e.department.name = :deptName")

  - Using the named query:

    Query query = session.getNamedQuery("findEmployeesByDepartment");

    query.setParameter("deptName", "IT");

    List<Employee> employees = query.list();

6. Parameterized Queries

- Positional Parameters (? syntax):

  FROM Employee e WHERE e.salary > ?1

  Example:

  Query query = session.createQuery("FROM Employee e WHERE e.salary > ?1");

  query.setParameter(1, 5000);

  List<Employee> employees = query.list();

- Named Parameters (**:parameterName** syntax):

  FROM Employee e WHERE e.department.name = :deptName

Example:

```
Query query = session.createQuery("FROM Employee e WHERE e.department.name = :deptName");

query.setParameter("deptName", "IT");

List<Employee> employees = query.list();
```

7. HQL vs SQL

Feature	HQL	SQL
Focus	Operates on entities and attributes	Operates on tables and columns
Joins	Object-oriented relationships	Requires manual joins
Database	Database-independent	Database-specific
Query Type	Object-oriented	Relational

8. Practical Use Cases

- **Fetching Employees by Department**:

  SELECT e FROM Employee e WHERE e.department.name = 'HR'

- **Finding Maximum Salary in a Department**:

  SELECT MAX(e.salary) FROM Employee e WHERE e.department.name = 'Finance'

- **Deleting Employees with Low Salary**:

  DELETE FROM Employee e WHERE e.salary < 3000

9. Best Practices for HQL

- Use **parameterized queries** to prevent SQL injection.
- Combine **joins** with FetchType.LAZY to avoid unnecessary data loading.
- Use **pagination** for large result sets.
- Avoid fetching unneeded data—retrieve only the required attributes.

❖

Hibernate Notes

6. Criteria API

- **What is Criteria API?**
    - A programmatic alternative to HQL for building dynamic queries.
- **Example**:

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> root = cq.from(Employee.class);
cq.select(root).where(cb.equal(root.get("name"), "John"));
List<Employee> employees = session.createQuery(cq).getResultList();
```

7. Transactions and Caching

- **Transactions**:
    - Hibernate uses database transactions for data consistency.
    - Typical workflow: beginTransaction() -> save/update/delete -> commit().
- **Caching**:
    - **First-Level Cache**: Enabled by default and works at the session level.
    - **Second-Level Cache**: Needs configuration (e.g., EHCache, Redis).
    - **Query-Level Cache**: Cache the results of frequently used queries.

8. Advanced Topics

- **Inheritance Mapping**:
    - Map class hierarchies with @Inheritance(strategy = InheritanceType.SINGLE_TABLE) or TABLE_PER_CLASS.
- **Native Queries**:
    - Use SQL directly with Hibernate for complex queries:

```
Query query = session.createSQLQuery("SELECT * FROM Employee");
```

- **Batch Processing**:
    - Efficiently handle large datasets with batching:
    - **Example:**

```java
for (int i = 0; i < employees.size(); i++) {
    session.save(employees.get(i));
    if (i % 50 == 0) {
        session.flush();
        session.clear();
    }
}
```

❖