1.Introduction to Hibernate Architecture

1.What is Hibernate?

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

1. ***Definition and purpose of Hibernate as an ORM(ObjectRelationalMapping)tool.***
   An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.The ORM tool internally uses the JDBC API to interact with the database.

Hibernate allows you to:

1. **Map Java objects to database tables**: Every Java class can be mapped to a database table, and each instance of the class corresponds to a row in the table.
2. **Automate SQL generation**: Hibernate automatically generates SQL statements to perform CRUD operations based on the defined mappings, reducing the need for manual SQL writing.
3. **Provide a persistence context**: Hibernate manages the lifecycle of entities (objects) in a session context, including loading, saving, and updating data in a database.

Purpose of Hibernate as an ORM Tool:

1. **Simplify Database Interaction**:
   - Hibernate abstracts away the complexity of database interactions, making it easier for developers to work with databases using plain Java objects. This reduces the need for boilerplate JDBC (Java Database Connectivity) code.
2. **Improve Productivity**:
   - By using Hibernate, developers can focus more on business logic rather than worrying about database-specific details like writing SQL queries or managing database connections and transactions.
3. **Cross-database Compatibility**:
   - Hibernate provides a layer of abstraction that allows applications to be database-agnostic. It supports multiple relational databases (e.g., MySQL, PostgreSQL, Oracle) and can switch between them with minimal changes to the code.
4. **Maintainability and Reusability**:
   - With Hibernate, database interaction logic is encapsulated within reusable entity classes and mappings, which makes the application easier to maintain and extend. Hibernate's caching mechanisms also improve performance and reduce redundant database queries.
5. **Support for Object-Oriented Programming (OOP)**:

    - Hibernate leverages the principles of object-oriented programming, such as inheritance, polymorphism, and associations (one-to-many, many-to-many, etc.), allowing developers to work in a natural, object-oriented manner instead of dealing directly with tables, rows, and columns.
6. **Automatic SQL Generation**:
    - Hibernate can automatically generate SQL statements for CRUD operations based on the object mapping configuration, saving developers from the repetitive task of writing SQL queries.
7. **Transaction Management**:
    - Hibernate integrates with Java's transaction management API (JTA) and can manage transactions, ensuring that operations like saving, updating, or deleting entities in the database happen in a consistent and reliable manner.
8. **Query Language**:
    - Hibernate provides its own query language, **HQL (Hibernate Query Language)**, which is similar to SQL but works with Java objects and their properties. It also supports **Criteria API** and **Native SQL** for more complex queries.
9. **Caching**:
    - Hibernate supports first-level (session) and second-level (global) caching, which helps optimize performance by reducing the number of database queries for frequently accessed data.

Key Features of Hibernate:

- **Mapping of Java Classes to Database Tables** using annotations or XML files.
- **Automatic SQL Generation** based on the Java object model.
- **Lazy Loading and Eager Loading** for efficient data fetching.
- **Transaction Management** that integrates with JTA.
- **Support for HQL (Hibernate Query Language)** for querying the database using Java objects.
- **Caching Mechanisms** for improving performance by reducing the number of database queries.

2. ***<u>Comparison between Hibernate and JDBC.</u>***

JDBC	Hibernate
In JDBC, one needs to write code to map the object model's data representation to the schema of the relational model.	Hibernate maps the object model's data to the schema of the database itself with the help of annotations.
JDBC enables developers to create queries and update data to a relational database using the Structured Query Language (SQL).	Hibernate uses HQL (Hibernate Query Language) which is similar to SQL but understands object-oriented concepts like inheritance, association etc.

JDBC	Hibernate
JDBC code needs to be written in a try-catch databases block as it throws checked exceptions (SQLexception).	Hibernate manages the exceptions itself by marking them as unchecked.
JDBC is database dependent i.e. one needs to write different codes for different database.	Hibernate is database-independent and the same code can work for many databases with minor changes.
Creating associations between relations is quite hard in JDBC.	Associations like one-to-one, one-to-many, many-to-one, and many-to-many can be acquired easily with the help of annotations.
It is a database connectivity tool.	It is a Java framework.
Lazy Loading is not supported.	Lazy Loading is supported.
It has low performance than Hibernate.	It has high performance.
One needs to maintain explicitly database connections and transactions.	It itself manages its own transactions.
It has a dedicated customer support service system.	Waiting time is more for any answer to an issue.

3. ***Why use Hibernate? (Advantages: Database independence ,automatic table creation, HQL, etc.)***

Hibernate is a powerful object-relational mapping (ORM) framework for Java applications that provides several key advantages when working with databases. Here are some of the main benefits:

1. **Database Independence**:
   - Hibernate provides a layer of abstraction over database-specific SQL, meaning your code does not need to be tightly coupled to a particular database system. This makes it easier to switch between different databases (e.g., from MySQL to PostgreSQL) without significant changes to your application code.

   - The framework automatically generates the appropriate SQL queries based on the underlying database's dialect.
2. **Automatic Table Creation**:
   - Hibernate can automatically generate database tables based on Java entity classes. It uses annotations or XML configuration to map objects to tables, allowing the schema to be automatically created or updated when the application starts.
3. **HQL (Hibernate Query Language)**:
   - Hibernate provides HQL, a database-independent query language, to query the database using object-oriented syntax. HQL allows you to write queries based on the Java objects (entities) rather than directly writing SQL queries.
4. **Automatic Handling of CRUD Operations**:
   - It provides built-in methods and mechanisms for saving, updating, deleting, and retrieving entities, reducing the amount of boilerplate code needed.
5. **Caching**:
   - Hibernate has built-in support for caching, which can significantly improve performance by reducing the number of database queries. First-level cache (Session cache) is enabled by default, and second-level cache (which can be configured with external cache providers) can be used to cache data across sessions.
6. **Lazy Loading**:
   - Hibernate supports lazy loading, meaning related objects are loaded only when they are accessed, rather than being fetched immediately. This can optimize performance and reduce unnecessary database queries.
7. **Relationship Mapping**:
   - Hibernate simplifies the mapping of complex relationships (e.g., one-to-many, many-to-many, one-to-one) between entities. Using annotations like `@OneToMany`, `@ManyToOne`, `@OneToOne`, etc., developers can model these relationships easily.
8. **Transaction Management**:
   - Hibernate integrates seamlessly with Java's transaction management, making it easy to manage database transactions. You can use Hibernate in conjunction with Java EE, Spring, or other frameworks to ensure proper transaction handling.
9. **Extensibility**:
   - Hibernate is highly extensible and supports custom user-defined types, custom SQL queries, and more. You can define your own persistence strategies and integrate Hibernate with other parts of your application as needed.

2.Hibernate Architecture

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.
The Hibernate architecture is categorized in four layers.

- Java application layer
- Hibernate framework layer
- Backhand api layer
- Database layer

1. ***Explanation of the Hibernate architecture components:***

- ***Session Factory: Configuration of Hibernate and creation of sessions.***

  The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Responsibilities:

- Configures Hibernate using the `hibernate.cfg.xml` file or annotations.
- Manages the database connection pool and provides an interface to create and manage `Session` objects.
- The `SessionFactory` is typically created once and reused throughout the application.

- ***Session: The main interface between the Java application and the database.***

  The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Responsibilities:

- Persists new objects to the database (via `save()`, `persist()`, etc.).
- Retrieves objects from the database (via `get()`, `load()`, etc.).
- Updates or deletes objects in the database (via `update()`, `delete()`).
- Manages the first-level cache (session cache), which stores entities and prevents duplicate database queries within the same session.
- A `Session` is short-lived and typically used within a single transaction.

- ***Transaction: Handling database transactions in Hibernate.***

  The `Transaction` object is used to manage database transactions in Hibernate. Transactions ensure that database operations are executed in a consistent and reliable manner, following the ACID (Atomicity, Consistency, Isolation, Durability) properties.

Responsibilities:

- Begins, commits, and rolls back database transactions.
- Ensures that changes to the database are made in a consistent manner and can be rolled back in case of errors.
- Can be associated with a `Session` to manage the lifecycle of a transaction.
- Hibernate provides integration with JTA (Java Transaction API) and native transaction management to handle transactions across different resources.

- ***Query: WritingHQL (HibernateQueryLanguage) queries to interact with the database.***

Hibernate allows querying the database using **HQL (Hibernate Query Language)**, an object-oriented query language. HQL is similar to SQL but operates on entity objects rather than database tables.
Responsibilities:

- Defines and executes queries using HQL, which is database-independent.
- Supports querying entities, relationships, and projections (specific columns).
- Can be parameterized to prevent SQL injection and handle dynamic queries.
- Supports both select queries and updates/deletes.

- ***Criteria: Criteria API for building dynamic queries.***

The `Criteria` API provides a programmatic way of building dynamic queries in Hibernate. It is an alternative to HQL and allows for creating queries in a type-safe, object-oriented manner, making it easier to build complex queries with conditions, joins, and projections.

Responsibilities:

- Facilitates dynamic query generation without needing to write raw HQL or SQL.
- Provides methods to specify query criteria, conditions, sorting, and projections.
- The Criteria API is useful for complex queries, especially when you need to build queries dynamically at runtime.

Note: As of Hibernate 5, the Criteria API is replaced by the **JPA Criteria API**, but it is still supported in Hibernate for backward compatibility.

2. How Hibernate works internally from loading configuration files to executing queries.

1. Configuration and Initialization

The first step in using Hibernate is setting up the configuration file (usually named `hibernate.cfg.xml`) and initializing Hibernate. This file contains the configuration settings necessary for Hibernate to connect to the database, manage entity mappings, and configure other properties.

Steps:

- **Loading Configuration:**

- Hibernate uses the `Configuration` class to load and read the `hibernate.cfg.xml` configuration file.
- The file contains properties like database connection details (URL, username, password), dialect (which SQL variant to use), and other settings like transaction management, cache settings, etc.

- **SessionFactory Creation:**

- After loading the configuration, Hibernate uses it to build a `SessionFactory`. The `SessionFactory` is a heavy object and should be created once per application lifecycle. It is responsible for creating `Session` instances, which are used to interact with the database.
- `SessionFactory` reads the mapping files or annotations that define how Java objects are mapped to database tables.

2. Session Creation

- **Opening a Session:**

- A `Session` is the main interface for interacting with the database. It is created from the `SessionFactory`. The session is a single-threaded object and should be used for a single unit of work.
- The `Session` is used to begin transactions, perform CRUD operations (Create, Read, Update, Delete), and execute HQL (Hibernate Query Language) or SQL queries.

3. Transaction Management

- **Begin Transaction:** A transaction must be started before any changes can be made to the database. This ensures consistency and rollback capability in case of failure.

- **Committing or Rolling Back:** After performing the required operations (insert, update, delete), the transaction is either committed (if successful) or rolled back (in case of an error).

4. Object-Relational Mapping (ORM)

Hibernate maps Java classes to database tables. This mapping can either be configured via XML files or annotations in the Java code.

- **Entity Classes:** Java classes representing database tables must be annotated with `@Entity` and other relevant annotations like `@Id`, `@Table`, `@Column`, etc. These annotations define how Java objects are mapped to the database schema.

5. Performing CRUD Operations

Once the session and transaction are ready, you can perform CRUD operations:

- **Save (Create):** To insert a new entity (record) into the database, the `save()` method is used.

- **Update (Update):** To update an existing entity in the database, you can use the `update()` method or modify the object and call `session.merge()`.

- **Delete (Delete):** To delete an entity from the database, you use the `delete()` method.

- **Retrieve (Read):** To retrieve an entity from the database, you can use methods like `get()`, `load()`, `createQuery()`, or `createCriteria()`.

6. Executing HQL (Hibernate Query Language) and SQL Queries

Hibernate allows querying the database using HQL (Hibernate Query Language), which is object-oriented and similar to SQL but operates on entities instead of database tables.

- **HQL Query:**

```
Query query = session.createQuery("FROM Employee WHERE name = :name");
query.setParameter("name", "John Doe");
List<Employee> employees = query.list();
```

- **SQL Query:**

```
SQLQuery sqlQuery = session.createSQLQuery("SELECT * FROM employee");
sqlQuery.addEntity(Employee.class);
List<Employee> employees = sqlQuery.list();
```

7. Session Closing

After the transaction is committed or rolled back and all operations are completed, the session should be closed to release the resources.

8. SessionFactory Closing

After the entire application finishes, the `SessionFactory` should also be closed to release database connections.

2.HibernateRelationships

1.Object Relationships in Hibernate:

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.

1.How Hibernate manages relationships between Java objects and database tables.

Hibernate manages relationships through annotations or XML configurations that define the type of association between entities (Java objects) and their corresponding tables in the database.

2.Overview of the different types of relationships:

❖ ***One-to-OneRelationship***

A **One-to-One** relationship is a type of relationship where each instance of one entity is related to exactly one instance of another entity. This means that for each record in the first table, there is one and only one corresponding record in the second table.

A single instance of an entity is related to a single instance of an other entity.

Characteristics of a One-to-One Relationship:

- **Uniqueness**: Each entity in the relationship is associated with exactly one entity on the other side.
- **Foreign Key**: The foreign key can be placed in either of the tables, depending on the direction of the association
- **Use Cases**: This type of relationship is often used when two entities are closely related, and each entity’s lifecycle is dependent on the other.

How to Model a One-to-One Relationship in Hibernate:

1. Owning Side with Foreign Key:

In a one-to-one relationship, you can place the foreign key either on the owning side or the inverse side. In Hibernate, the owning side is typically the side that holds the foreign key.

Example Code:

Person Entity (Owning Side):

```
@Entity
public class Person {
    @Id
    private Long id;

    @OneToOne
    @JoinColumn(name = "passport_id")
    private Passport passport;


}
```

Passport Entity (Inverse Side):

```
@Entity
public class Passport {
    @Id
    private Long id;

    @OneToOne(mappedBy = "passport")
    private Person person;

}
```

2. Using the `@PrimaryKeyJoinColumn` Annotation (Alternative Approach):

In some cases, you might want the foreign key to be a part of the primary key of one of the entities. This is useful when both entities are tightly coupled and share the same lifecycle.

Example Code:

Person Entity:

```
@Entity
public class Person {
    @Id
    private Long id;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Passport passport;

}
```

Passport Entity:

```
@Entity
public class Passport {
    @Id
    private Long id;


    @OneToOne(mappedBy = "passport")
    private Person person;

}
```

Cascade Operations in One-to-One Relationships:

This can be done using the **cascade** attribute in Hibernate.

Example with Cascade:

```
@Entity
public class Person {
    @Id
    private Long id;

    @OneToOne(cascade = CascadeType.ALL)
 @JoinColumn(name = "passport_id")
    private Passport passport;
}
```

- **`CascadeType.ALL`** means all operations (persist, merge, delete, etc.) will be cascaded to the associated `Passport` entity whenever an operation is performed on the `Person` entity.

❖ ***<u>One-to-ManyRelationship:</u>***

A **One-to-Many** relationship is one of the most common relationships in Hibernate and relational databases. In this relationship, one entity (the "one" side) can be associated with multiple entities (the "many" side). It allows you to model scenarios where one object has a collection of related objects.

<u>One entity can have multiple related entities.</u>

Characteristics of a One-to-Many Relationship:

- **One Entity, Multiple Related Entities**: An instance of one entity can have many related instances of another entity.
- **Foreign Key**: The foreign key is typically placed on the "many" side (child entity), referencing the "one" side (parent entity).

How to Model a One-to-Many Relationship in Hibernate:

In Hibernate, the **One-to-Many** relationship is typically set up by combining the **`@OneToMany`** annotation on the parent entity (the "one" side) and the **`@ManyToOne`** annotation on the child entity (the "many" side).

1. One-to-Many Relationship (Parent to Child):

The foreign key is stored in the child entity's table.

Example Code:

Department Entity (Parent):

```
@Entity
```

```
public class Department {
    @Id
    private Long id;

    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;    }
```

Employee Entity (Child):

```
@Entity
public class Employee {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
    // Getter and Setter methods
}
```

2. Cascade Operations:

In some cases, you may want operations like `Persist`, `Merge`, or `Delete` to propagate automatically from the parent entity (the "one" side) to the related child entities (the "many" side). This can be achieved using the **cascade** attribute.

Example Code with Cascade:

```
@Entity
public class Department {
    @Id
    private Long id;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)  //
Cascade operations
    private Set<Employee> employees;

    // Getter and Setter methods
}
```

3. Unidirectional One-to-Many:

In a unidirectional one-to-many relationship, only one side of the relationship is aware of the other.

Example Code (Unidirectional):

```
@Entity
public class Department {
    @Id
    private Long id;

    @OneToMany
    @JoinColumn(name = "department_id")  // Foreign key stored in Employee
table
```

```
    private Set<Employee> employees;

    // Getter and Setter methods
}
```

❖ ***Many-to-OneRelationship:***

Many entities are associated with a single entity.

A **Many-to-One** relationship is a type of relationship in which many entities (records or objects) are associated with a single entity. It is commonly used in databases and data modeling to describe how multiple instances of one entity relate to a single instance of another entity.

Example:

- **Many books** can be written by **one author**.
- Each book can only have one author, but an author can write multiple books.

❖ ***Many-to-ManyRelationship:***

Multiple instances of an entity are associated with multiple instances of an other entity.

A **Many-to-Many** relationship occurs when multiple instances of one entity are associated with multiple instances of another entity. In this type of relationship, each entity can have several associations with the other entity.

Example:

- A **student** can enroll in multiple **courses**.
- A **course** can have multiple **students** enrolled in it.

How it works:

This relationship is typically implemented by creating a **junction table** (also known as a **many-to-many relationship table**) to manage the connections between the two entities.

- **Students** table contains student details.
- **Courses** table contains course details.
- A **junction table** (often called **Student_Courses** or **Enrollments**) contains references to both the **Student** and **Course** tables through foreign keys, linking students and courses.

2.Mapping Relationships in Hibernate:

❖ ***The concept of owning and inverse sides in relationships.***

In Hibernate, the concepts of **owning** and **inverse** sides in relationships pertain to how the associations between entities are managed, especially in terms of database updates and the handling of foreign keys. These concepts are important in managing bidirectional relationships and ensuring that Hibernate correctly synchronizes both sides of the relationship. Here's a detailed explanation:

1. Owning Side

- **Definition**: The owning side of a relationship is the side that is responsible for maintaining the association and managing the foreign key in the database. Typically, it is the side where the foreign key column is located.
- **Responsibilities**:
  - The owning side is where Hibernate will perform the **update or insert** operations to maintain the relationship in the database.
  - It controls the cascade operations (e.g., cascading deletes, persist, etc.).
  - It contains the **mappedBy** attribute if the relationship is bidirectional.
- **Example**:

```
      @ManyToOne

@JoinColumn(name = "customer_id")
private Customer customer;
```

In this case, the `Order` entity is the owning side because it holds the foreign key (`customer_id`).

2. Inverse Side

- **Definition**: The inverse side refers to the side of the relationship that does **not** manage the foreign key. It is typically used to define the relationship but does not directly affect the database schema or operations.
- **Responsibilities**:
  - The inverse side does not manage the foreign key directly but helps Hibernate understand the bidirectional mapping.
  - It is the side that is **referenced by** the owning side but does not perform any database updates for the association.
  - The inverse side usually has the `mappedBy` attribute set to indicate the owning side.
- **Example**: `@OneToMany(mappedBy = "customer")`

```
private Set<Order> orders;
```

3. MappedBy Attribute

- The `mappedBy` attribute is used to designate the inverse side in a bidirectional relationship. It tells Hibernate that the relationship is already defined in the owning side, and the inverse side should not manage the foreign key or make updates to it.
- This attribute is typically used in one-to-many, many-to-one, and many-to-many relationships.

4. Example: One-to-Many Relationship

```
@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(mappedBy = "customer")
    private Set<Order> orders;
}

@Entity
public class Order {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;
}
```

5. Advantages of Understanding Owning and Inverse Sides

- **Clear Database Mapping**: Helps clarify which side of the relationship is responsible for maintaining foreign keys.
- **Efficient Cascading**: By properly designating the owning side, Hibernate can handle cascading operations (e.g., delete cascading) more predictably and consistently.
- **Bidirectional Relationships**: Ensures that both sides of the relationship are synchronized and consistent without unnecessary updates.

❖ ***Cascade types and how they affect related entities.***

cascade types define how operations on an entity (such as persist, update, delete) should be propagated to its associated entities. Cascading allows for the automatic propagation of certain operations (e.g., saving or deleting) to related entities in a relationship. This ensures consistency and reduces the need to manually handle operations for each related entity.

Cascade Types in Hibernate

There are several cascade types that Hibernate supports. These can be specified using the `@Cascade` annotation (for legacy Hibernate versions) or, more commonly, using JPA annotations like `@OneToMany`, `@ManyToOne`, `@OneToOne`, and `@ManyToMany`, combined with the `cascade` attribute.

Here are the main cascade types and how they affect related entities:

1. CascadeType.PERSIST

- **Definition**: When an entity is persisted (saved), the associated entities (on the owning side) are also persisted.
- **Effect**: If you call `save()` or `persist()` on an entity, all associated entities in the relationship will also be saved automatically.
- **Example**:

```
@OneToMany(cascade = CascadeType.PERSIST)
private Set<Order> orders;
```

2. CascadeType.MERGE

- **Definition**: When an entity is merged (updated), the associated entities are also merged (updated).
- **Effect**: When you call `merge()` on an entity, the associated entities will also be updated (if they are detached).
- **Example**:

```
@OneToMany(cascade = CascadeType.MERGE)
private Set<Order> orders;
```

3. CascadeType.REMOVE

- **Definition**: When an entity is deleted, the associated entities are also deleted.
- **Effect**: If you call `remove()` on an entity, the associated entities will be deleted as well.
- **Example**:

```
@OneToMany(cascade = CascadeType.REMOVE)
private Set<Order> orders;
```

4. CascadeType.REFRESH

- **Definition**: When an entity is refreshed (i.e., reloaded from the database), the associated entities are also refreshed.
- **Effect**: If you call `refresh()` on an entity, the associated entities are also reloaded from the database to reflect the current state.
- **Example**:

```
@OneToMany(cascade = CascadeType.REFRESH)
private Set<Order> orders;
```

5. CascadeType.DETACH

- **Definition**: When an entity is detached (i.e., removed from the session context), the associated entities are also detached.
- **Effect**: If you call `detach()` on an entity, the associated entities are also detached, meaning they will no longer be part of the persistence context.
- **Example**:

  ```
  @OneToMany(cascade = CascadeType.DETACH)
  private Set<Order> orders;
  ```

  When a `Customer` entity is detached, all associated `Order` entities will also be detached.

6. CascadeType.ALL

- **Definition**: This is a shortcut to specify that **all** cascade operations (`PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`) should be applied.
- **Effect**: This will apply all the cascade behaviors (save, update, delete, refresh, detach) to the related entities.
- **Example**:

  ```
  @OneToMany(cascade = CascadeType.ALL)
  private Set<Order> orders;
  ```

  This means that when a `Customer` is saved, updated, deleted, refreshed, or detached, the same operations will be performed on the `Order` entities.

Example Use Cases

Case 1: Saving a Parent Entity (PERSIST)

Suppose you have a `Customer` entity with a one-to-many relationship to `Order`:

```
@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.PERSIST)
    private Set<Order> orders;
}
```

- If you persist a `Customer` entity, all the associated `Order` entities will automatically be persisted as well.

```
Customer customer = new Customer();
customer.setName("John");
Order order1 = new Order();
order1.setAmount(100);
```

```
customer.addOrder(order1);

session.persist(customer); // This will persist both the customer and the
order.
```

Case 2: Updating an Entity (MERGE)

Consider a scenario where you want to update an entity:

```
@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.MERGE)
    private Set<Order> orders;
}
```

- If you call `merge()` on a `Customer` entity, all the associated `Order` entities (if they are detached) will also be merged (updated).

```
Customer customer = session.get(Customer.class, customerId);
customer.setName("Updated Name");

session.merge(customer);
```

Case 3: Deleting Entities (REMOVE)

In the case of deleting a `Customer`, you may want the associated `Order` entities to also be deleted:

```
@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.REMOVE)
    private Set<Order> orders;
}
```

- If you delete the `Customer`, all associated `Order` entities will be deleted from the database.

```
Customer customer = session.get(Customer.class, customerId);
session.remove(customer);
```

Cascading in Relationships

The effect of cascading depends on the type of relationship and the side of the relationship on which the cascade is applied:

- **One-to-Many and Many-to-One**: Cascading on the `One` side usually makes sense (i.e., cascading operations on the "owning" side). For example, if you delete a `Customer`, you may want to delete all associated `Order` entities.
- **Many-to-Many**: Cascading operations may be more complex here, and it's important to define cascading behavior carefully, as changes could potentially affect both entities involved in the relationship.

3. Hibernate CRUD Example Theory:

1.Understanding CRUD Operations in Hibernate:

1.Create(Insert): How to use Hibernate to insert records into a database.

To insert a record into the database, you need to create a `Session` object and call the `save()` or `persist()` method.

Example:

```
// Create an entity object
Employee employee = new Employee();
employee.setName("John");
employee.setSalary(5000);

// Open session
Session session = sessionFactory.openSession();
session.beginTransaction();

// Save the employee object (Insert)
session.save(employee);

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

// Close the session
session.close();
```

- `session.save()` inserts the entity into the database and generates the corresponding SQL `INSERT` statement.
- You can also use `session.persist()` to perform the same operation, though `save()` returns the generated identifier (ID).

2.Read(Select): Fetching data from the database using Hibernate.

To retrieve records from the database, you can use the `get()`, `load()`, or `createQuery()` methods to fetch data.

Example (Using `get()` method):

```
// Open session
Session session = sessionFactory.openSession();

// Fetch data (Select) by primary key
Employee employee = session.get(Employee.class, 1); // Fetch employee with ID
1

System.out.println(employee);

// Close session
session.close();
```

- `session.get()` retrieves the entity by its primary key. If no entity is found, it returns `null`.
- You can also use `session.createQuery()` for custom queries to fetch multiple records or complex conditions.

Example (Using `createQuery()` method):

```
// Open session
Session session = sessionFactory.openSession();

// Fetch data using HQL (Hibernate Query Language)
List<Employee> employees = session.createQuery("FROM Employee",
Employee.class).list();

for (Employee emp : employees) {
    System.out.println(emp);
}

// Close session
session.close();
```

3.Update: Modifying existing records in the database.

To update a record, you can load the entity from the database, modify its fields, and then call the `update()` method or simply merge the entity.

Example:

```
// Open session
Session session = sessionFactory.openSession();
session.beginTransaction();

// Fetch the entity you want to update
Employee employee = session.get(Employee.class, 1); // Fetch employee with ID
1
if (employee != null) {
    employee.setSalary(6000); // Update salary
    session.update(employee); // Update the entity in the database
}

// Commit the transaction
session.getTransaction().commit();
```

```
// Close session
session.close();
```

- `session.update()` updates the existing entity.
- Alternatively, you can use `session.merge()` if the object might be detached from the session, which will synchronize the state with the database.

4.Delete: Removing records from the database.

To delete a record, first, you load the entity, and then you can call the `delete()` method.

Example:

```
// Open session
Session session = sessionFactory.openSession();
session.beginTransaction();

// Fetch the entity to delete
Employee employee = session.get(Employee.class, 1); // Fetch employee with ID
1
if (employee != null) {
    session.delete(employee); // Delete the entity
}

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

// Close session
session.close();
```

- `session.delete()` removes the entity from the database.

2 Writing HQL (Hibernate Query Language):

1. ***Basics of HQL and how it differs from SQL.***

HQL (Hibernate Query Language) is a query language similar to SQL but it operates at the object level rather than the relational database level. It allows developers to query the database using Java objects instead of direct database tables.

Key Differences Between HQL and SQL:

- **Object-Oriented:** HQL queries are based on Java objects and their properties, while SQL works with database tables and columns.
- **No Need for Database Tables:** HQL refers to persistent Java classes (entities) and their attributes (properties), unlike SQL which refers to tables and columns.
- **Supports Polymorphism:** HQL can handle polymorphic queries, meaning it can query for subclasses and super classes in inheritance hierarchies.

- **Database Independence:** HQL abstracts the underlying database implementation, meaning it is more portable across different database systems.

HQL Syntax Example:

```
from Employee where salary > 50000
```

In SQL, the same query would look like:

```
SELECT * FROM Employee WHERE salary > 50000;
```

2. ***<u>How to perform CRUD operations using HQL.</u>***

CRUD stands for Create, Read, Update, and Delete. HQL allows you to perform these operations in a manner consistent with Java object manipulation.

Create (Insert)

To save an entity object to the database:

```
Session session = sessionFactory.openSession();
session.beginTransaction();

Employee employee = new Employee("John", 50000);
session.save(employee);

session.getTransaction().commit();
session.close();
```

Read (Select)

To fetch records based on a condition:

```
Session session = sessionFactory.openSession();
String hql = "FROM Employee WHERE salary > 50000";
Query query = session.createQuery(hql);
List<Employee> employees = query.list();
session.close();
```

Update

To update an existing record:

```
Session session = sessionFactory.openSession();
session.beginTransaction();

Employee employee = session.get(Employee.class, 1);  // Fetch employee with
id=1
employee.setSalary(60000);
session.update(employee);

session.getTransaction().commit();
session.close();
```

Delete

To delete a record:

```
Session session = sessionFactory.openSession();
session.beginTransaction();

Employee employee = session.get(Employee.class, 1);  // Fetch employee with
id=1
session.delete(employee);

session.getTransaction().commit();
session.close();
```

Notes on HQL for CRUD:

- **`save()`** and **`update()`** are for creating and updating objects.
- **`get()`** and **`list()`** are used for retrieving objects.
- **`delete()`** removes the entity from the database.

3. ***Introduction to the Criteria API for dynamic queries***

The **Criteria API** provides a programmatic way to create dynamic, type-safe queries in Hibernate. It is an alternative to HQL for constructing queries. The Criteria API is especially useful for building complex queries dynamically, such as when filters or conditions change at runtime.

Benefits of Criteria API:

- **Type-Safety:** It avoids errors due to incorrect property names by using Java objects and their attributes.
- **Dynamic Queries:** You can build queries at runtime based on conditions that are not known at compile time.
- **Abstraction:** It allows you to avoid writing raw SQL or HQL, making the code cleaner and more maintainable.

Example of Criteria API Usage:

1. **Creating a Criteria Query:**

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);

// Adding criteria
criteria.add(Restrictions.gt("salary", 50000));  // salary > 50000

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

```
session.close();
```

2. **Using Projections (Aggregation):**

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.setProjection(Projections.avg("salary"));  // Average salary
Double avgSalary = (Double) criteria.uniqueResult();
session.close();
```

3. **Combining Multiple Restrictions:**

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.gt("salary", 50000)); // salary > 50000
criteria.add(Restrictions.like("name", "J%")); // name starts with "J"

List<Employee> employees = criteria.list();
session.close();
```