Introduction to Hibernate (ORM Framework)

What is Hibernate?

○ A Java **ORM (Object-Relational Mapping)** framework. ○ Maps Java classes to database tables and Java objects to table rows.

Why Hibernate?

- Built as an alternative to **JDBC** (reduces boilerplate code).
- Simplifies database operations by handling SQL internally **Key Advantages**

Over JDBC:

- No need to write repetitive JDBC code (e.g., connections, result sets).
- Automatic mapping of objects to tables.
- Supports caching, transactions, and lazy loading.

Hibernate as an ORM Framework

- **Key Features of Hibernate**:

1. **ORM Framework**: Maps Java classes ↔ Database tables.
2. **Open Source**: Freely available, modifiable, and widely supported.
3. **JDBC Alternative**: Simplifies database interactions.
4. **Mapping**:

- Class → Table
- Object → Row
- Fields → Columns

5. **Simplified Code**: Minimal boilerplate for CRUD operations.
6. **Performance Optimization**:
    - Caching mechanisms (**First-Level** and **Second-Level Cache**).
    - Lazy loading (load data only when needed)
7. **Automatic Table Generation**: ○ Hibernate can **auto-create database tables** from entity classes using annotations

(e.g., `@Entity`, `@Table`).

8. **Exception Handling**: ○ Converts JDBC's checked exceptions (e.g., `SQLException`) to **unchecked exceptions** (e.g., `HibernateException`).

○ Example: Deleting a `null` object throws `IllegalArgumentException`, not `NullPointerException`.

9. **Query Capabilities**:

○ **HQL (Hibernate Query Language)**:

Database-agnostic language for complex operations (joins, aggregations).

Native SQL: Execute DB-specific queries with **Advantages Over JDBC**:

No manual handling of `ResultSet` or SQL syntax differences.

10. **Integration with JPA (Java Persistence API)**:

Hibernate is a **JPA-compliant** framework.

Uses standard JPA annotations (e.g., `@Entity`, `@Id`, `@OneToMany`).

11. **Simplified CRUD Operations**:

Perform database operations using objects (instead of raw SQL).

Explain JPA

- Enables developers to **map, store, update, and retrieve data** between **relational databases** (e.g., MySQL, PostgreSQL) and **Java objects** (and vice versa).
  - Simplifies database interactions by abstracting low-level SQL/JDBC code.
  - **Map**: Define relationships between Java classes and database tables using annotations (e.g., `@Entity`, `@Table`).
  - **Store**: Save Java objects to the database (e.g., `entityManager.persist(object)`).
  - **Update**: Modify existing database entries via object changes (e.g., `entityManager.merge(object)`).

- **Retrieve**: Fetch data from the database as Java objects (e.g., `entityManager.find(Employee.class, id)`).

Steps to Create a Maven Project

1. **Create Project**:
   - Open **Project Explorer** (in IDEs like Eclipse).
   - Press **Ctrl+N** (or use the IDE's menu for creating a new project).
   - Search for **"Maven"** in the wizard.
   - Select **Maven Project** → Click **Next**.
   - Check **"Create a simple project"** (skip archetype selection).
   - Click **Next**.
   - Enter:
     - **Group Id**: Your base package name (e.g., `com.example`).
     - **Artifact Id**: Your project name (e.g., `demo-app`).
   - Click **Finish**. The Maven project will be generated.
   - **Add Hibernate and MySQL Dependencies**:
     2. Go to **https://mvnrepository.com** (corrected URL).
     3. Search for:
        - **Hibernate Core**: Use version `5.6.10.Final`
        - **PostgreSql jdbc :**

Creating a `persistence.xml` File

1. **Purpose**:
   - A configuration file (XML format) for JPA to define database connections, entity mappings, and persistence settings.
2. **Steps**:
   - **Right-click** your project's source folder (e.g., `src/main/resources`).
   - Select **New** → **File** (or press **Ctrl+N**).
   - Create a folder named `META-INF`.

- Inside `META-INF`, create a file named `persistence.xml`

What is a JPA Entity Class?

An entity class is a **POJO (Plain Old Java Object)** annotated with `@Entity`. It represents a table in a relational database, where each instance of the class corresponds to a row in that table.

Requirements for Entity Classes

`1.@Entity` **Annotation**:

The class **must** be annotated with `@Entity` (`javax.persistence.Entity` or `jakarta.persistence.Entity`).

2.'No-Argument Constructor:

A **public or protected** no-argument constructor (not "publication protected") is required.

3.Non-Final Class and Members:

- The class and its fields **must not be declared** `final` (to allow proxy generation for lazy loading).

4.Encapsulation:

- Fields are typically `private` with **getter and setter methods**

5.Primary Key:

At least one field must be annotated with `@Id`.

Use `@GeneratedValue` to auto-generate primary keys.

Performing CRUD Operations

1) EntityManagerFactory (EMF):

   Creates `EntityManager` instances. Configured via `persistence.xml`

2) `EntityManager`

   Manages entity lifecycle (create, read, update, delete)

3) EntityTransactions:

   Wrap operations in transactions using `em.getTransaction().begin()` and `commit()`

EntityManagerFactory (EMF)

- Interface in the `javax.persistence` package.
- Creates a connection between a Java application and the database.
- Handles "Load or Register Driver"

EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("persistence_unitname");

`Persistence` Class:

- Helper class in `javax.persistence`.
- `createEntityManagerFactory()` method initializes EMF.

Key Points:

1. EMF objects are created via `createEntityManagerFactory()`.
2. The input to `createEntityManagerFactory()` is the **name of the persistence unit name** (e.g., `persistenceunit`).
3. EMF provides `EntityManager` instances for database operations

EntityManager

- An interface in the `javax.persistence` package used to perform **CRUD operations**.
- Manages entity lifecycle (e.g., persisting, querying, updating, deleting entities).
  - `EntityManager entityManager = entityManagerFactory.createEntityManager();`

- **Steps**:
    - Use the `EntityManagerFactory` to create an `EntityManager`.
    - `createEntityManager()` returns an `EntityManager` object.

EntityTransaction

- Interface in the `javax.persistence` package used to **manage database transactions**.
- `EntityTransaction entityTransaction = entityManager.getTransaction();`
- **Non-select operations** (e.g., INSERT, UPDATE, DELETE) **require an active transaction**

Core Methods in Hibernate/JPA

`persist(Object entity)`

- **Purpose**: Saves a new entity into the database.

`remove(Object entity)`

- **Purpose**: Deletes an entity from the database

`find(Class<T> entityClass, Object primaryKey)`

- **Purpose**: Retrieves an entity by its **primary key**.

`merge(T entity)`

- **Purpose**: Updates or inserts an entity.

`createQuery(String jpql)`

- **Purpose**: Executes a custom JPQL (Java Persistence Query Language) query

@OneToOne Mapping

Unidirectional

- One entity points to another using `@OneToOne`.
- A foreign key is created in the owning entity

```
@Entity

public class User {

    @Id

    private int id;



    @OneToOne

    @JoinColumn(name = "profile_id")

    private Profile profile;

}
```

- In the DB, `User` table will have a `profile_id` column (foreign key).
- **Profile** doesn't know about **User**.

Bidirectional

- Both entities are aware of each other.
- One side is **owning**, other side uses `mappedBy`

```
@Entity

public class User {

    @Id

    private int id;



    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL)

    private Profile profile;

}
```

```
@Entity

public class Profile {

    @Id

    private int id;



    @OneToOne

    @JoinColumn(name = "user_id")

    private User user;

}
```

Use `mappedBy` on the non-owning side to avoid duplicate foreign keys.

@OneToMany Mapping

Unidirectional

- One entity has a collection of another entity.
- Uses `@JoinColumn` since only one side knows the relation.

```
@Entity

public class Department {

    @Id

    private int id;

    @OneToMany

    @JoinColumn(name = "dept_id")  // column in Employee table

    private List<Employee> employees;

}
```

- Only `Department` knows about `Employee`.
- Foreign key `dept_id` will be in `Employee` table

Bidirectional

- Both entities are aware of each other.
- Use `mappedBy` in the parent entity

```
@Entity

public class Department {

    @Id

    private int id;



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

    private List<Employee> employees;

}



@Entity

public class Employee {

    @Id

    private int id;



    @ManyToOne

    @JoinColumn(name = "dept_id")

    private Department department;

}
```

`Employee` is the owning side with foreign key

@ManyToOne Mapping

- Many child entities are linked to one parent.
- Always **owning side** — must use `@JoinColumn`.

```
@Entity

public class Employee {

    @Id
```

```
    private int id;


    @ManyToOne
    @JoinColumn(name = "dept_id")
    private Department department;
}
```

- Creates a `dept_id` foreign key in `Employee` table.
- Common in bidirectional `@OneToMany` and `@ManyToOne`.

Mapping Type	Direction	Owning Side	Annotation Notes
@OneToOne	Unidirectional	Entity with FK	Use `@JoinColumn`
@OneToOne	Bidirectional	One with `@JoinColumn`	Other side uses `mappedBy`
@OneToMany	Unidirectional	Collection holder	Use `@JoinColumn` on parent
@OneToMany	Bidirectional	Child (with @ManyToOne)	Parent uses `mappedBy`
@ManyToOne	Always Owning	Child entity	Must use `@JoinColumn`

@ManyToMany

The `@ManyToMany` annotation is used in Hibernate when each entity can relate to multiple instances of the other entity**.**

- A **Student** can enroll in **multiple Courses**
- A **Course** can have **multiple Students**

```
@Entity
public class Student {
    @Id
    private int id;
    private String name;


    @ManyToMany
    @JoinTable
```

```
    private List<Course> courses;
}
@Entity
public class Course {
    @Id
    private int id;
    private String title;
}
```

Bidirectional ManyToMany

Both entities **are aware of the relationship**

```
@Entity
public class Student {
    @Id
    private int id;
    private String name;

    @ManyToMany
    @JoinTable
    private List<Course> courses;
}
@Entity
public class Course {
    @Id
    private int id;
    private String title;
```

```
    @ManyToMany(mappedBy = "courses")

    private List<Student> students;

}
```

- `Student` is still the **owning side** and defines `@JoinTable`.
- `Course` is the **inverse side** and uses `mappedBy = "courses"`.
- Only the owning side manages the association table

`@Entity`

- Marks a Java class as a **persistent entity**.
- Maps the class to a table in the database.
- Every entity must have a **primary key**

`@Id`

- Specifies the **primary key** of the entity.
- Must be applied to a single field/property per en

`@GeneratedValue`

- Used with `@Id` to **auto-generate primary key** values.
- Strategies:
  - `AUTO`: Default, lets provider choose strategy.
  - `IDENTITY`: Uses DB auto-increment column.
  - `SEQUENCE`: Uses database sequence.
  - `TABLE`: Uses a separate table for ID generation.

`@Table`

- Used to specify table-level details like:
  - Custom table name.
  - Unique constraints.
  - Indexes.
- Optional; if not provided, the table name defaults to the entity class name.

`@Column`

- Used to customize how a field maps to a database column.
- Common attributes:
  - `name`: Column name.
  - `nullable`: Allows null values or not.
  - `length`: Sets max length for `String` columns.
  - `unique`: Enforces uniqueness.

`@SequenceGenerator`

- Used with `@GeneratedValue(strategy = SEQUENCE)` to define a **database sequence**.
- Attributes:
  - `name`: Generator name.
  - `sequenceName`: Actual DB sequence name.
  - `allocationSize`: Increment size (default 50, often set to 1).

`@JoinColumn`

- Defines the **foreign key column** for associations like `@OneToOne` or `@ManyToOne`.
- Attributes:
  - `name`: Name of the foreign key column.
  - `referencedColumnName`: Column in the target table being referenced (default is `id`).

`@JoinTable`

- Used in `@ManyToMany` and some `@OneToMany` mappings.
- Specifies the **join table** that connects two entities.
- Attributes:
  - `name`: Join table name.
  - `joinColumns`: Foreign key column for the current entity.
  - `inverseJoinColumns`: Foreign key column for the other entity.

`OneToOne`

- Defines a one-to-one association between two entities.
- Owning side uses `@JoinColumn`.
- Can be **unidirectional** or **bidirectional**

`@OneToMany`

- Defines a one-to-many association (e.g., Department → Employees).
- Usually mapped by a collection (e.g., `List`, `Set`).
- Requires `mappedBy` on the inverse side in **bidirectional** mappings.

`@ManyToOne`

- Defines a many-to-one association (e.g., Employees → Department).
- Usually the **owning side** of `@OneToMany`.

`@ManyToMany`

- Defines a many-to-many association (e.g., Students ↔ Courses).
- Requires `@JoinTable` on the **owning side**.
- `mappedBy` is used on the inverse side in bidirectional mappings.

What is Cascading in Hibernate?

Cascading in Hibernate means that **operations on one entity can automatically propagate to related entities**. This is particularly useful in **parent-child relationships**, so you don't have to manually perform the same operation on all associated objects.

If you save/delete the parent, Hibernate will automatically save/delete the child too — based on the cascade type.

Where is Cascade Used?

Cascade is used in **association annotations** like:

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

CascadeType.PERSIST

- Saves parent and associated child entities automatically.

CascadeType.MERGE

- Updates both parent and child.

CascadeType.REMOVE

- Deletes both parent and child from DB.

CascadeType.ALL

- It includes **ALL** the above type

CascadeType	Description
`ALL`	Applies all cascade operations (`PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`)
`PERSIST`	When the parent is persisted (saved), the child is also persisted
`MERGE`	When the parent is merged (updated), the child is also merged
`REMOVE`	When the parent is removed (deleted), the child is also removed
`REFRESH`	When the parent is refreshed from DB, the child is also refreshed
`DETACH`	When the parent is detached from persistence context, the child is also detached

What is Fetching in Hibernate?

- **Fetching** is the process of **loading associated entities or collections** when an entity is retrieved from the database.
- It controls **when and how related data is loaded** (eagerly or lazily).

What is `FetchType`?

- `FetchType` is an **enum** in JPA that defines the **fetching strategy** for associations.
- It has two values:
  - `EAGER`
  - `LAZY`

EAGER Fetching

- Loads related entities **immediately** with the main entity query (usually via a JOIN).
- Pros:
  - No extra query later; data ready immediately.
- Cons:
  - May fetch more data than needed → **performance overhead**.
  - Can lead to **Cartesian product** or heavy joins in complex queries.

LAZY Fetching

- Loads related entities **only when accessed** in the code.
- Pros:
  - Improves performance by loading data **only when necessary**.
- Cons:
  - Accessing lazy-loaded data outside of a session causes `LazyInitializationException`.
  - Requires open session or proper transaction management.

Relationship	DefaultFetchType
`@OneToOne`	EAGER
`@ManyToOne`	EAGER
`@OneToMany`	LAZY
`@ManyToMany`	LAZY