Hibernate Note for Interview   Hariprasanth H A

What is hibernate

Hibernate is an **Object-Relational Mapping (ORM)** framework for Java. It simplifies the interaction between **Java applications** and **relational databases** by mapping Java classes (objects) to database tables.

 Open-source framework: Developed by Red Hat, written in Java.

 Implements JPA (Java Persistence API): Though Hibernate existed before JPA, it now acts as a JPA provider.

 Eliminates boilerplate JDBC code: No need to write SQL for most CRUD operations.

 Supports complex mappings: Between Java objects and database tables, including associations (OneToMany, ManyToOne, etc.).

 Built-in caching mechanisms: First-level and second-level caching to improve performance.

 Database independence: Allows switching databases with minimal changes

How Hibernate Works**:**

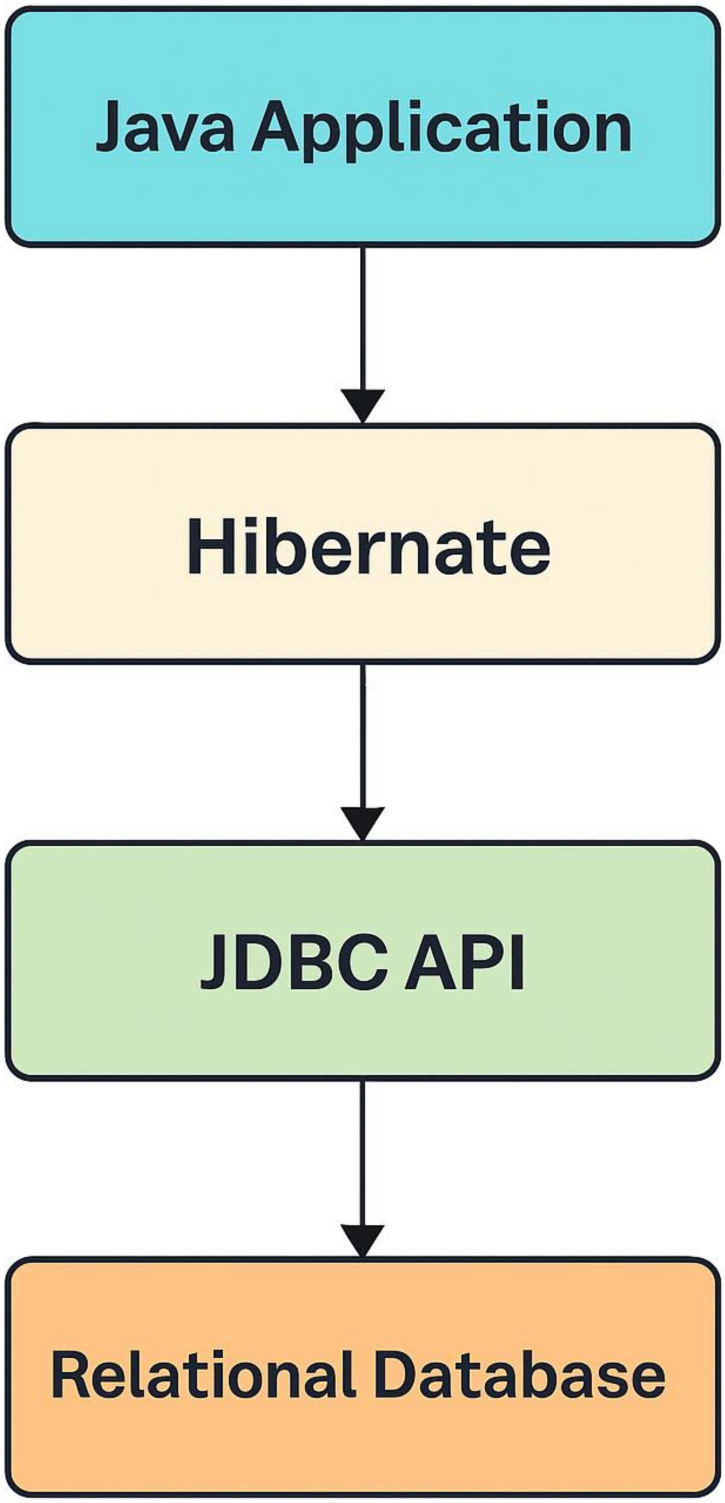
1. You create Java classes annotated with @Entity to represent tables.
2. Hibernate maps these entities to database tables using metadata (annotations or XML).
3. Use a **EntityManager** to interact with the database.

Features of Hibernate

Feature	Description
ORM Support	Maps Java objects to database tables
HQL	Hibernate Query Language (object-oriented SQL)
Automatic Table Creation	Can generate DB schema from entity classes
Lazy & Eager Loading	Fetch strategies for associated data
Caching	First-level (default) and second-level cache support

Why Use Hibernate?

- Avoids boilerplate JDBC code
- Handles database connections and transactions
- Makes switching databases easier
- Offers advanced features like caching, lazy loading, batch processing


Java Application
Hibernate
JDBC API
Relational Database

What is JPA?

JPA (Java Persistence API) is a **Java specification** for managing relational data in Java applications using **object-relational mapping (ORM)**.

 JPA is a specification, not an implementation.
It defines **APIs and annotations** to map Java classes to database tables.

 Requires a provider (implementation) to work — like **Hibernate**, EclipseLink, or OpenJPA.

 Manages data persistence using an EntityManager to perform operations like **insert, update, delete, and read**.

Common JPA Annotations:

Annotation	Purpose
@Entity	Marks a class as a database entity
@Id	Marks a field as the primary key
@Table	Specifies the table name
@Column	Maps field to column
@GeneratedValue	Auto-generates primary key
@OneToMany, @ManyToOne	Define relationships

How JPA Works:

1. You define **Java classes as entities** using annotations.
2. Configure persistence.xml file for DB connection and provider.
3. Use EntityManager to perform operations on the database.
4. JPA provider (like Hibernate) translates these into SQL and uses JDBC internally.

Advantages of JPA:

- Reduces boilerplate code
- Database-independent
- Manages complex relationships easily
- Integrates with frameworks like **Spring Boot**

What is FetchType in JPA?

In **JPA**, FetchType is an **enumeration** that defines how related entities should be **fetched (loaded)** from the database when you access an entity.

Purpose of FetchType

When you define relationships like @OneToMany, @ManyToOne, etc., you use FetchType to control **whether the related entities are fetched immediately or only when needed**.

Fetch Type	Description
EAGER	Loads the related entity **immediately**, along with the parent entity.
LAZY	Loads the related entity **only when accessed**, not at the time of fetch.

Why is FetchType important?

1. **Performance Optimization**
   - EAGER may cause **unnecessary database queries**.
   - LAZY allows **on-demand loading**, improving performance.
2. **Avoiding Memory Overhead**
   - Don't load huge related data unless needed.
3. **Avoiding Exceptions**
   - Lazy loading requires the entity to be accessed **within the same transaction/session**.

What is the difference between Eager Loading and Lazy Loading?

- **Eager Loading:**
  Eager loading is a data loading strategy that fetches related data and associated entities at the same time as the parent entity is loaded. This means all necessary data is retrieved in a single query (or as few queries as possible), reducing the need for additional database hits later.

  - Example: In Hibernate, using `fetch = FetchType.EAGER` on a relationship.

- **Lazy Loading:**
  Lazy loading is a strategy where related data is loaded only when it is actually accessed for the first time. The initial query fetches only the primary entity, and associated data is loaded on-demand, usually via additional queries.

  - Example: In Hibernate, using `fetch = FetchType.LAZY` on a relationship.

What is Hibernate Criteria API?

The Hibernate Criteria API is an advanced and object-oriented API provided by Hibernate for creating and executing database queries in a type-safe and dynamic way. Instead of writing SQL or HQL queries as strings, developers can use Java objects and methods to build queries programmatically. This increases code safety, readability, and maintainability.

- The Criteria API allows you to construct complex queries with restrictions (filters), projections (selecting certain columns), sorting, and joins.
- It is especially useful when query parameters are dynamic or unknown at compile time.
- In Hibernate 5 and later, the JPA Criteria API is recommended as the standard.

What is Primary Cache in Hibernate?

- **Primary Cache (First Level Cache) in Hibernate JPA:**
  The first level cache is associated with the EntityManager (Hibernate's Session). All entities loaded or saved within the same EntityManager instance are cached.
  - If you query the same entity twice in the same transaction, JPA returns the cached object instead of hitting the database again.
  - The cache is cleared when the EntityManager is closed.
  - Example:

    ```java
    EntityManager em = entityManagerFactory.createEntityManager();
    Employee emp1 = em.find(Employee.class, 1); // DB hit
    Employee emp2 = em.find(Employee.class, 1); // Cached
    ```

What is Secondary Cache in Hibernate?

The Secondary Cache (also known as the Second Level Cache) in Hibernate is a shared cache that stores entity data across multiple sessions and transactions. Unlike the first-level (primary) cache, which is bound to the lifecycle of a single EntityManager (or Hibernate Session), the second-level cache persists data beyond a single session and can be accessed by all sessions within the same application.
Key Points (JPA/Hibernate context):

- The second-level cache is optional and must be explicitly enabled and configured (e.g., using providers like Ehcache, Hazelcast, Infinispan).
- It helps reduce database access by caching frequently read entities, collections, or query results, improving application performance.
- It is configured at the session factory (or EntityManagerFactory) level and can persist cached data in memory or other external storage.

What is the difference between Hibernate and JPA?

- **JPA (Java Persistence API):**
  - JPA is a Java specification (part of Java EE/ Jakarta EE) that defines a standard for object-relational mapping (ORM) in Java.
  - It provides a set of interfaces and annotations for mapping Java objects to relational database tables.
  - JPA itself does not provide an implementation; it is just a set of guidelines and contracts.
- **Hibernate:**
  - Hibernate is a popular ORM framework for Java that implements the JPA specification.
  - In addition to supporting all JPA features, Hibernate provides extra features such as advanced caching, custom query languages (HQL), and proprietary APIs.
  - Hibernate can be used as a JPA provider, meaning you write JPA code and Hibernate does the heavy lifting behind the scenes.

What is the difference between JDBC and Hibernate?

- **JDBC (Java Database Connectivity):**
    - JDBC is a low-level Java API that enables direct interaction with relational databases using SQL queries.
    - Developers are responsible for managing database connections, writing SQL statements, handling result sets, and managing exceptions.
    - There is no automatic mapping between Java objects and database tables; all conversions must be done manually.
- **Hibernate:**
    - Hibernate is an Object-Relational Mapping (ORM) framework that abstracts away the low-level JDBC details.
    - It automatically maps Java classes to database tables (using annotations or XML) and handles CRUD operations using objects.
    - Hibernate provides advanced features like caching, lazy/eager loading, automatic transaction management, and query language (HQL/JPQL).
    - It reduces boilerplate code and improves productivity and maintainability.

What is the difference between EntityManager and EntityManagerFactory?

EntityManagerFactory:

- It is a heavyweight, thread-safe object used to create and manage multiple EntityManager instances.
- Usually created once per application (or per persistence unit) and shared.
- Responsible for setting up the underlying database connections, configuration, and caches.

EntityManager:

- It is a lightweight, non-thread-safe object used to manage the persistence operations (CRUD) for entities.
- Should be created, used, and closed per transaction or per request (short-lived).
- Handles the life cycle of entities (persist, find, remove, merge, etc.) and manages the first-level cache.

What is JPQL?

JPQL (Java Persistence Query Language) is an object-oriented query language defined as part of JPA.

- Unlike SQL, which works directly with database tables and columns, JPQL queries work with Java entity objects and their fields/properties.
- JPQL is database-independent and lets you query, update, and delete entity objects in a type-safe way.

How to Write a Custom Query in JPQL

- You can write custom JPQL queries using the @Query annotation (in Spring Data JPA) or with EntityManager.createQuery() in standard JPA.
- Syntax is similar to SQL, but you use entity class names and property names.

JPQL Query with EntityManager

```
String jpql = "SELECT e FROM Employee e WHERE e.department = :dept";
List<Employee> employees = entityManager.createQuery(jpql, Employee.class)
    .setParameter("dept", "IT")
    .getResultList();
```

JPQL Query with @Query Annotation (Spring Data JPA)

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    @Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")
    List<Employee> findEmployeesWithSalaryGreaterThan(@Param("minSalary") double minSalary);
}
```

- Use entity names and field names, not table or column names.
- Supports SELECT, UPDATE, DELETE operations.
- Supports joins, grouping, ordering, subqueries, etc.

What is EntityTransaction?

EntityTransaction is a JPA interface used to manage transactions in a Java application when working with a resource-local EntityManager (typically outside of a Java EE container or when not using JTA).
Key Points:

- It allows you to explicitly begin, commit, or roll back a transaction when performing database operations with JPA.
- It is typically used in standalone Java SE applications or environments where you control transaction boundaries manually.
- You obtain an EntityTransaction from an EntityManager using em.getTransaction().

Annotation

@Entity

- Marks a Java class as a JPA entity, meaning it is mapped to a database table.

```
@Entity
public class Employee { ... }
```

@Id

- Specifies the primary key field of the entity.

```
@Id
```

```
private Long id;
```

@GeneratedValue

- Specifies how the primary key value is automatically generated (e.g., AUTO, IDENTITY, SEQUENCE, TABLE).

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

@Table

- Specifies the table name in the database for the entity (optional, defaults to class name).

```
@Entity
@Table(name = "employees")
public class Employee { ... }
```

@Column

- Specifies the properties of a mapped column (name, length, nullable, etc.) in the table.

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

@OneToOne

- Defines a one-to-one relationship between two entities.

```
@OneToOne
private Address address;
```

@OneToMany

- Defines a one-to-many relationship (one parent, many children).

```
@OneToMany(mappedBy = "employee")
private List<Task> tasks;
```

@ManyToOne

- Defines a many-to-one relationship (many entities belong to one parent).

```
@ManyToOne
private Department department;
```

@ManyToMany

- Defines a many-to-many relationship between entities.

```
@ManyToMany
private List<Project> projects;
```

- **@JoinColumn**:
  Used for specifying the column in the child table that acts as a foreign key referencing the parent table. Commonly used with @ManyToOne and @OneToOne.

- **@JoinTable**:
  Used for mapping association tables in many-to-many (and sometimes one-to-many) relationships, allowing you to specify both join and inverse join columns.

What is Cascading in JPA/Hibernate?

Cascading refers to the automatic propagation of certain operations (like persist, merge, remove, etc.) from a parent entity to its related (child) entities.
When you perform an operation on a parent entity, the same operation is automatically applied to its associated entities, based on the cascade type specified.

Why use cascading?
It simplifies code and ensures consistency when dealing with entity relationships (e.g., saving an Order should also save its OrderItems).

How to Write SQL in Hibernate/JPA

Native SQL Queries

- Use EntityManager.createNativeQuery() to execute raw SQL statements.
- Useful when JPQL is not flexible enough or for complex database-specific operations.

Example

```
String sql = "SELECT * FROM employees WHERE salary > ?";

List<Object[]> results = entityManager.createNativeQuery(sql)

    .setParameter(1, 5000)

    .getResultList();
```

Example

```
List<Employee> employees = entityManager.createNativeQuery(
    "SELECT * FROM employees WHERE department_id = :deptId", Employee.class)
    .setParameter("deptId", 10)
    .getResultList();
```

- Use native SQL for advanced queries or database-specific features.
- Prefer JPQL or Criteria API for object-oriented queries when possible.
- Always be cautious of SQL injection when using raw SQL.

What is mappedBy in Hibernate/JPA?

mappedBy is an attribute used in association annotations like @OneToMany and @OneToOne to define the inverse side of a bidirectional relationship between two entities.

- mappedBy is placed on the **inverse (non-owning) side** of the relationship.
- It tells Hibernate/JPA which field in the owner entity maps the relationship.
- The **owning side** contains the actual foreign key and does not use mappedBy.

Rules for Creating an Entity Class in JPA/Hibernate

Rule / Requirement	**Description**	**Example / Note**
Annotate with @Entity	The class must be annotated with @Entity.	@Entity public class Employee { ... }
Have a public or protected no-arg constructor	Required by JPA to instantiate objects via reflection.	public Employee() {}
Have a primary key with @Id	At least one field must be annotated with @Id to serve as the primary key.	@Id private Long id;
Must not be final	The entity class and its persistent fields should not be declared as final.	—
Fields should be private or protected	For encapsulation and proxying (recommended, but not mandatory).	private String name;
Should be a top-level class	Cannot be a non-static inner class.	—
Must not have arguments in entity annotation	The @Entity annotation only takes an optional name parameter (not required).	@Entity(name = "Employee")
Should not use transient for persistent fields	Any field marked as transient will not be persisted.	—
Must not have duplicate field and property names	Field and getter/setter method names should not clash.	—

Rule / Requirement	Description	Example / Note
Provide getter and setter methods	For all persistent fields (depending on access type).	public String getName() { ... }
Optional: Annotate with @Table	To specify a custom table name (optional, defaults to class name).	@Table(name =

```
import javax.persistence.*;


@Entity

@Table(name = "employees")

public class Employee {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    @Column(nullable = false, length = 100)

    private String name;


    public Employee() {} // No-arg constructor


    // Getters and Setters

    public Long getId() { return id; }

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

    public String getName() { return name; }

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

}

What is `persistence.xml`?

The `persistence.xml` file is a configuration file used in Java Persistence API (JPA) to define persistence units and configure database connection and JPA provider settings.
It is required for JPA-based applications (unless you use Spring Boot, which can auto-configure JPA).

- Located in `META-INF/persistence.xml` inside your project's classpath (usually `src/main/resources/META-INF/persistence.xml`).
- Declares one or more `<persistence-unit>` sections, each with its own connection and entity settings.
- Configures the data source, JPA provider, transaction type, entity classes, and additional properties.

Section	Description
`<persistence-unit>`	Defines a persistence unit; `name` attribute gives it an identifier.
`<provider>`	Specifies the JPA provider (e.g., Hibernate, EclipseLink).
`<class>`	Lists entity classes (optional if auto-scan is enabled).
`<properties>`	Database connection details and other JPA/Hibernate properties.

Why do we use the directory name `META-INF` in Java projects?

Purpose of `META-INF`

- `META-INF` is a **special directory** in JAR, WAR, and EAR files (Java archives) recognized by the Java platform.
- It is **not a filename**, but a **folder** (directory) at the root of the archive.
- Java tools and the JVM look in `META-INF` for configuration, metadata, and manifest files that control the behavior of the application or library.

Why must configuration files like `persistence.xml` go in `META-INF`?

- The Java specification (e.g., JPA for `persistence.xml`, Java SPI for `services/`) **requires** config files to be in `META-INF`.

- The Java runtime and frameworks are programmed to only scan `META-INF` for these files.
- If you put the file elsewhere, it **won't be detected** and your application may fail to start or work incorrectly.

What is a Persistence Unit in JPA?

A **persistence unit** is a logical grouping of all the JPA entity classes and configuration for a particular data source in a Java application.
It defines how and where your entity objects are stored in the database and how JPA interacts with the underlying data source.

- Defined in the `persistence.xml` file under the `<persistence-unit>` tag.
- Contains configuration details such as:
    - The list of managed entity classes
    - Database connection properties
    - JPA provider (e.g., Hibernate, EclipseLink)
    - Transaction type and other JPA/Hibernate properties
- Each persistence unit has a unique `name` attribute.
- You can have multiple persistence units in one application (e.g., for multiple databases).

Dependencies for a Hibernate JPA Project

JPA API

Hibernate Core (JPA Implementation)

Database JDBC Driver

Hibernate Validator (Optional, for Bean Validation)

What is `EntityManagerFactory` in JPA/Hibernate?

Definition

- **`EntityManagerFactory`** is a JPA interface used to create and manage multiple `EntityManager` instances.
- It acts as a factory for `EntityManager`, which is responsible for managing the persistence and retrieval of entities from the database.
- **Heavyweight, Thread-safe:**
  `EntityManagerFactory` is designed to be an expensive, heavyweight object. It is thread-safe and should be created once per application (or per persistence unit) and shared.

- **Lifecycle:**
  - Typically created at application startup.
  - Closed at application shutdown to release resources.
- **Configuration:**
  Created using `Persistence.createEntityManagerFactory("unitName")`, where `"unitName"` is defined in `persistence.xml`.
- **Produces `EntityManager`:**
  Use `entityManagerFactory.createEntityManager()` to obtain lightweight, non-thread-safe `EntityManager` instances for database operations (like CRUD).

Common Methods of `EntityManager`

Method	Purpose
`persist(Object entity)`	Makes a new entity instance managed and persistent. Inserts into the database.
`find(Class<T> entityClass, Object pk)`	Finds an entity by its primary key. Returns the entity or `null` if not found.
`merge(Object entity)`	Updates (merges) the state of the given entity into the current persistence context.
`remove(Object entity)`	Deletes the entity from the database.
`createQuery(String qlString)`	Creates a JPQL query. Returns a `Query` object.
`createNamedQuery(String name)`	Creates a query using a named JPQL query.
`createNativeQuery(String sql)`	Creates a query using native SQL.
`getTransaction()`	Gets the `EntityTransaction` object for transaction management (in RESOURCE_LOCAL mode).

What is the `Query` Interface in JPA/Hibernate?

The **`Query` interface** in JPA (Java Persistence API) and Hibernate represents a databas e query object. It provides methods to define, configure, and execute queries for retri eving or updating entities in the database.

- **Creation:**
  You obtain a `Query` instance using methods like `EntityManager.createQuery()`, `createNamedQuery()`, or `createNativeQuery()`.
- **Usage:**
  The `Query` interface allows you to:

  - Set query parameters (for dynamic queries)
  - Execute select, update, or delete operations
  - Get results as lists or single values
- **Common Methods of `Query`:**

Method	Purpose
`setParameter()`	Set a value for a named or positional query parameter
`getResultList()`	Retrieve the results as a list
`getSingleResult()`	Retrieve a single result (throws exception if not exactly one)
`executeUpdate()`	Execute an update or delete operation
`setMaxResults(int)`	Limit the number of results returned
`setFirstResult(int)`	Set the starting position of the first result (pagination

Life Cycle of a JPA Entity

PA Entity Life Cycle: Steps and Explanations

1. **New (Transient)**

  - The entity object is created in memory.

  - It is not associated with any persistence context or database.

2. **Managed (Persistent)**

  - The entity becomes managed by the persistence context.

  - Changes made to the entity will be tracked and synchronized with the database.

3. **Detached**

  - The entity is no longer managed by the persistence context.

  - Changes to the entity are not tracked or saved to the database automatically.

4. **Removed**

- The entity is marked for deletion.
- It will be deleted from the database when the transaction is committed.

What is a Persistence Context in JPA/Hibernate?

Definition:
A **persistence context** is a set of entity instances in which for any persistent entity identity, there is a unique entity instance. In simple terms, it is a managed workspace or a cache that keeps track of all entities being managed by the `EntityManager` in a given session or transaction.