**Hibernate : => To overcome Object-Relational Impedance Mismatch**

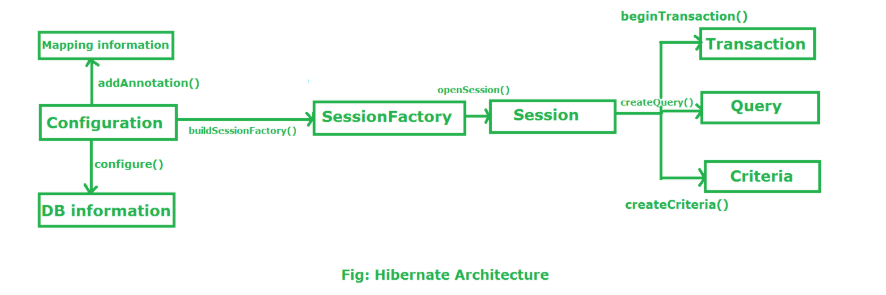
Hibernate is an Object relational mapping framework for java that simplifies the interaction between java applications and database objects

* It is an open source, lightweight and ORM tool
* Map java objects to database tables
* Automatically converts data from relational format to object-oriented format and vice versa

**Key advantages of hibernate**

* Reduces the boilerplate code of plain jdbc api because when we want to switch database we have to change underlying code of api also but hibernate is database independent
* Supports HQL (hibernate query language) – using it generates database independent query
* Supports Transaction, concurrency and automatic schema management
* First level and second level cache support ( default - first level cache)
* Auto schema management using hbm2ddl.auto (like create,update,delete)

**Arch. Of hibernate:**

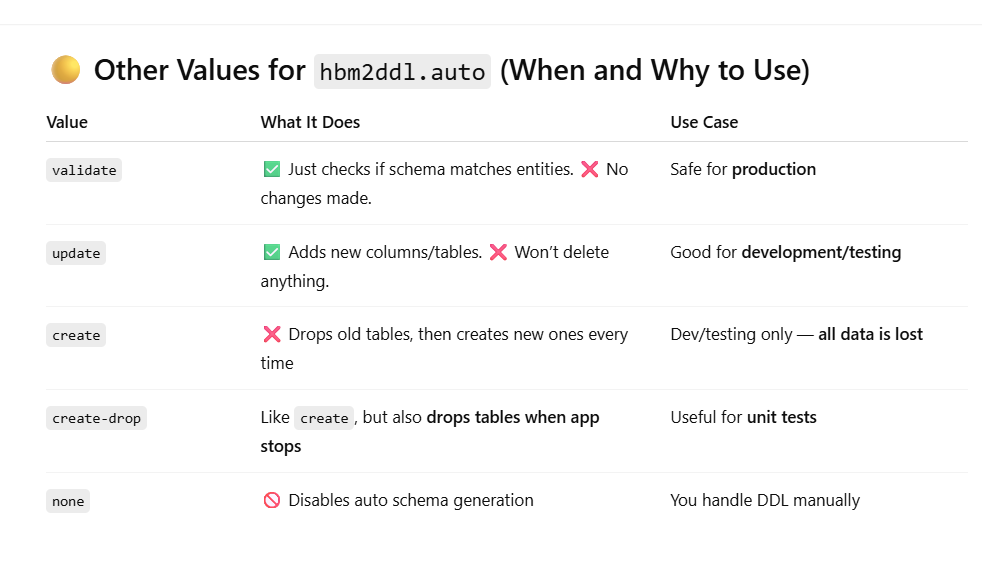


**Configuration:** loads all mappings and settings from **hibernate.cfg.xml** and annotations also

<property name="hbm2ddl.auto">update</property>

Hbm – hibernate mapping ddl -> data definition language

Update – when application starts, it matches the entity mapping class with database schema and update it without doing any delete operation



**What is hibernate.properties?**

* It's a **plain properties file** (key = value)
* Used to configure Hibernate in place of (or in addition to) hibernate.cfg.xml
* Typically placed in the **src/main/resources** directory (so it's on the class path)

They are two ways to configure the hibernate configuration

1. Hibernate.cfg.xml => supports advanced settings like multi-session factory and jndi, for complex mapping

Sessionfactory factory=new Configuration().configure().buildSessionfactory() //loads hibernate.cfg.xml file

Sessionfactory factory=new Configuration().configure(“hibernate.cfg.xml”).buildSessionfactory() //loads hibernate.cfg.xml file

Here mapping we will add in xml file using <mapping class=”com.example.employee”/>

1. Hibernate.Properties => simple and concise way of representing properties in key value pairs, limited to basic configuration.

Here we are mapping programmatically in code itself

Configuration cfg=new Configuration(); //Automatically loads hibernate.properties from src/main/resources folder

Cfg.addAnnonatedClass(Employee.class);

Sessionfactory=cfg.buildSessionFactory();

1. Instead of having these configuration files externally we also give them in programmatic way inside our code

**private** **static** SessionFactory *factory*;

**static** {

Configuration cfg=**new** Configuration();

cfg.setProperty("hibernate.connection.driver\_class", "com.mysql.cj.jdbc.Driver");

cfg.setProperty("hibernate.connection.url", "jdbc:mysql://localhost:3306/hibernateddb");

cfg.setProperty("hibernate.connection.username", "root");

cfg.setProperty("hibernate.connection.password", "root");

cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");

cfg.setProperty("hibernate.hbm2ddl.auto", "create");

cfg.setProperty("hibernate.show\_sql", "true");

cfg.setProperty("hibernate.format\_sql", "true"); //formats the sql in understandle way works when show\_sql is true

cfg.addAnnotatedClass(Employee.**class**);

ServiceRegistry registry=**new** StandardServiceRegistryBuilder().applySettings(cfg.getProperties()).build();

*factory*=cfg.buildSessionFactory(registry);

}

**public** **static** SessionFactory getSessionFactory() {

**return** *factory*;

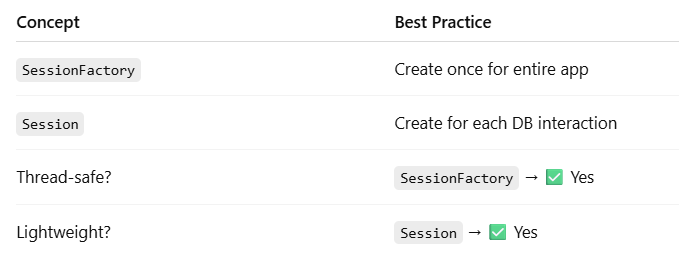
}

Service registry is a core component of hibernate it will access services like jdbc connections, transaction manager and caching. It will wire them together internally before building a session factory

* It will take all properties through configuration object and pass it to the registry builder.

**Session factory :** It is an interface to which provides factory methods to get the object of session. ( org.hibernate.sessionfactory)

* A **heavyweight**, thread-safe object that acts as a **factory for Session objects**.
* It is created during the application startup often in a static block or using singleton
* Used to read the config and db connections



* Yes — one SessionFactory per application (or per database) is the best practice.
* You're connecting to multiple databases in the same app
* You need different configurations for different modules (e.g., read-only replica DB)

**Session :** The session object provides an interface between the application and data stored in the database. These are short-lived objects and wraps the jdbc connection

* A **single-threaded, lightweight** object representing a **unit of work** (like a DB operation).
* It holds first level cache of data and it is a factory of query, transaction and criteria.
* Session interface provides various methods to create, update, delete the objects

**Transaction:** It is an interface to provide methods for transaction management.

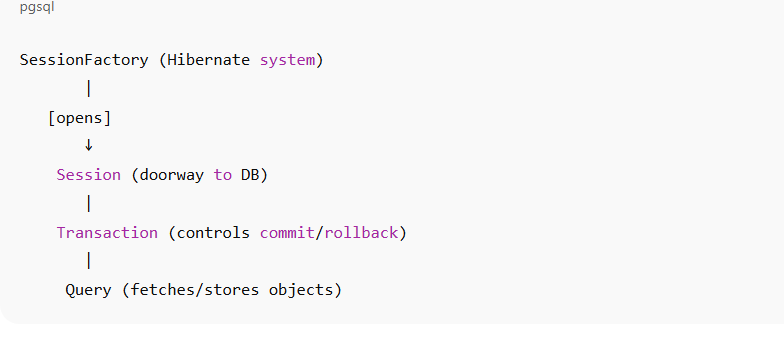
* A wrapper around a series of DB operations that ensures **ACID properties** (atomicity, consistency, isolation, durability).
* ✅ If all steps succeed → **commit ()**
* ❌ If any fails → **rollback()**

**Query :** A way to **retrieve or manipulate data** using:

* **HQL** (Hibernate Query Language)
* **Native SQL**
* **Criteria API**

Configuration class : org.hibernate.cfg.Configuration is a class in Hibernate used to:

* **Load** Hibernate configuration files
* **Set up** database properties
* **Register** annotated entity classes
* **Build** the SessionFactory



Annotations :

@Entity -> mark class as a persistent database object

@Table -> maps class to database table (optional)

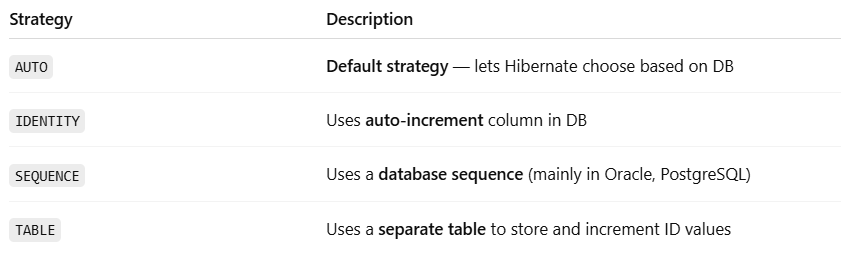
@Column -> maps the fields of the class as database table column

@Column(name = "employee\_name", length = 100, nullable = false) => doesnot allow null values in this column

private String name; // employee\_name varchar(255) not null

@Id -> specifies the primary key of the table

@GeneratedValue => auto generates the value for primary key



Auto => pick the strategy based on underlying dialect db , like mysql => identity , oracle.pl/sq/ uses sequence

Sequence => Uses a **DB sequence** to get the next ID. More efficient than IDENTITY because **it supports pre-allocation and batching**.

Table => Uses a **separate table** to simulate sequence generation. More portable (works on all DBs), but **slower** than others. (legacy db’s)

@Transient => ignores this field not saved to db (data is not inserted to db)

@Temporal => used with date to map time/date types

Relational Mapping :

In JPA, **one side of a relationship must be the "owning side"**, and the other is the "inverse side" (also called the mapped side).

@Joincolumn => used to map the foreign key column in relationships like onetoone and Manytoone

@OneToOne: One row in table A relates to one row in table B.

Example :

@OneToOne(cascade=CascadeType.***ALL***)

@JoinColumn(name="address\_id",referencedColumnName="id")

**private** Address address;

/\*

\* Employee has one address and address belong to only one employee

\* This column acts as a foreign key in employees table

\* @JoinColumn(name="address\_id",referencedColumnName="address.id")

\* name="address\_id" => name of this column in employees table

\* referencedColumnName="id" name of primary key in address table

\* \*/

@JoinColumn → Owning Side

In Employee, you define:

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "address\_id")

private Address address;

This side owns the foreign key, meaning: employees table will have an address\_id column.

Hibernate uses this column to manage the relationship. => "This side controls the DB join."

**mappedBy = "address" → Inverse Side**

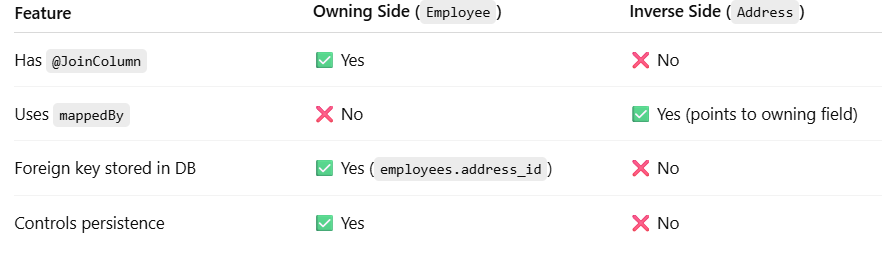
In Address:

@OneToOne(mappedBy = "address")

private Employee employee;

This is **not** the owning side. The other side (Employee.address) owns the relationship."

Hibernate uses this for **bidirectional navigation** only, not for managing the foreign key.



Cascade : what operations should be automatically applied on the associated entities while we are performing operations on owning entities. We can control this behaviour with the help of cascade types

A screenshot of a computer

AI-generated content may be incorrect.

@OneToMany: One row relates to many rows in another table. And @ManyToOne: Many rows relate to one row.

Example :

one employee can complete many tasks and many tasks can be pointed to same employee



@ManyToMany: Many-to-many link using join table.

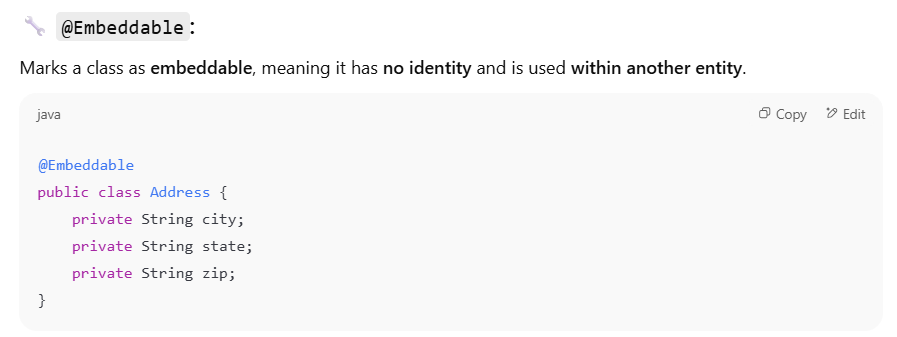
An Employee can have multiple Skills. => A single Skill can be possessed by multiple Employees

* **One record** in **Table A** can be related to **many records** in **Table B**.
* And **one record** in **Table B** can also be related to **many records** in **Table A**.
* Many-to-Many means **both sides can have many of each other**.
* A **join table** is used to maintain this mapping.



**@Embeddable & @Embedded:** These are used when **you want to include a reusable value object (not an entity)** inside another entity.

You have a common structure like an **Address** used in multiple entities.



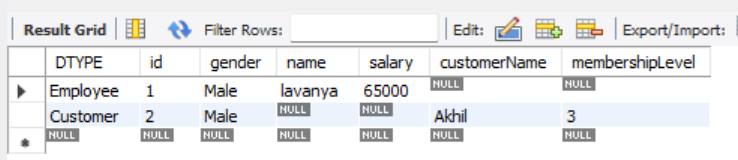
A screenshot of a computer program

AI-generated content may be incorrect.

**@Inheritance:** This is used when you have a **class hierarchy** (inheritance in Java) and want to map it to database tables

**@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)** =>

This strategy maps the **entire inheritance hierarchy to a single database table**.  
It introduces a special column called the **discriminator column** to indicate **which subclass** a particular row belongs to.



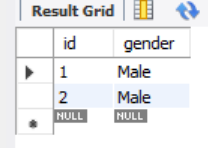
**@Inheritance(strategy = InheritanceType.JOINED)**

* Each class has its own table
* Subclass tables have a foreign key that refers to the primary key of Person.

Person => id | name

Employee => id (foreign key references to person.id) | name

Person



Employee

A screenshot of a computer

AI-generated content may be incorrect.

Customer

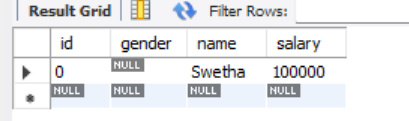
A screenshot of a computer

AI-generated content may be incorrect.

Here one foreign key is added to each and every column

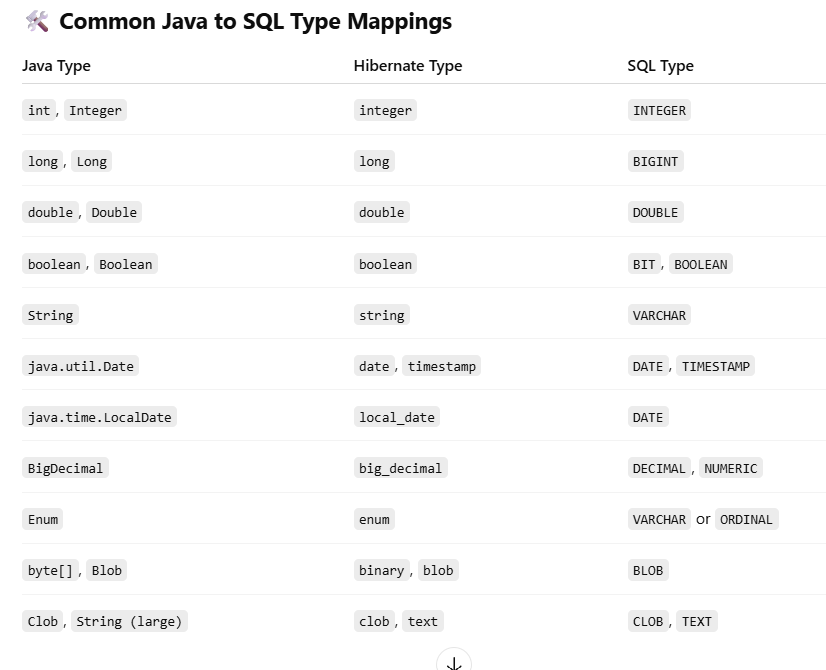
**@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)**

* **Each subclass gets its own full table**, including inherited fields.
* No common person table.

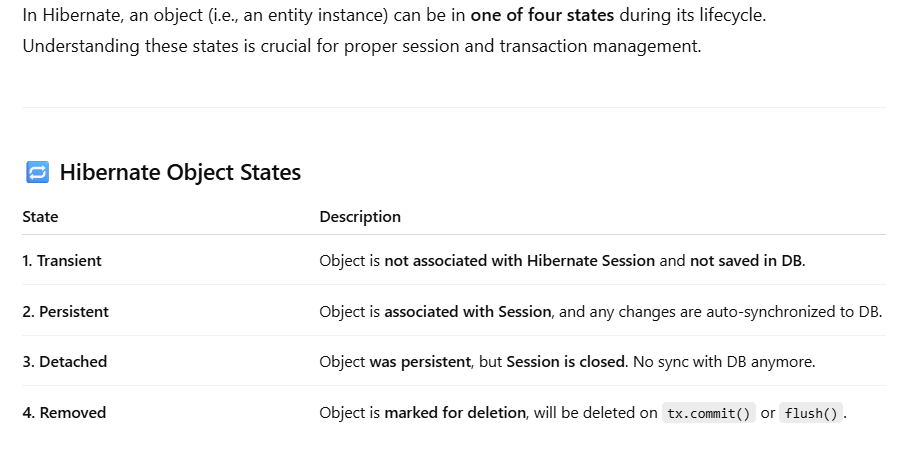


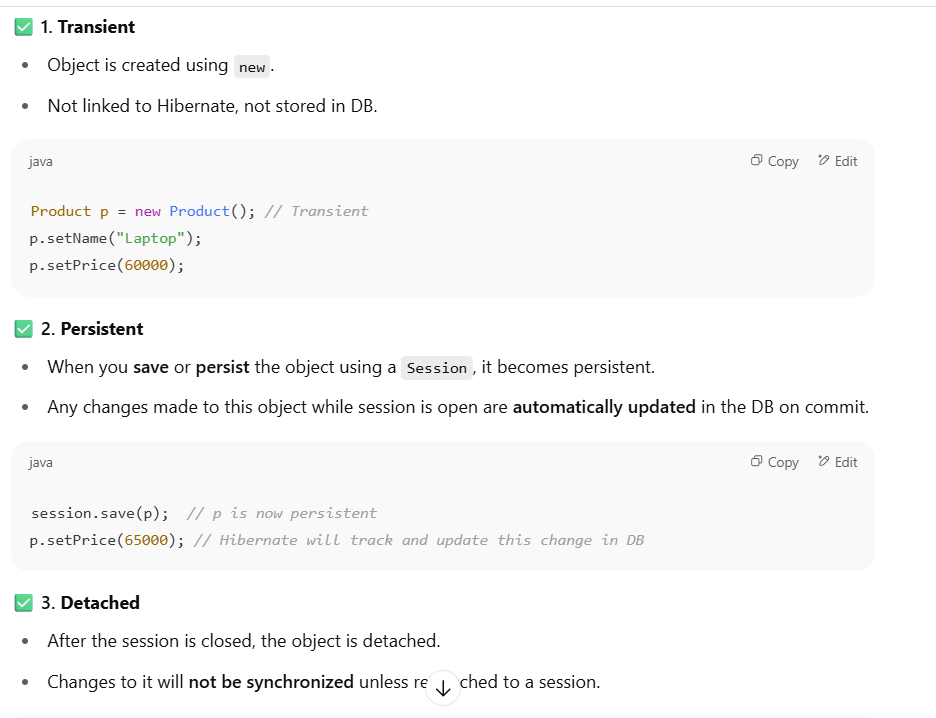
Gender is column from parent class employee. Here every table got all columns even from parent also

**Datatypes mapping from java to sql using hibernate**



**Hibernate life cycle**





A screenshot of a computer program

AI-generated content may be incorrect.

Crud operations hibernate:

**Save(Object obj )** : Which is used to save a new transient object to db. It returns an ID -> typically holds the primary key (optional to use)

* session.save(p) returns Serializable, which typically holds the primary key

**Persist(Object obj)** : Like save but it doesn’t return any id. Throws exception if used it outside a transaction

**Get(class,id)** : fetches the object egarly from db and return null if not found

**load(Class, id) =>** Returns a **proxy object** (lazy loading). **Throws ObjectNotFoundException** if entity not found when accessed.

**update(Object entity) =>** Use case: Bring a detached object back into persistent state.Error-prone: Throws exception if entity with same ID is already in session

**Throws exception (NonUniqueObjectException) if a different instance with the same identifier is already associated with the session.**

**merge(Object entity) =>** Use case: Safely copies detached object data into a managed instance. Returns a new managed copy of the detached object given.

**Safe to use** even if the same identifier is already in the session.

**delete(Object entity) =>** Use case: Delete a persistent object from DB.

****

Flush() and clear()

**Hibernate will automatically update the changes to db when flush()/commit() is called -> here object is in persistent state ( moved from transisent -> persistent )**

* Forces Hibernate to **flush changes** in memory (session cache) to the database **without committing**.
* Useful when you want the DB to reflect changes **before the transaction ends** (e.g., for constraints, triggers, or querying updated values).
* Doesn’t close the session.

session.flush(); // Pushes all pending changes to the DB

✅ Use when you want to ensure the DB is updated *before* the transaction is committed.

**Clear() :**

* Definition: Clears the first-level cache (session cache), effectively detaching all persistent objects.
* This is useful to free memory or avoid unintended updates if you don’t want Hibernate to auto-sync entities anymore.

| **Method** | **Purpose** |
| --- | --- |
| **flush()** | **Syncs session changes to DB immediately** |
| **clear()** | **Empties session cache (detaches all entities)** |

**Dirty Checking (Automatic Update)**

Hibernate detects field changes of persistent objects and auto-generates SQL UPDATE when transaction is committed or flush() is called.

Example:

--------------

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Product p = session.get(Product.class, 1); // Persistent

p.setPrice(9999); // Hibernate tracks this change

tx.commit(); // Hibernate fires: UPDATE Product SET price = 9999 WHERE id = 1

session.close();

You don’t need to call update () for persistent objects. Hibernate checks and updates dirty fields automatically.

Hibernate query language:

* **HQL (Hibernate Query Language)** is an **object-oriented** query language.
* It’s similar to SQL, but instead of querying tables and columns, you query **Java objects and their properties**.
* It’s **database-independent**, as it operates on entity classes, not tables.
* Supports **joins, subqueries, aggregations, group by, order by**, etc.
* Fully supports **polymorphism and inheritance**.

Native SQL : allows us to write plan sql instead of using HQL or criteria api

* We can use => session.createNativeQuery(sql, product.class);
* We can’t reuse it and complex to manage the joins as well
* Not database independent, not type safe and harder to maintain

NamedNativeQuery (Named sql query)

* **Uses native SQL**, i.e., raw SQL as you'd write in the database.
* Refers to **table names** and **column names**, not Java class/field names.
* You **must specify** the result mapping with resultClass or a @SqlResultSetMapping.

@Entity

@NamedNativeQuery(

name = "Product.findByName",

query = "SELECT \* FROM Product WHERE name = :name",

resultClass = Product.class

)

Class Product{}

NamedQuery (HQL ) :

* **Uses HQL (Hibernate Query Language)** – object-oriented and uses **entity names** and **Java field names**.
* Automatically maps to entity classes.
* More **portable** and database-agnostic.

@Entity

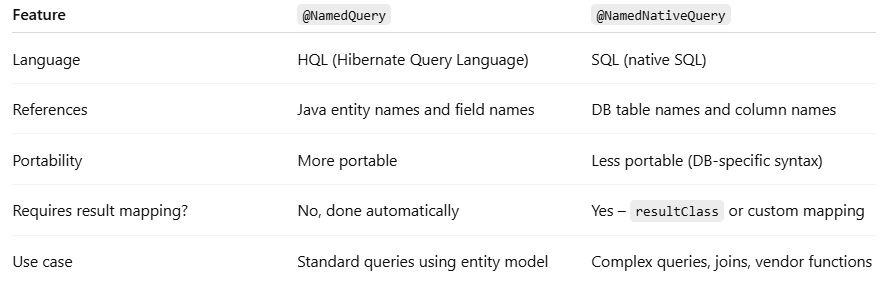
@NamedQuery(

name = "Product.findByPriceGreaterThan",

query = "FROM Product p WHERE p.price > :price"

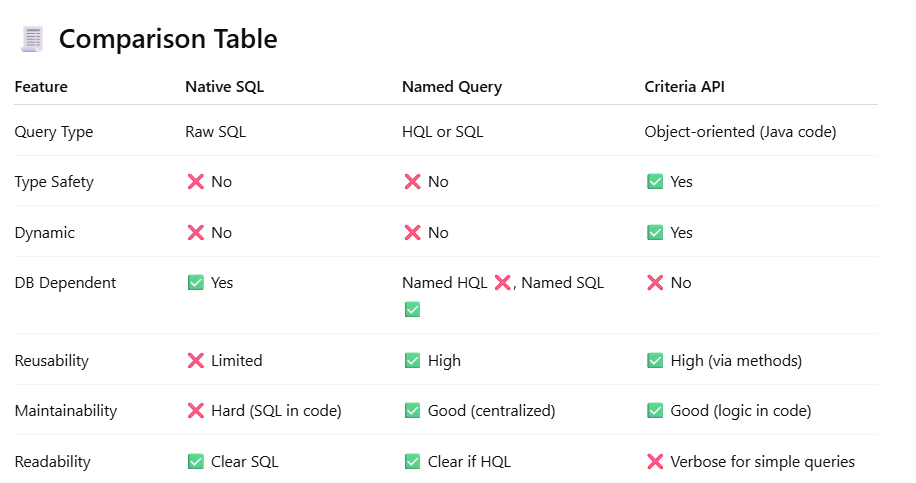
)

public class Product { ... }



CriteriaAPI :

* Criteria API provides a **type-safe**, **object-oriented**, and **dynamic** way of building queries using Java code
* **Dynamic queries** built based on runtime conditions.
* **Type-safe** development (compile-time checking).
* Ideal for complex search/filtering with multiple optional criteria.
* **Verbosity** — more boilerplate code compared to HQL/SQL.
* **Steeper learning curve** for beginners.
* It is harder to read than plain HQL for simple queries.



**Fetch types in hibernate:**

In Hibernate, fetch types determine when related entities or collections are loaded from the database.

1. LAZY Fetching

* Definition: Loads of related data only when accessed.
* Performance: More efficient for large data sets or when related data is rarely used.
* Default: For collections (@OneToMany, @ManyToMany).

**@OneToMany(mappedBy = "employee", fetch = FetchType.LAZY)**

**private List<Task> tasks; // Not fetched until getTasks() is called**

Use when:

* Related data is not always needed.
* You want better performance by loading minimal data.

**2.Eager Fetching**

* Definition: Loads related entities immediately with the main entity.
* Performance: Can cause unnecessary joins and performance issues.
* Default: For singular associations (@ManyToOne, @OneToOne).

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "employee\_id")

private Employee employee; // Immediately fetched when Task is loaded

⛔ Avoid eager fetching on collections, unless you absolutely need it.

| **Fetch Type** | **Load Time** | **Default For** | **Best Use Case** |
| --- | --- | --- | --- |
| LAZY | On demand | Collections | When data is not always needed |
| EAGER | Immediately | Single objects | When data is always required |

**Propagation Types:**

These control how transaction boundaries behave when a method is called from within another method that’s already in a transaction.

**✅ Propagation.REQUIRED (default)**

* Joins existing transaction if one exists, otherwise starts a new one.

👉 Use Case: Typical service method updates.

**✅ Propagation.REQUIRES\_NEW**

* **Always creates a new transaction**, suspends the current one.

👉 Use Case: Logging, auditing, partial rollback logic.

**✅ Propagation.NESTED**

* Starts a nested transaction using save points (if supported by DB).
* Allows partial rollback within outer transaction.

👉 Use Case: Rollback a portion of work without rolling back the entire transaction.

**✅ Propagation.SUPPORTS**

* Runs in a transaction **if one exists**, else runs non-transactionally.

👉 Use Case: Read-only methods that may or may not require consistency.

**✅ Propagation.NOT\_SUPPORTED**

* Always runs **non-transactionally**, suspends current transaction.

👉 Use Case: For large read-only or reporting queries that should not interfere with transactions.

**✅ Propagation.NEVER**

* **Throws exception** if a transaction exists.

👉 Use Case: Should never be called inside a transactional context.

**✅ Propagation.MANDATORY**

* **Requires** an existing transaction; throws exception if none.

👉 Use Case: Enforce transactional access for sensitive operations.

**Isolation Levels:** These control how concurrent transactions interact with shared data

**✅ Isolation.DEFAULT**

* Uses the **database’s default** isolation level.

**✅ Isolation.READ\_UNCOMMITTED**

* Lowest level: allows dirty reads, non-repeatable reads, phantom reads.

👉 Fast but unsafe.

**✅ Isolation.READ\_COMMITTED**

* Prevents dirty reads but allows non-repeatable reads & phantoms.

**✅ Isolation.REPEATABLE\_READ**

* Prevents dirty & non-repeatable reads but allows phantom reads.

👉 Use for high consistency in reads.

**✅ Isolation.SERIALIZABLE**

* Highest level: avoids dirty, non-repeatable, and phantom reads.
* Slower, locks entire range/table.

👉 Use for banking, strict consistency.

**What Is a Phantom Read?**

* A **phantom read** happens when:
  + Transaction A reads a set of rows matching some WHERE clause.
  + Transaction B inserts **new rows** that match the same WHERE clause and commits.
  + Transaction A **re-runs** the same query and sees **new rows** — i.e., the **"phantoms"**.

This **violates repeatability of a query result**, even if the **same rows aren't changed**.

**❗ N+1 Problem in Hibernate**

**🔍 What is it?**

The **N+1 select problem** is a **performance issue** that occurs when your application executes:

* **1 query** to retrieve a list of parent entities (like Author),
* followed by **N additional queries** — one for each parent — to fetch their associated child entities (like Book).

**Solution 1: JOIN FETCH in HQL**

**Purpose:** To load parent and child entities in a single SQL query using Hibernate Query Language (HQL).

****

**Solution 2: LAZY Fetching with Batch Fetching :** To retain LAZY loading (which is default for @OneToMany) but still reduce the number of queries using batching.

You tell Hibernate: "Instead of one SQL per child list, batch them in groups (like 10 at a time)."

****

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a web page

AI-generated content may be incorrect.**

**A screenshot of a white box

AI-generated content may be incorrect.**

**Pagination =>** In Hibernate, pagination is implemented using the setFirstResult() and setMaxResults() methods on a query. These are used to limit the number of records returned from the database, just like LIMIT and OFFSET in SQL**.**

To avoid fetching large datasets at once (e.g., thousands of rows), which:

* Wastes memory
* Slows down performance
* Increases response time

Query<Product> query = session.createNamedQuery("Product.findByPriceGreaterThan", Product.**class**);

query.setParameter("price", minPrice);

query.setFirstResult(0); //Limit

query.setMaxResultsN(2); //Offset

**return** query.getResultList();

**Caching :**

Hibernate uses **caching** to reduce database hits and improve performance by storing frequently accessed data in memory.

**🔹 1. First-Level Cache (L1)**

* **Scope**: Per Session (default behavior).
* **Enabled**: Always ON by default.
* **Description**:
  + Every Hibernate Session maintains its own cache of persistent objects.
  + When you call session.get() or load(), Hibernate first checks this cache.

Session session = factory.openSession();

Product p1 = session.get(Product.class, 1); // Hit DB

Product p2 = session.get(Product.class, 1); // Fetched from L1 cache

session.close(); // L1 cache destroyed

**2. Second-Level Cache (L2)**

* **Scope**: Across multiple sessions and sessions factories (optional).
* **Shared**: Yes, shared globally across sessions.
* **Providers**: EHCache, Infinispan, Redis, etc.
* **Requires Configuration**.

🧩 Use when:

* You frequently access the same entity in different sessions.
* Performance and read-heavy applications matter.



**Types of Caches**

**1. READ\_ONLY**

* **What:** Only allows reads, no updates.
* **Why:** Safe because data never changes, no concurrency issues.
* **When:** Perfect for **reference data**.

**✅ Real-world Example:**

* **Countries, States, Currencies**
* Product categories, age groups, static lookup tables

**2. NONSTRICT\_READ\_WRITE**

* **What:** Caching with no strict guarantees on consistency.
* **Why:** Faster but may serve stale data.
* **When:** Use when **stale data is acceptable**.

**✅ Real-world Example:**

* **Blog posts**, user bios, product descriptions updated infrequently

**3. READ\_WRITE**

* **What:** Uses a **timestamp-based** mechanism to maintain consistency.
* **Why:** Ensures data consistency between cache and DB.
* **When:** For data that is **read often and occasionally updated**.

**✅ Real-world Example:**

* Product inventory, Employee profiles.

**4.TRANSACTIONAL**

* **What:** Integrates with **JTA transactions**, maintains full ACID consistency.
* **Why:** Very strong guarantees, but **heavier**
* **When:** Use in **enterprise apps** with distributed transactions (e.g. banking)

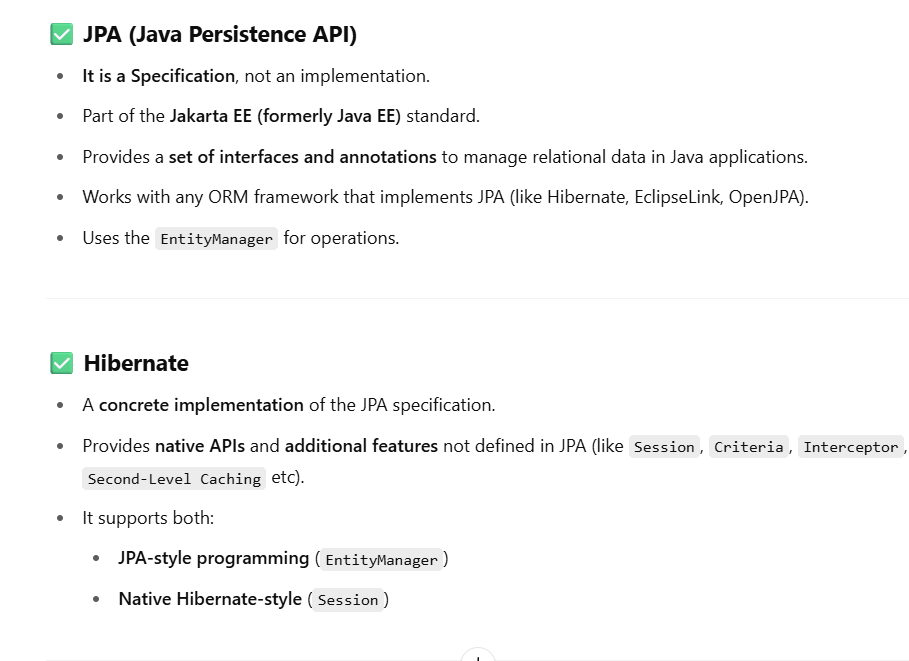
**3. Query Cache**

* **Caches result sets** of queries.
* **Useful for repeated queries** (e.g., select \* from Product where price > 1000).

A screenshot of a computer code

AI-generated content may be incorrect.

JPA VS HIBERNATE => JPA is interface and hibernate is a library (implements JPA )



A screenshot of a computer program

AI-generated content may be incorrect.

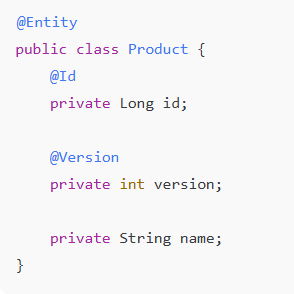
**Transaction management**

Concurrency issues arise when multiple transactions access the same data at the same time.

Hibernate offers two strategies:

**A. Optimistic Locking**

* **No actual DB lock is used**.
* Hibernate checks if the data has been modified before committing.
* You use a @Version field to track updates.
* Hibernate includes the version in UPDATE statements.
* If another transaction modified the row, your update will **fail with an exception** (typically OptimisticLockException).



**B. Pessimistic Locking**

* **Locks the row in the DB**, preventing others from modifying it until the transaction completes.
* Suitable for critical data where concurrent edits must be avoided.

A screenshot of a computer

AI-generated content may be incorrect.