

## Hibernate Introduction

### What is Hibernate?

- A Complete solution for managing **persistence** in Java.
- **ORM tool** (Object Relational Mapping) → maps Java objects (POJOs) to database tables.
- Provides **automatic & transparent persistence**.
- An **implementation of JPA** (Java Persistence API) specification.
- Open-source, lightweight, founded by **Gavin King (2001)**.
- Current versions: **Hibernate 5.x / 6.x/7.1.1**

### JPA vs Hibernate

- **JPA** → A specification (standard) under Jakarta EE. Defines interfaces only.
- **Hibernate** → A JPA **implementation** (actual working classes, provider of implementation JARs).

### Why Hibernate? (Compared to JDBC)

- With **JDBC**, developer manually manages SQL, connections, result sets.
- Hibernate abstracts that → focus on **data access logic** instead of SQL/DB management.
- Avoids boilerplate JDBC code, exception handling, and database vendor lock-in.

### Key Advantages of Hibernate

1. **Open-source & lightweight**
  - Easy to integrate, doesn't add heavy overhead.
2. **Fast performance via caching**
  - **First-level cache** → enabled by default (session-level).
  - **Second-level cache** → configurable.
  - **Query cache** → optional.
3. **Database-independent queries**
  - Uses **HQL (Hibernate Query Language)** or **JPQL**(Java Persistence Query Language)
  - If DB changes (MySQL → Oracle), no need to rewrite queries & change DAO layer.
4. **Automatic table creation**
  - Auto-generation based on entity mapping (property - hibernate.hbm2ddl.auto).

## 5. Simplified joins & associations

- Handles complex joins easily (One-to-One, One-to-Many, Many-to-Many).
- Example: Fetch all courses with students using JPQL

## 6. Inheritance support

- Supports mapping Java inheritance → DB tables (SINGLE\_TABLE, JOINED, TABLE\_PER\_CLASS).

## 7. Unchecked exceptions

- Converts checked SQLException → unchecked HibernateException.
- Reduces boilerplate try-catch code.

## 8. Automatic Primary Key generation

- @GeneratedValue(strategy=...) for IDs.

## 9. Caching mechanism

- Reduces DB round-trips, improves performance.

## 10. Annotation & XML support

- Can configure mappings via **annotations** or **hbm.xml**.

## 11. Dialect classes

- Hibernate automatically generates SQL suitable for the chosen database (MySQLDialect, OracleDialect, etc).

## 12. Pagination support

- Built-in APIs (setFirstResult(), setMaxResults()).

## 13. Statistics & monitoring

- Query performance and DB stats available via Hibernate API.

## 14. JDBC vs Hibernate DB Connection Handling

### JDBC (traditional):

You manually call `DriverManager.getConnection(...)` every time. , so each request creates a **new DB connection**. It is Very **expensive** because DB connections are heavyweight. This is **fixed, one-connection-per-request** behavior.

### Hibernate (ORM):

When you bootstrap Hibernate and call ,

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Hibernate reads `hibernate.cfg.xml` & internally configures a **connection (pool)**. It uses by default a very **basic internal connection pool** (not for production).Property:

```
<property name="hibernate.connection.pool_size">10</property>
```

Meaning, Hibernate keeps **up to maximum 10 JDBC connections ready**. In real-world apps, we will use external **connection pool providers**:- **C3P0, Apache DBCP or Hikari DBCP with Spring boot**. **Later** every time you call `sessionFactory.openSession()` or `getCurrentSession()`, Hibernate borrows a connection from the pool (instead of creating a fresh one). This benefits in to Huge **performance gain** compared to plain JDBC.

### Other popular ORM Frameworks

- **EclipseLink** (JPA Reference Implementation).
- **iBATIS / MyBatis** (SQL-centric ORM).
- **OpenJPA, Kodo, JDO** (less popular today).

### Summary -

Hibernate = **JPA implementor + ORM framework + persistence abstraction + caching + DB portability**.

### Important Hibernate Building Blocks

#### 1. `org.hibernate.Session`

- **Interface** (implementation in Hibernate core JAR).
- **Extends `jakarta.persistence.EntityManager`**
- Represents the **main runtime interface** between application and Hibernate.
- **Jobs:**
  - Provides CRUD APIs:
    - `save()`, `persist()`, `get()`, `load()`, `merge()`, `update()`, `delete()`, `createQuery()` ...
  - Acts as a **thin wrapper** around a *pooled JDBC connection*.
  - Maintains **L1 cache (Persistence Context)**. Entities in current session are cached automatically.
- **Characteristics:**
  - Lightweight, short-lived.
  - **One session per unit of work (request/transaction)** → DAO layer opens/closes it.
  - **Not thread-safe** → different client requests should use their own session.
  - No need for synchronization → each thread gets its own instance.

#### 2. `org.hibernate.SessionFactory`

- **Interface** (implementation in Hibernate core JAR).

- **Jobs:**
  - Bootstrap point → **provides Session objects** via `openSession()` / `getCurrentSession()`.
  - Encapsulates **database-specific configuration** including:
    - Connection pool
    - SQL dialect
    - Caching setup
- **Characteristics:**
  - **Heavyweight**, expensive to create. Create only **one singleton per DB**.
  - Immutable, inherently **thread-safe**.
  - Maintains **L2 cache** (must be configured explicitly, unlike L1 cache).

### 3. `org.hibernate.cfg.Configuration`

- Helper class for bootstrapping Hibernate.
- Reads **`hibernate.cfg.xml`** (or `hibernate.properties`) from **classpath**.
- Builds `SessionFactory`.
- Usage:
  - `Configuration cfg = new Configuration().configure();` // loads `hibernate.cfg.xml`
  - `SessionFactory sf = cfg.buildSessionFactory();`

### 4. Additional APIs

- **Transaction** → ensures **ACID compliance** for DB operations.
- **Query (HQL/JPQL)** → Object-oriented queries.
- **CriteriaQuery (JPA Criteria API)** → Type-safe query construction.

### 5. `hibernate.cfg.xml`

- Centralized XML-based configuration file.
- Defines:
  - DB connection details (URL, username, password).
  - Hibernate properties (dialect, caching, pooling, DDL).
  - Mappings (entity classes or `.hbm.xml` files).

**Important property in the hibernate configuration file , for auto schema generation**

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

Value	Behavior	When to use
<b>none</b>	(default if omitted) Hibernate does not manage schema at all.	Production (you manage schema manually).
<b>validate</b>	Hibernate only <b>validates schema</b> against entity mappings. Throws error if mismatch.	Safe for production if you want to ensure mapping = schema.
<b>update</b>	Hibernate <b>creates missing tables/columns</b> but does not drop or modify existing ones.	Development (keeps data while evolving schema).
<b>create</b>	Hibernate <b>drops and re-creates</b> the schema at startup. <b>All data lost.</b>	Unit tests or demo environments.
<b>create-drop</b>	Same as create, but also <b>drops schema when SessionFactory is closed.</b>	Temporary setups (JUnit tests).

## 6. HibernateUtils (to create Singleton instance of SessionFactory , per DB)

```
public class HibernateUtils {  
    private static SessionFactory factory;  
    static {  
        System.out.println("in static init block");  
        // 1. Create empty Configuration instance  
        // 2. Configure it (loads hibernate.cfg.xml)  
        // 3. Build SessionFactory  
        factory = new Configuration().configure() // loads xml-based props & mappings  
            .buildSessionFactory();  
    }  
    public static SessionFactory getFactory() {  
        return factory;  
    }  
}
```

- **Usage:** DAO layer creates sessions using `HibernateUtils.getFactory().getCurrentSession`

## 7. Hibernate managed POJO (Entity | Model) layer

- Legacy approach involved creating POJO.hbm.xml file per Entity. Leading to too much overheads.
- Modern favorite approach is using annotations

### POJO Annotations (package - jakarta.persistence)

Annotation	Purpose
@Entity	Mandatory at class level, marks persistent entity
@Id	Primary key , Unique Identifier
@GeneratedValue	Auto-generates PK (using strategy - IDENTITY, SEQUENCE, TABLE, AUTO)
@Column	Customize column name, length, constraints
@Transient	Skip persistence
@Temporal	Date/Time mapping for java.util.Date
@Lob	Blob/CLOB mapping
@Enumerated	Enum mapping (EnumType.STRING recommended)
@Table	Optional table name, schema, catalog

**@GeneratedValue** in Hibernate/JPA tells the persistence provider how to generate primary key values automatically. The **strategy attribute** determines the **generation strategy**. Here are the most common examples:

#### 1. AUTO

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

- The persistence provider (Hibernate) chooses the best strategy based on the database dialect.
- Example:
  - MySQL → Creates hibernate\_sequence table (emulated sequence)
  - Oracle → uses a sequence actually

#### 2. IDENTITY

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

- Relies on database identity column (AUTO\_INCREMENT in MySQL, IDENTITY in SQL Server).
- The database generates the value on insert.
- Hibernate does **not** pre-allocate IDs, so the entity must be inserted in database , to get the ID.

### 3. SEQUENCE

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user\_seq")

@SequenceGenerator(name = "user\_seq", sequenceName = "user\_sequence", allocationSize = 1)

- Uses a database sequence object.
  - sequenceName → actual DB sequence.
  - allocationSize → batch allocation (default 50 in Hibernate, 1 means increment by 1).
  - Works well with Oracle, PostgreSQL.
- 

### 4. TABLE

@Id

@GeneratedValue(strategy = GenerationType.TABLE, generator = "user\_table\_gen")

@TableGenerator(

name = "user\_table\_gen",

table = "id\_generator",

pkColumnName = "gen\_name",

valueColumnName = "gen\_val",

pkColumnValue = "user\_id",

allocationSize = 1

)

private Long id;

- Uses a separate table to generate IDs.
- Database-independent.
- Slower than IDENTITY or SEQUENCE, but works on all DBs.

## Extra Hibernate/JPA Annotations

Annotation	Purpose
<b>@CreationTimestamp</b>	Auto-populates date/time when the entity is first persisted (inserted).
<b>@UpdateTimestamp</b>	Auto-updates date/time whenever the entity is updated.
<b>@Version</b>	Enables optimistic locking using a version column (prevents lost updates).
<b>@Basic(fetch = FetchType.LAZY)</b>	Allows lazy fetching of large/basic attributes (e.g., @Lob).
<b>@ColumnDefault("value")</b> (Hibernate)	Sets a default column value in generated DDL.
<b>@UniqueConstraint</b> (inside @Table)	Enforces uniqueness across one or more columns.
<b>@Index</b> (Hibernate, inside @Table)	Adds a database index on specified columns.
<b>@NotNull, @Size, @Email</b> , etc. (Bean Validation)	Validate entity fields before persistence (integrates with Hibernate Validator).

After adding required annotations in Entity class , **Add mapping entry** in hibernate.cfg.xml:

```
<mapping class="Fully Qualified Entity class"/>
```

- This tells Hibernate to manage persistence for the User class.
- With hibernate.hbm2ddl.auto=create or update, the table will be created automatically.

## 8. Steps to Implement Hibernate-Specific DAO Layer

### 1. Create DAO Interface

- Define CRUD methods that your DAO will support.
- No Hibernate dependencies here; just Java interfaces.

Eg. public interface UserDao {

```
    void save(User user);  
    User getById(Long id);  
    void update(User user);  
    void delete(User user);  
    List<User> getAll();
```

```
}
```

## 2 Implement DAO using Hibernate API

- Uses SessionFactory, Session, Transaction, and Query.
- No instance variables needed.
- Each CRUD method manages its own **Session** and **Transaction**.

### Typical Dev steps

- Get Hibernate Session From Session Factory  
API of org.hibernate.SessionFactory  
public Session **openSession()** throws HibernateException  
OR  
public Session **getCurrentSession()** throws HibernateException
- Begin a Transaction  
API of org.hibernate.Session  
Public Transaction beginTransaction() throws HibernateException  
Here Hibernate transaction object is created which starts a DB transaction(setAutoCommit=false)
- Create a try block to add CRUD work  
Eg. Session.persist | session.get | session.update | session.delete
- Commit the transaction at the end of try block .  
API of org.hibernate.Transaction  
public void commit() throws HibernateException
- In catch block(RuntimeException) roll back the transaction  
API of org.hibernate.Transaction  
public void rollback() throws HibernateException
- In case of openSession , you will have to explicitly close the Session.
- In case of getCurrentSession, Session is implicitly closed upon transaction boundary(i.e. commit | rollback)
- Eg. For saving an entity to DB

```
public class UserDaoImpl implements UserDao {  
  
    @Override  
  
    public void save(User user) {  
  
        Session session = HibernateUtils.getFactory().getCurrentSession();  
  
        Transaction tx = session.beginTransaction();  
  
        try {  
  
            session.persist(user); // Hibernate-native API  
  
            tx.commit();  
  
        } catch (RuntimeException e) {  
  
            if (tx != null) tx.rollback();  
  
            throw e;}} }
```

## 9. Testing DAO

- Create a **main()** method or a JUnit test case
- Create DAO instance & Call DAO methods

### Note

- All Hibernate-specific operations (CRUD, JPQL queries, session management) are in **DAO implementation**.
- Entities (POJOs with JPA annotations) remain vendor-independent.
- The **service layer** can be optional. DAO can be called directly by Main class.

### Why Hibernate prefers wrapper types (Integer / Long) over primitive type (int/long) for ID field ?

- Wrapper types (Integer / Long) can be **null by default**.
- Hibernate uses **null as a marker to determine if an entity is transient** (not yet saved in DB).
  - If ID is null → entity is **new/transient**, needs insert.
  - If ID is non-null → Hibernate assumes entity is **persistent**, might try update.
- If you use primitives (int / long):
  - Default value is 0.
  - Hibernate may **misinterpret ID = 0 as already persistent**, causing **unexpected behavior**.

### Entity class requirements

To let Hibernate manage persistence correctly, an entity class must have:

1. **Default (no-arg) constructor**
  - Needed by Hibernate to instantiate objects using reflection.
  - Can be public or protected.
2. **Getters and setters for all properties**
  - Hibernate relies on them to read/write property values (unless you use field access with annotations on fields).
3. **ID property** (with @Id and optionally @GeneratedValue)
  - Preferably Integer / Long to enable null detection for transient entities.

## Entity State Transitions | Entity Life cycle Description (Refer to the diagram)

1. Entity life cycle begins with the state “Does Not Exist” , in heap.
2. It enters the “Transient” state when you create entity instance using new keyword with the constructor. Here it is in java object heap , but neither associated with L1 cache nor with DB
3. Using save | persist | saveOrUpdate | merge , Transient entity becomes Persistent.  
Here entity(actually its reference) is added to L1 cache. But it does not have DB identity yet.  
Such persistent entity , gains DB identity upon commit.
4. Entity transition from “Does Not Exist” → Persistent can also happen on Session API – get ,load , result of JPQL queries
5. When entity is in persistent state, Hibernate is responsible for tracking its state , in L1 cache.
6. Persistent entity, enters a “Detached” state , with these triggers – session closing , evict or clear.  
Here it’s detached from L1 cache BUT it’s corresponding record still exists in DB.
7. Detached → Persistent can be done with session.update | merge.
8. Persistent entity, enters a “Removed” state when you call session.remove | session.delete. Here entity is simply marked for removal. Its present in L1 cache n DB. Upon commit, hibernate triggers session.flush() , performs DML - delete , removing entity from DB . Then session closes , removing entity from L1 cache. So becomes transient all over again!
9. If no of references to such transient entity reduce to 0 (eg - object created in method & its reference not returned to caller) , then this entity is eligible for GC. Thus ending entity life cycle.

## Important Understanding

### What happens on transaction.commit() in Hibernate / JPA?

#### 1. Flush (Automatic or Manual)

- Hibernate **performs an automatic flush** before the actual commit on DB (unless FlushMode.MANUAL is set).
- **Flush = Synchronizing the in-memory state (L1 cache) with the database.**
- During flush:
  1. **Dirty checking** occurs:
    - Hibernate compares the **current state of entities in L1 cache** with their **snapshot state** ( The state of the entity when it was loaded or last flushed.).
    - Determines what needs to be persisted.
  2. **SQL operations are generated:**
    - **New entities** → INSERT
    - **Updated entities** → UPDATE

- **Entities marked for deletion** → DELETE
- These SQL statements are **executed in the database** (depending on the JDBC batch settings, sometimes they are batched).

## 2. Transaction Commit

- After flush, Hibernate **commits the database transaction**.
- This ensures that all changes are **durably saved** in the database.

## 3. Session Close (done implicitly, or done in finally)

- `session.close()`:
  - **Destroys the L1 cache** (the session is no longer usable).
  - **Returns the DB connection** to the connection pool (e.g., DBCP, HikariCP).
  - Makes the connection available for reuse, improving scalability.

**What happens on `transaction.rollback()` in Hibernate / JPA?**

**Step 1 & 2 are not performed. Hibernate directly performs Step 3 of closing the Session.**

## Hibernate API

### 1. SessionFactory's `openSession()`

`public Session openSession()` throws `HibernateException`

- Always opens a new Session from the SessionFactory.
- **Results into a new L1 cache.**
- The session is not bound to any thread.
- You are responsible for closing it explicitly to avoid resource leaks.
- Typically used in manual session management or when you need multiple independent sessions.

### 2. SessionFactory's `getCurrentSession()`

`public Session getCurrentSession()` throws `HibernateException`

- Returns a session associated with the current thread
- If no session exists for the current thread, Hibernate opens a new session and binds it to the thread.
- Each thread can only have one current session.
- The session is automatically flushed and closed at the transaction boundary (commit or rollback).
- You do not need to close the session manually.
- Requires configuration in `hibernate.cfg.xml` or `hibernate.properties`:  
`<property name="hibernate.current_session_context_class">thread</property>`  
 or managed automatically, in modern Spring Boot application. **(Via `@Transactional` - Spring manages the transaction and session lifecycle automatically.)**

### 3. Hibernate Session API

`public void persist(Object transientRef)`

- `persist()` is meant for new (transient) entities that are not yet associated with the session.
- Hibernate expects the entity to not have an identifier assigned (i.e default value), unless you are using assigned IDs, and even then it must not conflict with existing rows.
- Behavior:
  - Adds the entity to L1 cache
  - Schedules INSERT at flush/commit
  - Fails & throws `org.hibernate.PersistentObjectException` , if you pass any detached entity to it.
- Hibernate treats the entity as detached if it has non-null ID (!= un saved value) and the object was not just created.

When does `persist()` throw `PersistentObjectException`?

- In case a detached entity passed to `persist` .

### 4. `public Object save(Object entity)` **Deprecated**

- Old behavior:
  - If you pass a non-null ID (even existing), Hibernate would ignore it and generate a new identifier.
  - Always returned a `Serializable` (the new ID).
  - Not JPA-compliant → **deprecated in Hibernate 6.**
- Why removed: It led to confusing cases where developers thought their manually assigned ID would be used, but Hibernate silently ignored it.
- Use `persist` instead.

### 5. `public void saveOrUpdate(Object entity)` **Deprecated**

- Use case: Can handle both transient and detached.
- Behavior:
  - null ID → insert (like `save()`)
  - non-null, existing ID → update
  - non-null, non-existing ID → `StaleStateException` (trying to update/delete a row that doesn't exist)
- Not JPA standard API — Hibernate-specific convenience API.

### Recommended Hibernate 6+ Practice

- Prefer `persist()` for new entities.
- Use `merge()` if you want to reattach/update a detached entity.

### 6. `public Object merge(Object entity)` in Hibernate

- Works with both transient and detached entities.
- Creates or retrieves a managed (persistent) copy of the entity.
- The original argument object (the one you pass in) remains detached/transient — it does not become managed.
- Always returns the managed (persistent) instance.
- If you pass transient entity (null ID)
  - Hibernate fires insert query (like `save()`/`persist()`)
  - Returns a new persistent instance with generated ID
  - The passed entity is still transient
- If you pass Detached entity with existing ID
  - Hibernate sees non-null, existing ID → update
  - Issues a select (to fetch persistent instance if not already in session)
  - Copies fields from detached entity → persistent entity
  - Returns persistent entity to the caller
- If you pass Detached entity without existing ID
  - Hibernate does select → finds no row
  - Ignores your passed ID → generates a new one
  - Issues insert
  - Returns a new persistent instance

Important difference vs `saveOrUpdate()`:

- `saveOrUpdate()` would throw `StaleStateException` in this case.
- `merge()` instead saves a new row, ignoring your bogus ID.
- Does not throw `NonUniqueObjectException`  
Even if another persistent entity with the same ID is already in session. It just copies state onto the persistent one.
- The input object is never attached  
Hibernate creates or retrieves a persistent copy, and that's what you should use afterward.

## 7. Hibernate `Session.get()` API

**<T> T get(Class<T> entityClass, Object id) throws HibernateException**

#### **Parameters**

- **entityClass** → the entity class type (e.g. Employee.class)
- **id** → the identifier (primary key)

#### **Behavior**

1. **First checks L1 cache (PersistenceContext)**
  - If entity with that ID is already in the session → returns the same persistent instance (no DB hit).
2. **If not in L1 cache → hits the database immediately**
  - Issues a SELECT query
  - Loads entity into session (L1 cache)
  - Returns it as a persistent instance
3. **If no row found → returns null**
  - This is a key difference from **load()**, which returns a proxy and throws **ObjectNotFoundException** if the entity doesn't exist.

#### **Return Type**

- Returns a managed (persistent) entity instance if found.
- Returns null if not found.

#### **Quick Comparison: get() vs load()**

Method	L1 Cache	DB Hit	Not Found Behavior
get()	Checks L1 first	Immediate	DB query
load()	Checks L1 first	Returns proxy (DB hit only on access)	Throws ObjectNotFoundException

#### **Why Hibernate Recommends get() over load() in Modern Apps**

##### **1. get() is predictable**

- **get()** always returns either:
  - A persistent entity instance (if found), or
  - null (if not found).
- No surprises here.

## 2. load() relies on proxies

- load() does not immediately hit the DB. Instead:
  - Returns a proxy object (lazy placeholder).
  - DB is queried only when you access a non-identifier property.
- If the entity doesn't exist:
  - You only find out later when accessing the proxy → `ObjectNotFoundException`.
- It throws `LazyInitializationException` when you try to access an uninitialized proxy, outside the session scope.

## 3. Modern apps don't benefit much from load()'s lazy proxies

- In older Hibernate days, load() was used to avoid unnecessary queries (e.g., only needing the ID).
- But now:
  - Because modern Hibernate has bytecode enhancement and fetch profiles for efficient lazy loading, so we don't need load()'s proxy-based trick anymore. `get()` | `find()` is clearer and JPA-standard.

## 8. Get all entities.

### Steps

#### 1. Use HQL (hibernate query language) OR JPQL (Java Persistence query language)

- Object oriented Query language, DB independent. Here table name is replaced by POJO class name, column name by POJO property name.
- ? as positional parameter is supported but better recommendation is to use named IN parameters.

eg - sql – select \* from users

hql –from User

jpql –select u from User u

u – alias to an entity (u.\* implies all properties, implying all columns)

#### 2. Create Query object to hold HQL/JPQL

##### Session API

`public Query<T> createQuery(String jpql/hql, Class<T> result)` throws `HibernateException`

T – type of the result

#### 3. API of `org.hibernate.query.Query<T>` - i/f

##### 3.1 To execute the select query, with multiple results

**public List<T> getResultList() throws HibernateException**

**T - type of result**

**Returns List of PERSISTENT entities**

### **3.2 In case of single result**

**public T getSingleResult() throws HibernateException**

**Throws -**

**NoResultException - in case of no results**

**IllegalStateException - in case of DML**

**NonUniqueResultException - in case of multiple results.**

## **9. To set Named Parameter in JPQL**

**// Suppose you want user by email(unique)**

**User user =session.createQuery("select u from User u where u.email = :em", User.class)**

**.setParameter("name", userName)**

**.getSingleResult();**

- **u → alias , email →field in User class , :em → named parameter**
- **setParameter("name", value) binds the variable**

## **2 Using DTO Projection (JPQL constructor expression) with Named Parameter**

**UseDTO dto= session.createQuery(**

**"SELECT new com.app.dto.UserDTO(u.firstName, u.lastName) FROM User u WHERE u.email = :em", UserDTO.class)**

**.setParameter("em",someEmail)**

**.getSingleResult();**

- **Still type-safe and avoids returning entities if you only need certain fields**

## **3. Multiple Named Parameters example**

- **session.createQuery( "SELECT new com.app.dto.UserDTO(u.firstName, u.lastName)  
"FROM User u WHERE u.age > :age AND u.status = :sts**

## **4 IN Clause with Named Parameter**

**List<String> emails = List.of("rama@gmail.com", "kiran@gmail.com");**

```
session.createQuery(
```

```
List<UserDTO> dtos= session.createQuery( "SELECT new com.app.UserDTO(u.firstName,  
u.lastName) FROM User u WHERE u.email IN :emails", UserDTO.class)
```

```
.query.setParameter("names", names)
```

```
.getResultList();
```