# **Spring Data JPA**

Spring Data JPA is a part of the **Spring Data** project that makes it easier to work with relational databases using JPA (Java Persistence API). It reduces boilerplate code and provides an abstraction layer over **JPA** and **Hibernate**.

### 

### **Why Use Spring Data JPA?**

✅ **Less Boilerplate Code:** No need to write EntityManager operations manually.  
✅ **Built-in CRUD Methods:** Predefined methods like save(), findById(), findAll(), deleteById().  
✅ **Derived Query Methods:** Automatically generates queries from method names (findByName, findByEmail).  
✅ **Pagination & Sorting Support:** Easily paginate and sort results.  
✅ **JPQL & Native Queries:** Write custom queries if needed.  
✅ **Transaction Management:** Supports @Transactional for handling transactions.

### **Setting Up Spring Data JPA (Project Configuration)**

<dependencies>

<!-- Spring Boot Starter for JPA -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- MySQL Driver (if using MySQL) -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

# **Repositories**

In Spring Data JPA, Repository, CrudRepository, and JpaRepository are interfaces used to define the persistence layer for interacting with databases. They differ in terms of the functionality they provide and the level of abstraction they represent.

### 1. Repository

* Definition: The base interface in Spring Data. All other repository interfaces extend from it.
* Purpose: Acts as a marker interface to enable Spring Data repositories.
* Features:
  + Provides no methods by itself; it simply marks a class as a repository.
  + Cannot be used directly for CRUD operations.

Example:

public interface MyRepository extends Repository<Entity, ID> {

// No methods are provided by default

}

### 2. CrudRepository

* Definition: An extension of the Repository interface, providing CRUD (Create, Read, Update, Delete) operations.
* Purpose: To simplify CRUD operations on an entity.
* Features:
  + Provides methods like save(), findById(), findAll(), delete(), etc.
  + Suitable for basic CRUD functionality.

Example:

public interface MyRepository extends CrudRepository<Entity, ID> {

// Inherits CRUD methods

}

Common Methods:

* S save(S entity)
* Optional<T> findById(ID id)
* boolean existsById(ID id)
* Iterable<T> findAll()
* long count()
* void deleteById(ID id)
* void delete(T entity)

### 3. JpaRepository

* Definition: An extension of CrudRepository, with additional JPA-specific functionality.
* Purpose: To provide JPA-related operations such as batch processing and pagination.
* Features:
  + Adds JPA-specific methods like flush() and saveAndFlush().
  + Provides built-in support for pagination and sorting.
  + Suitable for advanced use cases requiring JPA-specific features.

Example:

public interface MyRepository extends JpaRepository<Entity, ID> {

// Inherits CRUD and JPA-specific methods

}

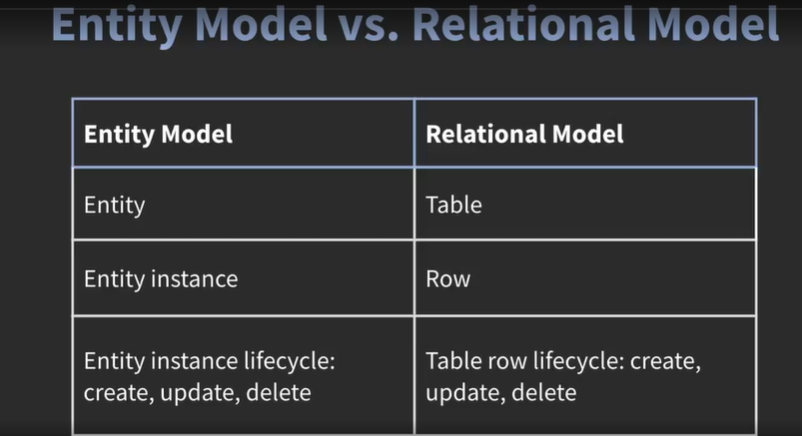
Additional Methods:

* void flush(): Flushes changes to the database.
* S saveAndFlush(S entity): Saves and immediately flushes changes.
* void deleteInBatch(Iterable<T> entities): Deletes entities in a batch.
* List<T> findAll(Sort sort): Returns all entities sorted by the specified criteria.
* Page<T> findAll(Pageable pageable): Returns entities in a paginated format.

### Comparison Table Repository, CrudRepository, and JpaRepository

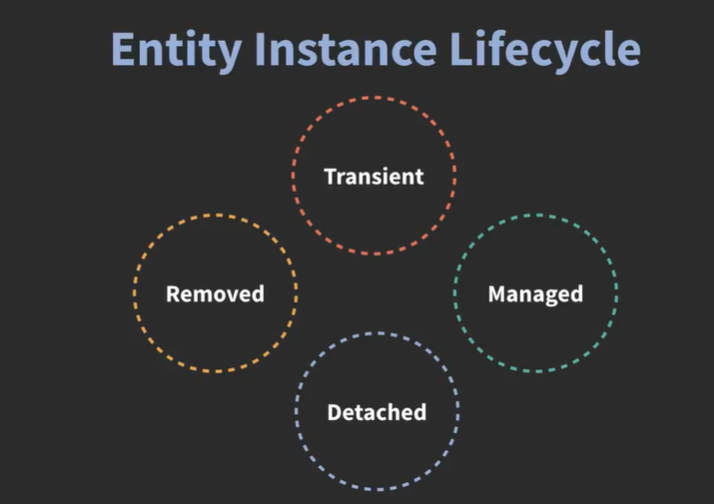
| Feature | Repository | CrudRepository | JpaRepository |
| --- | --- | --- | --- |
| Marker Interface | Yes | No | No |
| CRUD Methods | No | Yes | Yes |
| Pagination & Sorting | No | No | Yes |
| JPA-Specific Features | No | No | Yes |
| Use Case | Base marker | Basic CRUD operations | Advanced JPA functionality |

# **Entity Instance Life Cycle in JPA**



In JPA, an entity instance goes through different states in its life cycle.

The four primary states are:



### 

### **1. Transient (New) State**

* An entity is in a **transient state** when it is just created using the new keyword but is not yet associated with any persistence context.
* The entity **does not exist in the database** and is **not managed** by the EntityManager.
* If the application loses the reference to this object, it will be **garbage collected**.

User user = new User(); // New entity, not managed yet

user.setName("John Doe");

### **2. Managed (Persistent) State**

* An entity enters the **managed (persistent) state** when it is associated with a persistence context.
* The entity's state is **synchronized with the database** (i.e., any changes made to the entity are tracked and saved automatically).
* This happens when the entity is:
  + Persisted using persist(entity)
  + Retrieved from the database using find(), getReference(), or createQuery()
  + Merged using merge(entity)

EntityManager em = entityManagerFactory.createEntityManager();

em.getTransaction().begin();

User user = new User();

user.setName("John Doe");

em.persist(user); // Now the entity is managed

em.getTransaction().commit();

### **3. Detached State**

* An entity enters the **detached state** when it was previously managed but is now **removed from the persistence context**.
* The entity still **exists in the database**, but **changes made to it will not be persisted** unless explicitly merged back.
* This happens when:
  + The persistence context is closed (em.close())
  + The entity is manually detached using detach(entity)
  + The transaction is committed or rolled back

User user = em.find(User.class, 1); // Managed state

em.detach(user); // Now it's detached

user.setName("Updated Name"); // No effect in the database

To persist changes to a detached entity, use merge():

em.getTransaction().begin();

em.merge(user); // Now it’s managed again

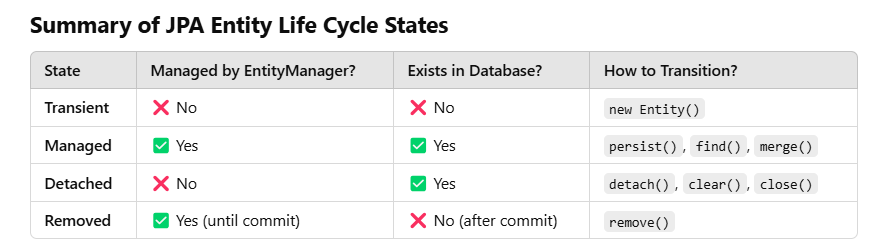
em.getTransaction().commit();

### **4. Removed (Deleted) State**

* An entity enters the **removed state** when it is scheduled for deletion.
* The entity is still **managed** until the transaction is committed, after which it is permanently deleted from the database.
* The entity is removed using remove(entity)

User user = em.find(User.class, 1); // Managed state

em.remove(user); // Now it's in the removed state



Note: Entity Manager manages the entity Persistence stages

# **Derived Query Methods**

Derived query methods use **method names** to define queries instead of writing @Query annotations or JPQL.

✅ **No need to write SQL manually**✅ **Spring automatically generates queries**✅ **Easy to use for basic filtering**

User findByName(String name); SELECT \* FROM users WHERE name= ?;

User findByEmail(String email); SELECT \* FROM users WHERE email = ?;

User findByNameAndEmail(String name, String email); SELECT \* FROM users WHERE name = ? AND email = ?;

User findByNameOrEmail(String name, String email); SELECT \* FROM users WHERE name = ? OR email = ?;

List<User> findByNameStartingWith(String prefix); SELECT \* FROM users WHERE name LIKE 'prefix%';

List<User> findByNameEndingWith(String suffix); SELECT \* FROM users WHERE name LIKE '%suffix';

List<User> findByNameContaining(String keyword); SELECT \* FROM users WHERE name LIKE '%keyword%';

List<User> findByNameIgnoreCase(String name); SELECT \* FROM users WHERE LOWER(name) = LOWER(?);

List<User> findByAgeGreaterThan(int age); SELECT \* FROM users WHERE age > ?;

List<User> findByAgeLessThan(int age); SELECT \* FROM users WHERE age < ?;

List<User> findByAgeBetween(int startAge, int endAge); SELECT \* FROM users WHERE age BETWEEN ? AND ?;

List<User> findByNameOrderByEmailAsc(String name); SELECT \* FROM users WHERE name = ? ORDER BY email ASC;

List<User> findByNameOrderByEmailDesc(String name); SELECT \* FROM users WHERE name = ? ORDER BY email DESC;

List<User> findTop3ByOrderByAgeDesc(); SELECT \* FROM users ORDER BY age DESC LIMIT 3;

List<User> findFirst2ByOrderByNameAsc(); SELECT \* FROM users ORDER BY name ASC LIMIT 2;

List<User> findByActiveTrue(); SELECT \* FROM users WHERE active = true;

List<User> findByActiveFalse(); SELECT \* FROM users WHERE active = false;

List<User> findByNameAndAgeGreaterThan(String name, int age); SELECT \* FROM users WHERE name = ? AND age > ?;

List<User> findByEmailContainingAndActiveTrue(String email); SELECT \* FROM users WHERE email LIKE '%email%' AND active = true;

# **What is JPQL(Java Persistence Query Language)?**

JPQL is **object-oriented** and works with **entity names and fields** instead of table names and columns.  
✅ **Portable across databases**✅ **Uses Java entity names & field names instead of SQL table names & column names**✅ **Supports joins, filtering, and aggregation**

@Query("SELECT u FROM User u WHERE u.email = :email")

User findUserByEmail(@Param("email") String email);

Equivalent SQL:

SELECT \* FROM users WHERE email = ?;

**Notice:** User is the entity, and email is the field (not database table or column names).

# **What is a Native Query?**

A **native query** is a raw SQL statement executed directly in the database using @Query.

Native Queries use **raw SQL** and can reference **table names** and **column names directly**.  
✅ **Full control over SQL syntax**✅ **Best for complex joins or database-specific queries**

**We can pass parameters using named (:param) or positional (?1) parameters.**

@Query(value = "SELECT \* FROM employee WHERE department = :dept", nativeQuery = true)

List<Employee> findByDepartment(@Param("dept") String department);

📌 **Usage in Service Layer:**

**List<Employee> itEmployees = employeeRepository.findByDepartment("IT");**

**🔹 Notice:** This uses employee (table name) and department (column name) directly.

### **Updating Data with Native Query**

We can also perform **update and delete** operations using @Modifying.

@Modifying

@Transactional

@Query(value = "UPDATE employee SET salary = salary + 5000 WHERE department = 'IT'", nativeQuery = true)

void increaseSalaryForIT();

📌 **Notes:**

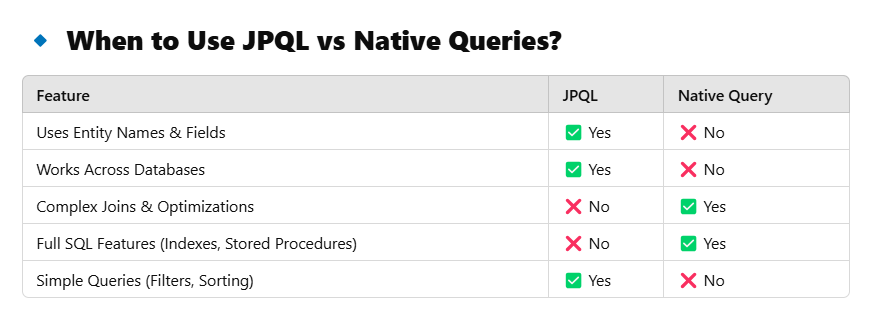
* @Modifying → Required for UPDATE or DELETE queries.
* @Transactional → Required to ensure data consistency.

By default, **Spring Data JPA assumes queries are SELECT statements**.

@Modifying tells Spring **this is an UPDATE or DELETE query**.

Without @Modifying, you'll get an error like:

org.springframework.dao.InvalidDataAccessApiUsageException: Executing an update/delete query



# **Named Queries in JPA**

JPA provides two types of Named Queries:  
 1️⃣ **@NamedQuery** → Uses **JPQL** (database-independent).  
 2️⃣ **@NamedNativeQuery** → Uses **Raw SQL** (database-dependent).

### 1️⃣ Named JPQL Query (@NamedQuery)

A **Named Query** is defined at the **entity level** and can be called multiple times in different parts of the code.

@Entity

@NamedQuery(name = "Employee.findByDepartment", query = "SELECT e FROM Employee e WHERE e.department = :dept")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

}

**Using the Named Query in Repository**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

**@Query(name = "Employee.findByDepartment")**

List<Employee> findByDepartment(@Param("dept") String department);

}

📌 **Notes:**

* The **query name** (Employee.findByDepartment) **must match** the one in @NamedQuery.
* @Query(name = "queryName") tells Spring to look for a **Named Query** instead of writing JPQL inside @Query.

### 

### 2️⃣ Named Native Query (@NamedNativeQuery)

If we need raw SQL, we use @NamedNativeQuery.

**@Entity**

**@NamedNativeQuery(**

**name = "Employee.findByDepartmentNative",**

**query = "SELECT \* FROM employee WHERE department = :dept",**

**resultClass = Employee.class**

)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

}

✅ **Uses raw SQL** but still maps the result to the Employee entity.

**Using the Named Native Query in Repository**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

**@Query(name = "Employee.findByDepartmentNative", nativeQuery = true)**

List<Employee> findByDepartmentNative(@Param("dept") String department);

}

**📌 Notes:**

* **nativeQuery = true → Tell JPA that this is a Native Query.**
* **resultClass = Employee.class → Maps the raw SQL result to Employee.**

# **Pagination & Sorting in Spring Data JPA**

When dealing with large datasets, fetching all records at once can lead to **performance issues**.  
Spring Data JPA provides **pagination and sorting** to efficiently manage large data queries.

✅ Improves performance by fetching only a subset of data

✅ Optimizes database queries using LIMIT and OFFSET

✅ Reduces memory usage in the application

✅ Enables better user experience for UI-based applications

**Repository:**

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.repository.PagingAndSortingRepository;

import java.util.List;

public interface UserRepository extends JpaRepository<User, Long>, **PagingAndSortingRepository**<User, Long> {

// Custom method with pagination support

List<User> findByAgeGreaterThan(int age, Pageable pageable);

}

**Service:**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Pageable;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getUsersOlderThan(int age, int page, int size) {

**Pageable pageable = PageRequest.of(page, size);**

return userRepository.findByAgeGreaterThan(age, **pageable**);

}

}

**Controller:**

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/older-than")

public List<User> getUsersOlderThan(

**@RequestParam int age,**

**@RequestParam(defaultValue = "0") int page,**

**@RequestParam(defaultValue = "5") int size)** {

return userService.getUsersOlderThan(age, page, size);

}

}

**URL: GET** [**http://localhost:8080/users/older-than?age=30&page=0&size=5**](http://localhost:8080/users/older-than?age=30&page=0&size=5)

# 

# **JPA Criteria API**

The EntityManager is responsible for executing Criteria Queries.

@Repository

public class EmployeeCriteriaRepository {

@PersistenceContext

private EntityManager entityManager;

public List<Employee> getEmployeesWithHighSalary() {

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Employee> query = cb.createQuery(Employee.class);

Root<Employee> root = query.from(Employee.class);

// WHERE salary > 50000

Predicate salaryPredicate = cb.greaterThan(root.get("salary"), 50000);

Predicate activePredicate = cb.isTrue(root.get("active"));

// Combine both conditions with AND

query.where(cb.and(salaryPredicate, activePredicate));

// Execute the query

return entityManager.createQuery(query).getResultList();

}

}

# **@EntityGraph**

### **What is the N+1 Problem in JPA?**

👉 Imagine you have two entities:

* **Department** → OneToMany → **Employee**.
* You want to fetch the Department **with all Employees**.

public interface DepartmentRepository extends JpaRepository<Department, Long> {

List<Department> findAll();

}

### **Generated SQL:**

SELECT \* FROM department;

SELECT \* FROM employee WHERE department\_id = 1;

Total Queries Executed = 1 (for Department) + N (for each Employee List)

✅ This is called the **N+1 Problem**.

* 🚨 **1 query** to fetch all departments.
* 🚨 **5 additional queries** to fetch employees for each department.

👉 **@EntityGraph** is designed to solve this problem.

👉 It allows JPA to fetch **Parent + Child in ONE SQL QUERY**.

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

**@EntityGraph(attributePaths = {"employees"})**

List<Department> findAll();

}

@EntityGraph(attributePaths = {"employees"})

✅ Tells JPA:

* 💡 "Hey JPA, when fetching Department → **automatically fetch Employees**."
* 💡 Do NOT do **lazy loading** → Combine the query.

SELECT d.id, d.name, e.id, e.name, e.salary FROM department d LEFT JOIN

employee e ON d.id = e.department\_id;

| **Approach** | **Total Queries** |
| --- | --- |
| ❌ Without @EntityGraph | **1 + N Queries** (Very Slow) |
| ✅ With @EntityGraph | **1 Query Only** (Super Fast) |

# **Mapping in Spring Data JPA (Entity Relationships)**

In Spring Data JPA, entity relationships are defined using **annotations** like:

* @OneToOne (One-to-One)
* @OneToMany (One-to-Many)
* @ManyToOne (Many-to-One)
* @ManyToMany (Many-to-Many)

### **1. One-to-One Mapping (@OneToOne)**

@Entity

public class Address {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String street;

private String city;

private String country;

}

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@OneToOne(cascade = CascadeType.ALL) //Automatically saves address when saving User

@JoinColumn(name = "address\_id") // Defines foreign key column in `user` table

private Address address;

}

### **2. One-to-Many & Many-to-One Mapping**

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

**@OneToMany(mappedBy = "department", cascade = CascadeType.ALL) // One-to-Many relation**

private List<Employee> employees;

// Getters and Setters

}

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

**@ManyToOne // Many Employees belong to One Department**

**@JoinColumn(name = "department\_id") // Defines foreign key**

private Department department;

}

### **3. Many-to-Many Mapping (@ManyToMany)**

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

**@ManyToMany**

**@JoinTable(**

**name = "student\_course",**

**joinColumns = @JoinColumn(name = "student\_id"),**

**inverseJoinColumns = @JoinColumn(name = "course\_id")**

**)**

private List<Course> courses;

}

CREATE TABLE student (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255)

);

CREATE TABLE course (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255)

);

CREATE TABLE student\_course ( -- Join table

student\_id BIGINT,

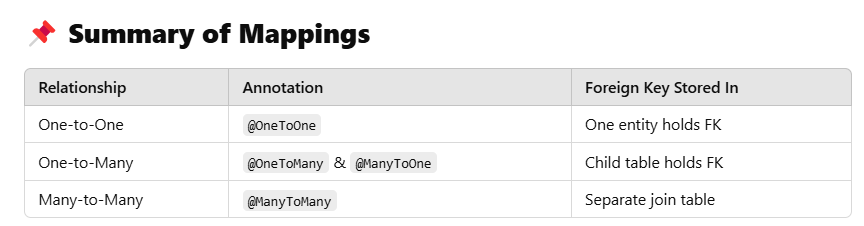
course\_id BIGINT,

PRIMARY KEY (student\_id, course\_id),

FOREIGN KEY (student\_id) REFERENCES student(id),

FOREIGN KEY (course\_id) REFERENCES course(id)

);



# **Fetch Types & Cascade Types in Spring Data JPA**

When working with **entity relationships**, we often need to control:

1️⃣ **How data is loaded from the database** → FetchType

2️⃣ **How operations on one entity affect related entities** → CascadeType

### **Fetch Types (FetchType.LAZY vs FetchType.EAGER)**

1. FetchType.LAZY (Default for @OneToMany, @ManyToMany)

* Related entity is **not loaded** immediately.
* It is fetched **only when accessed** (using a proxy).
* More efficient for large datasets.

1. FetchType.EAGER (Default for @ManyToOne, @OneToOne)

* Related entities are **always loaded immediately** with the main entity.
* Increases performance for **small** relationships but can lead to **performance issues** for large datasets.

@ManyToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)

### **Cascade Types (CascadeType)**

The cascade attribute determines how operations (**persist, remove, merge, etc.**) on a parent entity affect related child entities.

| **Cascade Type** | **Description** |
| --- | --- |
| PERSIST | Saves the child entity when the parent is saved. |
| MERGE | Updates the child entity when the parent is updated. |
| REMOVE | Deletes the child entity when the parent is deleted. |
| REFRESH | Refreshes the child entity when the parent is refreshed. |
| DETACH | Detaches the child entity when the parent is detached. |
| ALL | Applies all above operations. |

# **@Transactional in Spring Data JPA**

The @Transactional annotation in Spring is used to manage database transactions. It ensures that operations (like save, update, delete) are executed **within a single transaction** and either **fully complete or rollback** in case of an error.

### **1️⃣ Why Use @Transactional?**

* Ensures **data consistency** (either all changes are committed or none).
* Helps manage **LazyInitializationException** in FetchType.LAZY.
* Avoids multiple unnecessary database queries by using the **same persistence context**.

### **2️⃣ How @Transactional Works**

When a method is marked with @Transactional:

1. Spring **opens a transaction** before the method executes.
2. It runs all queries inside a **single transaction**.
3. If no exceptions occur, it **commits** the transaction.
4. If an exception occurs, it **rolls back** all changes.

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void updateEmployee(Long id, String newName) {

Employee emp = employeeRepository.findById(id).orElseThrow();

emp.setName(newName);

// No need to explicitly call save(), as transaction commits changes automatically

}

}

✅ **No need to call save(emp) explicitly.**

✅ **Transaction commits when the method completes successfully.**

### **3️⃣ Transaction Rollback on Exception**

If an exception occurs inside a @Transactional method, all database operations in that transaction will be **rolled back**.

@Transactional

public void updateAndFail(Long id, String newName) {

Employee emp = employeeRepository.findById(id).orElseThrow();

emp.setName(newName);

if (true) {

throw new RuntimeException("Something went wrong!");

}

}

**Result:**

❌ The update operation is rolled back, and no changes are saved!

### **4️⃣ Controlling Rollback with rollbackFor**

By default, transactions **only roll back for unchecked exceptions (RuntimeException)**. To roll back on **checked exceptions**, specify rollbackFor.

**@Transactional(rollbackFor = Exception.class)**

public void updateEmployee(Long id, String newName) throws Exception {

Employee emp = employeeRepository.findById(id).orElseThrow();

emp.setName(newName);

throw new Exception("Checked Exception!"); // Transaction will roll back

}

✅ **Ensures rollback for checked exceptions.**

### 5️⃣ **Read-Only Transactions (readOnly = true)**

If you only want to fetch data without modifying it, use @Transactional(readOnly = true).  
🔹 **This improves performance** because Hibernate **does not track entity changes.**

@Transactional(readOnly = true)

public Employee getEmployee(Long id) {

return employeeRepository.findById(id).orElseThrow();

}

### 6️⃣ **Transaction Propagation Types**

Spring allows different transaction propagation behaviors using propagation.

| **Propagation Type** | **Description** |
| --- | --- |
| REQUIRED (Default) | Uses an existing transaction or creates a new one if none exists. |
| REQUIRES\_NEW | Always creates a new transaction, suspending the existing one. |
| SUPPORTS | Runs inside a transaction if one exists; otherwise, runs without one. |
| NOT\_SUPPORTED | Runs without a transaction, suspending any existing one. |
| MANDATORY | Throws an exception if no transaction exists. |
| NEVER | Throws an exception if a transaction exists. |
| NESTED | Creates a new nested transaction within the existing one. |

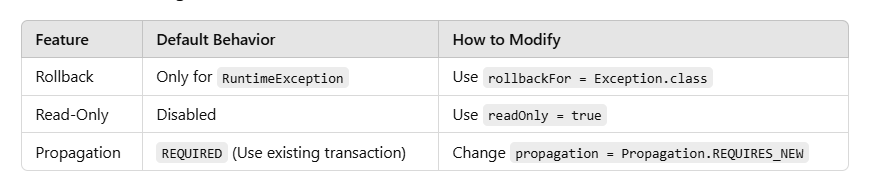
**@Transactional(propagation = Propagation.REQUIRES\_NEW)**

public void saveLog(String action) {

logRepository.save(new Log(action));

}

✅ **Ensures saveLog() runs in a separate transaction, even if the main transaction fails.**



# 

# **@Embeddable and @Embedded in JPA**

Sometimes, we need to **reuse a group of fields** across multiple entities without creating separate tables. Instead of making a separate entity, we can **embed** an object into an entity using @Embeddable and @Embedded.

@Embeddable → Marks a class as **a reusable component** that can be embedded in entities.

@Embedded → Marks a field in an entity that holds an embeddable class.

### **Storing Address in an Employee**

Instead of making Address a separate entity with its own table, we **embed it inside** Employee.

**@Embeddable**

public class Address {

private String city;

private String state;

private String zipCode;

}

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@Embedded

private Address address;

}

### **Generated Table (employee Table)**

| **id** | **name** | **city** | **state** | **zipCode** |
| --- | --- | --- | --- | --- |
| 1 | John | NY | NY | 10001 |
| 2 | Alice | LA | CA | 90001 |

✅ The Address fields are **embedded inside the Employee table** (No separate Address table is created).

### **Customizing Embedded Field Names (@AttributeOverrides)**

By default, the embedded fields **use the same column names**. If we have **multiple embedded fields** in one entity, we need **custom column names**.

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@Embedded

@AttributeOverrides({

@AttributeOverride(name = "city", column = @Column(name = "home\_city")),

@AttributeOverride(name = "state", column = @Column(name = "home\_state")),

@AttributeOverride(name = "zipCode", column = @Column(name = "home\_zip"))

})

private Address homeAddress;

@Embedded

@AttributeOverrides({

@AttributeOverride(name = "city", column = @Column(name = "work\_city")),

@AttributeOverride(name = "state", column = @Column(name = "work\_state")),

@AttributeOverride(name = "zipCode", column = @Column(name = "work\_zip"))

})

private Address workAddress;

}

### **Generated Table (employee Table)**

| **id** | **name** | **home\_city** | **home\_state** | **home\_zip** | **work\_city** | **work\_state** | **work\_zip** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | John | NY | NY | 10001 | LA | CA | 90001 |

✅ Now, homeAddress and workAddress **do not overwrite each other**.

==================================================================================

# **What is First-Level Cache in Hibernate?**

**First-Level Cache** in Hibernate is an **in-memory cache (RAM)** that automatically stores:

* ✅ **Entity Objects** (like Employee, Product, etc.)
* ✅ **Data fetched from the database** during a Hibernate Session.

The First-Level Cache exists **inside the Hibernate Session (Session object)**.

Session session = sessionFactory.openSession();

👉 **A new First-Level Cache is created** inside the Session.

👉 Once you **close the Session**, the cache is destroyed.

Hibernate enables First-Level Cache **by default**.

**Can I clear Cache without closing the session?**

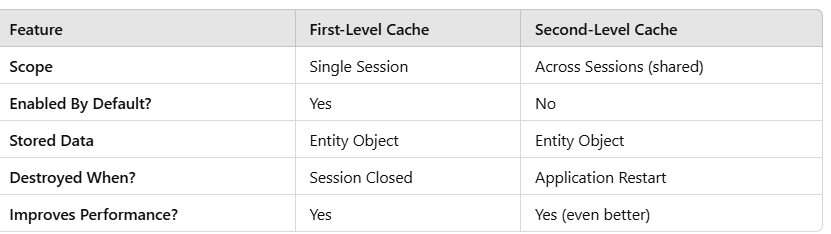
✅ Yes. Use session.clear() or session.evict().

**Does Cache work across multiple sessions?**

❌ No. Cache is destroyed once session is closed

# **What is Second-Level Cache?**

👉 Second-Level Cache is a **shared cache** in Hibernate that stores **entity data** **even after the Session is closed**.  
 👉 It works **at the SessionFactory level** (shared across multiple sessions).



# **What is Query Cache in Hibernate?**

**Query Cache** caches the **result of a query (HQL/Native)** instead of caching the Entity.  
 👉 It is used when you want to cache:

* ✅ **The result of a complex query**.
* ✅ **Multiple rows (like List of Employees)**.
* ✅ **Join queries, Aggregation queries, etc.**

#### **Enable Query Cache in Hibernate**

spring.jpa.properties.hibernate.cache.use\_query\_cache=true

Query query = session.createQuery("FROM Employee WHERE salary > 50000");

query.setCacheable(true);

List<Employee> empList = query.getResultList();

👉 Query Cache **depends on Second-Level Cache**.

👉 So if you enable Query Cache, you must enable Second-Level Cache.

# **Can you clear First-Level and Second-Level Cache manually? How?**

| **Cache Type** | **How To Clear** | **When Is It Useful?** |
| --- | --- | --- |
| **First-Level Cache** | session.clear() or session.evict() or session.close() | During same session only |
| **Second-Level Cache** | sessionFactory.getCache().evictAll();  This will clear the **entire cache (all entities)**.  sessionFactory.getCache().evict(Employee.class);  This will clear cache only for **Employee Entity**.  sessionFactory.getCache().evict(Employee.class, 1L);  This will clear cache **only for Employee ID = 1**. | Across sessions |
| **Query Cache** | sessionFactory.getCache().evictQueryRegion() | For expensive queries |

# **What Happens If You Forget To Close a Session?**

👉 This is the **deadliest mistake** you can do in Hibernate.  
 👉 If you **forget to close** a Session:

| **Problem** | **What Will Happen?** |
| --- | --- |
| **Memory Leak** | Session will keep holding memory. |
| **DB Connection Leak** | Session won't release connection. |
| **Transaction Lock** | Session won't release lock. |
| **Heap Size Full** | JVM will throw OutOfMemoryError. |
| **Too Many Connections** | DB will reject new connections. |

# What is the difference between Session.save() vs Session.persist()?

| **Feature** | **session.save()** | **session.persist()** |
| --- | --- | --- |
| **API Type** | Hibernate-specific (org.hibernate.Session) | JPA Standard (javax.persistence.EntityManager) |
| **Returns ID?** | ✅ Yes (Primary Key) | ❌ No (Does not return ID) |
| **Follows JPA Specification?** | ❌ No | ✅ Yes |
| **Attach Entity to Persistence Context?** | ✅ Yes | ✅ Yes |
| **Good For Large Projects?** | ❌ No | ✅ Yes |
| **Use In JPA Projects?** | ❌ No | ✅ Yes |
| **Use In Small Projects?** | ✅ Yes | ✅ Yes |

# **Key Differences Between get() vs load()**

| **Feature** | **session.get()** | **session.load()** |
| --- | --- | --- |
| **Database Hit** | ✅ Hits Database Immediately | ❌ Does NOT Hit Database Immediately( Only When Field Is Accessed) |
| **Returns Object?** | ✅ Returns Real Object | ✅ Returns Proxy Object (Fake Object) |
| **If Record Not Found** | ✅ Returns null | ❌ Throws ObjectNotFoundException |
| **Performance** | ❌ Slow (Immediate Hit) | ✅ Fast (Lazy Initialization) |
| **Use In Real Time?** | ✅ When you are sure ID exists. | ✅ When you want lazy load. |
| **Exception On Missing Record** | ❌ No Exception | ✅ Throws Exception |

# **Difference Between merge() vs update() In Hibernate**

| **Feature** | **merge() (Recommended)** | **update()** |
| --- | --- | --- |
| **If Record Does Not Exist?** | ✅ It will **INSERT** a new record (no exception). | ❌ It will throw **StaleObjectStateException**. |
| **Creates A New Object In Session?** | ✅ Yes, it creates a **new copy** of the object. | ❌ No, it uses the **same object**. |
| **Safe From Exception?** | ✅ Yes, no exception even if the record is missing. | ❌ Throws exception if the record does not exist. |
| **Recommended In Real-Time?** | ✅ Always recommended. 🚀🔥 | ❌ Use only if you're 100% sure the record exists. |