**Cascade Types**

JPA provides the following options:

| **Cascade Type** | **What It Does** |
| --- | --- |
| **CascadeType.PERSIST** | When you save the parent (entityManager.persist()), all related children are also saved. |
| **CascadeType.MERGE** | When you update the parent, all related children are updated. |
| **CascadeType.REMOVE** | When you delete the parent, all related children are deleted too. *(Danger zone 🚨)* |
| **CascadeType.REFRESH** | When the parent is refreshed from the DB, children are refreshed too. |
| **CascadeType.DETACH** | When the parent is detached from the persistence context, children are detached too. |
| **CascadeType.ALL** | Applies all the above types. |

**What is FetchType?**

FetchType controls **when** related entities are loaded from the database:

* **EAGER** → Load **immediately** with the parent.
* **LAZY** → Load **only when accessed** (on-demand).

Think of it as:

* EAGER → "Bring all my friends with me right now."
* LAZY → "Call my friends only if you need them."

**1. EAGER Fetch**

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

private List<Employee> employees;

**What happens:**  
When you load a Department, **all Employees** are fetched immediately in the same SQL query (or via joins).

**2. LAZY Fetch**

@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)

private List<Employee> employees;

**What happens:**  
When you load a Department, **employees are not loaded yet**.  
They’re loaded **only when you call** department.getEmployees().

**Two Types**

java

CopyEdit

public enum FetchType {

EAGER,

LAZY

}

**Default Fetch Modes in JPA**

| **Relationship** | **Default FetchType** |
| --- | --- |
| @OneToOne | EAGER |
| @ManyToOne | EAGER |
| @OneToMany | LAZY |
| @ManyToMany | LAZY |

**Examples**

**1. EAGER Fetch**

java

CopyEdit

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

private List<Employee> employees;

**What happens:**  
When you load a Department, **all Employees** are fetched immediately in the same SQL query (or via joins).

**2. LAZY Fetch**

java

CopyEdit

@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)

private List<Employee> employees;

**What happens:**  
When you load a Department, **employees are not loaded yet**.  
They’re loaded **only when you call** department.getEmployees().

**SQL Difference**

**LAZY**

sql

CopyEdit

SELECT \* FROM department WHERE id = 1; -- Only department

-- Later...

SELECT \* FROM employee WHERE department\_id = 1; -- Triggered when accessed

**EAGER**

sql

CopyEdit

SELECT \* FROM department d

LEFT JOIN employee e ON d.id = e.department\_id

WHERE d.id = 1; -- Both tables loaded immediately

**When to Use**

* **LAZY (Recommended)**
  + Reduces initial query cost.
  + Good for large collections or rarely used relationships.
* **EAGER**
  + Use if you always need the related data right away (e.g., User ↔ Profile).

⚠ **Pitfalls**

* **EAGER** can cause "Cartesian product" problems → huge, slow queries.
* **LAZY** can cause LazyInitializationException if accessed outside a Hibernate session (common in detached objects in web apps).
  + Fix: Use @Transactional, fetch joins in JPQL, or DTO projection.

**So in short**

* **Hibernate**: Works with **POJO ↔ DB** (actual ORM work).
* **JPA**: Standard API to tell ORM (like Hibernate) how to map **POJO ↔ DB**.
* **DTO**: Separate Java objects used to carry **only the required data** from entity to external layers — mapping is your responsibility, JPA/Hibernate do not auto-convert to DTO.

💡 **Simple analogy**

* **JPA** = Law/Rules (Specification)
* **Hibernate** = Police Officer following the law (Implementation)
* **Entity (POJO)** = Original document in a secure archive (Database table mapping)
* **DTO** = A photocopy of only the parts you need to show someone (safe, filtered version for API/UI)
* **JPA annotations** = portable → works with any JPA provider (Hibernate, EclipseLink, OpenJPA, etc.)
* **Hibernate annotations** = only work with Hibernate (lock-in).

**1️⃣ JPA Specification Annotations**

*(Package: javax.persistence or jakarta.persistence in newer versions)*

**Entity & Table Mapping**

* @Entity
* @Table
* @Id
* @GeneratedValue
* @SequenceGenerator
* @TableGenerator
* @Column
* @Transient (ignore field in persistence)
* @Lob (Large Object)
* @Enumerated
* @Version (optimistic locking)

**Relationships**

* @OneToOne
* @OneToMany
* @ManyToOne
* @ManyToMany
* @JoinColumn
* @JoinTable
* @MappedBy (via mappedBy attribute)
* @MapKey / @OrderBy

**Embedded & Inheritance**

* @Embeddable
* @Embedded
* @AttributeOverride / @AttributeOverrides
* @Inheritance
* @DiscriminatorColumn
* @DiscriminatorValue

**Querying**

* @NamedQuery
* @NamedQueries
* @NamedNativeQuery
* @NamedStoredProcedureQuery
* @SqlResultSetMapping

**Lifecycle Callbacks**

* @PrePersist
* @PostPersist
* @PreUpdate
* @PostUpdate
* @PreRemove
* @PostRemove
* @PostLoad

**Locking & Fetch**

* @Lock *(via EntityManager API in code)*
* FetchType.LAZY / FetchType.EAGER (used inside relationship annotations)
* CascadeType.\*

**2️⃣ Hibernate-Specific Annotations**

*(Package: org.hibernate.annotations)*

**Fetching & Performance**

* @Fetch(FetchMode.JOIN / SELECT / SUBSELECT)
* @BatchSize
* @LazyCollection
* @LazyGroup
* @DynamicInsert
* @DynamicUpdate
* @OptimisticLock
* @SelectBeforeUpdate

**ID Generation**

* @GenericGenerator (custom strategies)
* @NaturalId (business key)

**Column & Type Mapping**

* @Type (custom Hibernate type)
* @Formula (computed column via SQL expression)
* @ColumnTransformer (custom SQL on read/write)

**Caching**

* @Cache
* @Cacheable *(also in JPA 2.0, but Hibernate has extra options)*
* @NaturalIdCache

**Filters & Interceptors**

* @Filter
* @FilterDef
* @ParamDef
* @Where
* @WhereJoinTable
* @Any / @ManyToAny (non-standard associations)

**Time & Audit**

* @CreationTimestamp
* @UpdateTimestamp

**1. Entity & Relationship Mapping Exceptions**

These happen when annotations or table structures don’t match the DB.

| **Exception** | **Cause** |
| --- | --- |
| **org.hibernate.AnnotationException** | Invalid mapping — e.g., using mappedBy on both sides incorrectly. |
| **org.hibernate.MappingException** | Hibernate cannot map an entity to the DB table (missing columns, invalid joins). |
| **javax.persistence.PersistenceException** | Generic JPA exception — often wraps mapping or DB-related errors. |
| **org.hibernate.DuplicateMappingException** | Same property or table mapping declared twice. |
| **org.hibernate.id.IdentifierGenerationException** | Primary key generation strategy misconfigured or missing ID. |

**2. FetchType & Lazy Loading Exceptions**

Mostly occur with FetchType.LAZY.

| **Exception** | **Cause** |
| --- | --- |
| **org.hibernate.LazyInitializationException** | Accessing a LAZY-loaded collection outside of an active Hibernate session/transaction. |
| **org.hibernate.ObjectNotFoundException** | Lazy loading tries to fetch an entity that no longer exists in the DB. |
| **javax.persistence.EntityNotFoundException** | Similar to above, but JPA wrapper. |

**3. Cascade & Constraint Exceptions**

Related to CascadeType.REMOVE, CascadeType.PERSIST, or CascadeType.MERGE.

| **Exception** | **Cause** |
| --- | --- |
| **javax.persistence.EntityExistsException** | Trying to persist() an entity that already exists in the DB (PK conflict). |
| **org.hibernate.TransientObjectException** | Trying to persist a parent with a transient child entity without CascadeType.PERSIST. |
| **org.hibernate.ConstraintViolationException** | Foreign key constraint fails — e.g., deleting a parent without removing children first when no cascade is defined. |
| **javax.validation.ConstraintViolationException** | Bean Validation (JSR-380) fails — e.g., @NotNull field is null. |
| **org.hibernate.PropertyValueException** | Null value assigned to a non-nullable column. |

**4. Projection Exceptions**

Happen when using DTO or interface-based queries.

| **Exception** | **Cause** |
| --- | --- |
| **org.springframework.core.convert.ConverterNotFoundException** | Spring can’t map the projection result to the target DTO/interface. |
| **java.lang.IllegalArgumentException: Unable to locate appropriate constructor** | DTO projection query doesn’t match constructor parameters. |
| **org.hibernate.QueryException** | JPQL or SQL syntax issue in projection query. |
| **org.springframework.dao.IncorrectResultSizeDataAccessException** | Query expects a single result but got multiple. |

**5. Transaction & Session Exceptions**

More about persistence context handling.

| **Exception** | **Cause** |
| --- | --- |
| **javax.persistence.TransactionRequiredException** | Executing a query that modifies data without an active transaction. |
| **org.hibernate.SessionException** | Using a closed Hibernate session. |
| **javax.persistence.RollbackException** | Transaction rollback due to any failure (e.g., constraint violation). |

**6. Common Causes in the Above Entities**

* **One-to-Many** without mappedBy → extra join table unexpectedly created.
* **Bidirectional relationships** without helper methods → foreign key not set, causing PropertyValueException.
* **EAGER fetch with large collections** → OutOfMemoryError (not strictly JPA exception, but common in practice).
* **Many-to-Many with CascadeType.REMOVE** → deletes shared entities accidentally → FK violation.

**How JpaRepository Works Internally – Step-by-Step**

**1. You Define an Interface:**

java

CopyEdit

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartmentId(Long deptId);

}

You didn’t provide an implementation — **Spring generates it at runtime.**

**2. Spring Boot Auto-Configuration:**

When the app starts:

* Spring Boot sees @EnableJpaRepositories (enabled via @SpringBootApplication)
* It scans packages for interfaces like EmployeeRepository

**3. Spring Creates a Proxy:**

For each repository interface, Spring:

* Creates a **dynamic proxy** (using **JDK proxy** or **CGLIB**)
* Delegates calls to a class like SimpleJpaRepository

java

CopyEdit

public class SimpleJpaRepository<T, ID> implements JpaRepository<T, ID> {

private final EntityManager em;

public SimpleJpaRepository(JpaEntityInformation<T, ?> entityInfo, EntityManager em) {

this.em = em;

}

@Override

public Optional<T> findById(ID id) {

return Optional.ofNullable(em.find(domainClass, id));

}

@Override

public void deleteById(ID id) {

T entity = findById(id).orElseThrow();

em.remove(entity);

}

}

So even though you didn’t write any code, Spring wires it all under the hood using this SimpleJpaRepository.

**4. Method Name Parsing (Query Derivation):**

For methods like:

java

CopyEdit

List<Employee> findByDepartmentId(Long deptId);

Spring uses **QueryLookupStrategy**:

* Parses findByDepartmentId
* Translates to JPQL:

sql

CopyEdit

SELECT e FROM Employee e WHERE e.department.id = :deptId

* Uses **JPA Criteria API** or JPQL to run the query

**5. EntityManager Executes the Query:**

Spring delegates query execution to:

java

CopyEdit

javax.persistence.EntityManager

Example:

java

CopyEdit

em.createQuery("SELECT e FROM Employee e WHERE e.department.id = :deptId")

.setParameter("deptId", 10L)

.getResultList();

**🛠 Behind-the-Scenes Classes Used:**

| **Spring Class** | **Role** |
| --- | --- |
| JpaRepositoryFactoryBean | Creates JpaRepository instances |
| SimpleJpaRepository | Core class that implements CRUD |
| EntityManager | Low-level interface that talks to DB |
| QueryLookupStrategy | Parses method names into queries |
| ProxyFactory | Generates dynamic proxy for your interface |

**Diagram View (High-Level)**

pgsql

CopyEdit

You → EmployeeRepository

↳ is a JpaRepository

↳ dynamic proxy created

↳ delegates to SimpleJpaRepository

↳ uses EntityManager

↳ interacts with Database

**✅ Benefits**

* Zero boilerplate
* Type-safe, readable method names
* Auto-transactional support
* Pluggable: You can add your own implementations or overrides