Spring

ORM (Object-Relational Mapping)

Definition / Concept:
ORM is a programming technique that allows you to interact with a relational database using objects in your programming language instead of SQL queries.

Key Points / Features:

- Maps **database tables** to **classes** and **rows** to **objects**.
- Eliminates most manual SQL, reducing code complexity.
- Supports **CRUD operations** (Create, Read, Update, Delete) via objects.
- Examples: Hibernate (Java), Entity Framework (.NET), Sequelize (Node.js).

Purpose / Importance:

- Simplifies database interaction for developers.
- Helps maintain consistency between code and database structure.
- Reduces errors from manual SQL and improves productivity.

Real-life Example / Analogy:

- Think of ORM like a **translator**: your code speaks in objects, the database speaks in tables, and ORM translates automatically.
- Example: Instead of writing INSERT INTO Users VALUES (...), you just do user.save() in code.

Interview Tip / Common Question:
Q: *What is the advantage of using ORM over JDBC/SQL?*
A: It reduces manual SQL, avoids boilerplate code, ensures object-database mapping consistency, and simplifies CRUD operations.

Advantages of ORM (Object-Relational Mapping)

Definition / Concept:
Advantages are the key benefits developers get by using ORM to interact with databases through objects instead of manual SQL.

Key Points / Features:

- **Reduces Boilerplate Code:** No need to write repetitive SQL queries.
- **Faster Development:** CRUD operations can be performed with simple object methods.
- **Database Abstraction:** Makes switching databases easier without changing much code.
- **Maintains Consistency:** Object-oriented code stays synchronized with database tables.
- **Improves Security:** Prevents SQL injection by using parameterized queries internally.

Purpose / Importance:

- Helps freshers and developers focus on business logic rather than SQL syntax.
- Improves productivity and reduces chances of database-related errors.

Real-life Example / Analogy:
Like using **apps to control your smart home** instead of manually flipping switches for every device—ORM handles all the “manual work” behind the scenes.

Interview Tip / Common Question:
Q: *Why would you use ORM in a project?*
A: It speeds up development, reduces SQL errors, provides database independence, and simplifies code maintenance.

JPA (Java Persistence API)

Definition / Concept:
JPA is a **Java specification** that defines a standard way to manage relational data in Java applications using ORM concepts.

Key Points / Features:

- Provides **object-relational mapping** for Java objects to database tables.
- Standardizes persistence, so you can switch implementations easily (e.g., Hibernate, EclipseLink).
- Supports **CRUD operations, JPQL (Java Persistence Query Language), and caching**.
- Annotations like @Entity, @Table, @Id, and @OneToMany simplify mapping.

Purpose / Importance:

- Ensures **database-independent** code using standard Java APIs.
- Helps developers focus on business logic rather than writing database-specific queries.
- Widely used in enterprise-level Java applications for consistent ORM implementation.

Real-life Example / Analogy:

- JPA is like a **universal adapter**: no matter which database you use, you interact with objects the same way.
- Example: EntityManager.persist(user) saves a User object to any underlying database.

Interview Tip / Common Question:
Q: *What is the difference between JPA and Hibernate?*
A: JPA is a **specification**, whereas Hibernate is an **implementation** of that specification.

JPA vs Hibernate

Definition / Concept:
JPA: A **Java specification** for ORM; defines rules and interfaces for mapping Java objects to database tables.
Hibernate: A **framework/implementation** of JPA that provides the actual functionality to interact with databases.

Key Points / Features:

Feature	**JPA (Specification)**	**Hibernate (Implementation)**
Type	Standard API / Specification	Framework / Library
Purpose	Defines ORM rules and guidelines	Implements ORM functionality
Dependency	No external dependency needed	Requires JPA or can be used standalone
Features	Basic ORM, JPQL, Entity management	Advanced ORM features like caching, lazy loading, batch processing
Flexibility	Can switch implementations easily	Tied to Hibernate unless using pure JPA API

Purpose / Importance:

Understanding the difference helps interviewers know if you can **choose the right tool** for projects.

Freshers must know: **Use JPA for standardization, Hibernate for advanced features.**

Real-life Example / Analogy:

Think of **JPA as the blueprint** of a house, and **Hibernate as the contractor** who builds it.

Interview Tip / Common Question:

Q: *Why use Hibernate if JPA already exists?*

A: Hibernate provides **extra features** not in JPA, like caching, custom SQL, and better performance tuning.

Spring Data Repository

Definition / Concept:

Spring Data Repository is an **interface-based mechanism** in Spring that simplifies database operations by providing ready-made CRUD and query methods without writing boilerplate code.

Key Points / Features:

- Extends interfaces like CrudRepository, JpaRepository, or PagingAndSortingRepository.
- Provides **built-in CRUD methods**: save(), findById(), findAll(), delete().
- Supports **custom queries** using method names (findByName) or @Query annotation.
- Reduces boilerplate code and integrates seamlessly with Spring Boot and JPA.

Purpose / Importance:

- Speeds up development for freshers and developers.
- Avoids repetitive DAO implementation; you just define the interface, and Spring provides the implementation at runtime.

Real-life Example / Analogy:

Like **pre-packaged tools in a toolkit**: instead of building every tool yourself, you use ready-made tools to do the job faster.

Example: userRepository.findByEmail("abc@mail.com") fetches the user without writing SQL.

Interview Tip / Common Question:
Q: *What is the difference between CrudRepository and JpaRepository?*
A: JpaRepository extends CrudRepository and provides **additional features** like batch operations, pagination, and sorting.

CrudRepository (Spring Data)

Definition / Concept:
CrudRepository is a **Spring Data interface** that provides generic CRUD (Create, Read, Update, Delete) operations for entities without writing implementation code.

Key Points / Features:

- Provides methods like save(), findById(), findAll(), deleteById(), and count().
- Works with any entity class by specifying the entity type and ID type.
- Automatically implemented by Spring at runtime; no need for manual DAO coding.
- Can be extended to add **custom query methods**.

Purpose / Importance:

- Simplifies data access for freshers by **eliminating boilerplate code**.
- Promotes rapid development in Spring Boot applications.

Real-life Example / Analogy:
Like a **universal remote**: it works with any device (entity) to perform standard operations without configuring each one manually.
Example: userRepository.deleteById(101L) deletes a user without writing SQL.

Interview Tip / Common Question:
Q: *What is the difference between CrudRepository and JpaRepository?*
A: JpaRepository extends CrudRepository and adds **pagination, sorting, and batch operations**.

JpaRepository (Spring Data)

Definition / Concept:
JpaRepository is a **Spring Data interface** that extends CrudRepository and provides **JPA-specific methods** for advanced database operations like pagination and sorting.

Key Points / Features:

- Inherits all **CRUD methods** from CrudRepository.
- Adds **JPA-specific features** like findAll(Pageable pageable) for pagination and sorting.
- Supports **batch operations** and flushing changes to the database.
- Can define **custom queries** using method names or @Query.

Purpose / Importance:

- Makes database operations more powerful and flexible for enterprise applications.
- Reduces manual query handling for freshers while supporting complex operations.

Real-life Example / Analogy:
Like a **premium toolkit**: it has all basic tools plus advanced tools for bigger or complex tasks.
Example: userRepository.findAll(PageRequest.of(0, 10, Sort.by("name"))) fetches first 10 users sorted by name.

Interview Tip / Common Question:
Q: *When should you use JpaRepository instead of CrudRepository?*
A: Use JpaRepository when you need **pagination, sorting, or batch operations** in addition to basic CRUD.

@Entity and @Table (JPA Annotations)

Definition / Concept:

- **@Entity:** Marks a Java class as a **JPA entity**, meaning it will be mapped to a database table.
- **@Table:** Specifies the **database table name** to which the entity is mapped (optional; defaults to class name).

Key Points / Features:

- @Entity is mandatory for every persistent class.
- @Table(name = "table_name") allows mapping to a **custom table name**.
- Both annotations help JPA/Hibernate manage ORM efficiently.
- Can include other attributes in @Table like schema, uniqueConstraints, and indexes.

Purpose / Importance:

- Helps map Java objects to relational database tables, forming the core of ORM.
- Ensures JPA knows which classes to **persist, retrieve, and manage**.

Real-life Example / Analogy:
Think of @Entity as **registering a class for database storage**, and @Table as **giving it a custom storage box name.**

Example:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    private Long id;
    private String name;
}
```

Interview Tip / Common Question:
Q: *What happens if you don’t use @Table?*
A: JPA will use the **class name as the default table name.**

Q: *Is @Entity mandatory?*
A: Yes, without @Entity, the class cannot be mapped to a database table.

@Id and @GeneratedValue (JPA Annotations)

Definition / Concept:

- **@Id:** Marks a field as the **primary key** of a JPA entity.
- **@GeneratedValue:** Specifies how the **primary key value is automatically generated** by the database or JPA provider.

Key Points / Features:

- @Id is **mandatory** for every entity to identify each record uniquely.
- @GeneratedValue strategies include:
  - AUTO – JPA chooses the generation strategy automatically.
  - IDENTITY – Database generates a unique value (e.g., auto-increment).
  - SEQUENCE – Uses a database sequence for generation.
  - TABLE – Uses a special table to generate IDs.
- Can be combined with @Column for additional constraints.

Purpose / Importance:

- Ensures each entity has a **unique identifier** for CRUD operations.
- Reduces manual effort in generating unique primary key values.

Real-life Example / Analogy:
Think of @Id as a **unique ID card** for each person, and @GeneratedValue as the **system automatically issuing ID numbers**.

Example:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
}
```

Interview Tip / Common Question:
Q: *What is the difference between GenerationType.AUTO and GenerationType.IDENTITY?*
A: AUTO lets JPA decide; IDENTITY relies on the database auto-increment feature.

Q: *Is @Id mandatory for JPA entities?*
A: Yes, without @Id, JPA cannot track entity uniqueness.

@Column (JPA Annotation)

Definition / Concept:
@Column is used to **map a Java class field to a database table column** and define column-specific properties.

Key Points / Features:

- Optional if **field name matches column name**; otherwise, used to specify custom column names.
- Common attributes:
  - name – custom column name.
  - nullable – allows/disallows null values.
  - length – sets maximum length for string columns.
  - unique – enforces unique values.
  - columnDefinition – defines SQL data type explicitly.
- Works with all basic types (String, int, Date, etc.).

Purpose / Importance:

- Provides **fine-grained control** over how fields are stored in the database.
- Helps maintain **data integrity and constraints** at the database level.

Real-life Example / Analogy:
Like **labeling a box with specific rules** (size, uniqueness, optional/mandatory).

Example:

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "user_name", nullable = false, length = 50)
    private String name;
}
```

Interview Tip / Common Question:
Q: *Is @Column mandatory?*
A: No, if the field name matches the column name, JPA maps it automatically.

Q: *What is the use of nullable = false?*
A: It ensures the column **cannot have null values**, enforcing data integrity.

@OneToOne (JPA Annotation)

Definition / Concept:
@OneToOne defines a **one-to-one relationship** between two entities, where **one record in an entity maps to exactly one record in another entity**.

Key Points / Features:

- Can be **unidirectional** (only one entity knows about the other) or **bidirectional** (both entities reference each other).
- Often combined with @JoinColumn to specify the foreign key column.
- Helps enforce **data integrity** for closely linked entities.
- Example use cases: User ↔ UserProfile, Person ↔ Passport.

Purpose / Importance:

- Models real-world one-to-one relationships in the database.

- Reduces data redundancy and improves relational design.

Real-life Example / Analogy:
Like **a person and their passport**: one person has one passport, and one passport belongs to one person.

Example:

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "profile_id")
    private UserProfile profile;
}
```

Interview Tip / Common Question:
Q: *What is the difference between OneToOne and ManyToOne?*
A: OneToOne links exactly one record to one record; ManyToOne links **many records** to a single record in another entity.

Q: *Why use @JoinColumn?*
A: To define which column in the table acts as the **foreign key**.

@OneToMany (JPA Annotation)

Definition / Concept:
@OneToMany defines a **one-to-many relationship** between two entities, where **one record in an entity maps to multiple records in another entity.**

Key Points / Features:

- Often paired with @ManyToOne on the other side for **bidirectional relationships.**
- Uses mappedBy to indicate the **owning side** of the relationship.
- Can cascade operations (CascadeType.ALL) to child entities.
- Example use cases: Department ↔ Employees, Customer ↔ Orders.

Purpose / Importance:

- Models **real-world hierarchical relationships** in a database.
- Enables automatic management of child entities through the parent entity.

Real-life Example / Analogy:
Like **a classroom and students**: one classroom has many students, but each student belongs to only one classroom.

Example:

```
@Entity
public class Department {
    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Employee> employees;
}
```

Interview Tip / Common Question:

Q: *What is the difference between OneToMany and ManyToOne?*
A: OneToMany is the **parent side** (one record to many), ManyToOne is the **child side** (many records to one).

Q: *Why use mappedBy?*
A: To **avoid creating an extra join table** and define the owning side.

@ManyToMany (JPA Annotation)

Definition / Concept:
@ManyToMany defines a **many-to-many relationship** between two entities, where **multiple records in one entity relate to multiple records in another entity.**

Key Points / Features:

- Often requires a **join table** (@JoinTable) to manage the relationship.
- Can be **bidirectional** or **unidirectional.**
- Supports cascading operations if needed (CascadeType.ALL).
- Example use cases: Student ↔ Course, Author ↔ Book.

Purpose / Importance:

- Models **complex relationships** in databases where multiple associations exist.
- Avoids data duplication and maintains normalized design.

Real-life Example / Analogy:
Like **students and courses**: one student can enroll in many courses, and each course can have many students.

Example:

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

Interview Tip / Common Question:
Q: *Why do we need a join table in ManyToMany?*
A: Relational databases cannot store many-to-many directly, so a **join table** maps the associations.
Q: *Can ManyToMany be unidirectional?*
A: Yes, but usually bidirectional is preferred for querying from both sides.

@ManyToOne (JPA Annotation)

Definition / Concept:
@ManyToOne defines a **many-to-one relationship** between two entities, where **many records in one entity relate to a single record in another entity**.

Key Points / Features:

- Usually the **child side** of a @OneToMany relationship.
- Uses @JoinColumn to specify the **foreign key column** in the child table.
- Supports cascading operations if needed (CascadeType.ALL).
- Example use cases: Employee ↔ Department, Order ↔ Customer.

Purpose / Importance:

- Helps map **hierarchical relationships** in a database efficiently.
- Enables easy navigation from child entity to parent entity.

Real-life Example / Analogy:
Like **many students in one classroom**: each student belongs to exactly one classroom.

Example:

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

Interview Tip / Common Question:
Q: *What is the difference between ManyToOne and OneToMany?*
A: ManyToOne is the **child side** (many records pointing to one), OneToMany is the **parent side** (one record pointing to many).

Q: *Why use @JoinColumn?*
A: To define which column in the child table acts as the **foreign key** referencing the parent table.

EntityManager Methods in JPA

Definition / Concept:
The **EntityManager** manages entity lifecycle in JPA and provides methods to **perform CRUD operations** on entities in the database.

Key Methods & Features:

Method	Purpose / Description
persist()	Adds a **new entity** to the persistence context; inserts it into the database.
merge()	Updates an **existing entity** or attaches a detached entity to the persistence context.
remove()	Deletes an entity from the database.
flush()	Synchronizes the persistence context changes with the database **immediately**.

Purpose / Importance:
Allows fine-grained control of **entity lifecycle** (create, update, delete).
Ensures database consistency and efficient persistence management.

Real-life Example / Analogy:

- **persist():** Like adding a new book to a library catalog.
- **merge():** Like updating a book's details that were temporarily removed from the catalog.
- **remove():** Like removing a book permanently from the catalog.
- **flush():** Like sending all pending catalog changes to the main library database immediately.

Interview Tip / Common Question:
Q: *What is the difference between persist() and merge()?*
A: persist() is for **new entities**, merge() is for **updating detached/existing entities**.

Q: *Does remove() delete immediately?*
A: It removes from the persistence context; actual deletion occurs on **flush/commit**.

Fetch Types (JPA)

Definition / Concept:
Fetch type determines **how related entities are loaded from the database** when you access a parent entity in JPA.

Key Types / Features:

- **EAGER:** Loads the related entities **immediately** along with the parent entity.
- **LAZY:** Loads the related entities **on demand**, only when accessed.
- Default fetch types:
  - @OneToMany & @ManyToMany → **LAZY**
  - @ManyToOne & @OneToOne → **EAGER**

Purpose / Importance:

- Controls **performance** and **memory usage**.

- Avoids unnecessary data fetching, improving efficiency in large applications.

Real-life Example / Analogy:

- **EAGER:** Like buying a combo meal where everything is served immediately.
- **LAZY:** Like ordering items **only when you actually need them.**

Example:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "department")
private List<Employee> employees;
```

Interview Tip / Common Question:
Q: *What's the difference between LAZY and EAGER fetch?*
A: EAGER fetch **loads related entities immediately**, LAZY fetch **delays loading until accessed.**

Q: *Why is LAZY preferred for collections?*
A: To **avoid loading large datasets unnecessarily**, improving performance.

Lazy vs Eager Fetch (JPA)

Definition / Concept:
Fetch type defines **when related entities are loaded** from the database in JPA relationships.

Key Points / Features:

Fetch Type	Description	Default Usage
EAGER	Loads related entities **immediately** with the parent entity.	@OneToOne, @ManyToOne
LAZY	Loads related entities **only when accessed** (on demand).	@OneToMany, @ManyToMany

---

Purpose / Importance:

- **EAGER:** Ensures all necessary data is available immediately.
- **LAZY:** Optimizes **performance and memory** by loading data only when needed.

Real-life Example / Analogy:

- **EAGER:** Buying a **combo meal**—everything comes at once.
- **LAZY:** Ordering **a la carte items**—only when you want them.

Example:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "department")
private List<Employee> employees;
```

Interview Tip / Common Question:
Q: ***Which fetch type is better for collections?***
A: LAZY, to avoid loading large datasets unnecessarily.

Q: ***Can EAGER cause performance issues?***
A: Yes, because it **loads all related entities even if not needed**, potentially slowing queries.

@JoinColumn (JPA Annotation)

Definition / Concept:
@JoinColumn specifies the **foreign key column** in the database that is used to join two entities in a relationship (@OneToOne, @ManyToOne, etc.).

Key Points / Features:

- Defines the **name of the column** in the child table that references the parent table.
- Often used with @OneToOne or @ManyToOne relationships.
- Supports attributes like:
    - name – column name in the database.
    - referencedColumnName – column in the parent table being referenced (default is primary key).
    - nullable – allows/disallows null values.
- Ensures **proper foreign key mapping** between entities.

Purpose / Importance:

- Makes relationships **explicit** and ensures database integrity.
- Helps JPA understand how entities are linked for queries and joins.

Real-life Example / Analogy:
Like **labeling a pointer in a spreadsheet**: the child table column points to the parent table's unique identifier.

Example:

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "department_id", nullable = false)
    private Department department;
}
```

Interview Tip / Common Question:
Q: ***What happens if you don't use @JoinColumn?***
A: JPA will **generate a default foreign key column** using the parent entity name and primary key.

Q: ***Which relationships use @JoinColumn?***
A: Typically @OneToOne and @ManyToOne.

@JoinTable (JPA Annotation)

Definition / Concept:
@JoinTable is used to **define a join table** in the database that manages a **many-to-many relationship** between two entities.

Key Points / Features:

- Works with @ManyToMany relationships.
- Specifies the **name of the join table** and the **foreign key columns** referencing both entities.
- Common attributes:
  - name – name of the join table.
  - joinColumns – foreign key column(s) referencing the **current entity.**
  - inverseJoinColumns – foreign key column(s) referencing the **other entity.**
- Helps JPA **map many-to-many relationships** without duplicating data.

Purpose / Importance:

- Ensures **normalized database design** for many-to-many relationships.
- Makes queries and entity management more consistent and easier.

Real-life Example / Analogy:
Like a **link table in a school**: one student can enroll in many courses, and one course can have many students; the join table keeps track of who is in which course.

Example:

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

Interview Tip / Common Question:
Q: ***Why do we need @JoinTable in ManyToMany?***
A: Relational databases cannot store many-to-many relationships directly, so **a join table maps the associations.**

Q: ***What is the difference between @JoinColumn and @JoinTable?***
A: @JoinColumn is for **single foreign keys** (OneToOne or ManyToOne), @JoinTable is for **mapping ManyToMany relationships.**

Persisting Objects into Database (JPA)

Definition / Concept:
Persisting is the process of **saving Java objects (entities) into a database** using JPA.

Key Points / Steps:

- **Create an EntityManager:** Interface to interact with the persistence context.
- **Begin Transaction:** Start a transaction using em.getTransaction().begin().

- **Persist Object:** Use em.persist(entity) to save a new object.
- **Commit Transaction:** Finalize changes using em.getTransaction().commit().
- **Close EntityManager:** Release resources after operation.

Purpose / Importance:

- Converts **Java objects into database records** automatically.
- Reduces manual SQL writing and ensures **data consistency**.

Real-life Example / Analogy:
Like **registering a new student in a school system**: the student object is entered into the database record.

Example:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

User user = new User();
user.setName("Shaik Gaffoor");
user.setEmail("shaik@example.com");

em.persist(user);  // Persist object into DB

em.getTransaction().commit();
em.close();
emf.close();
```

Interview Tip / Common Question:
Q: ***What is the difference between persist() and merge()?***
A: persist() is for **new entities**, merge() is for **updating detached/existing entities**.
Q: ***Do you need a transaction to persist objects?***
A: Yes, **all persist operations must be within a transaction** in JPA.

Transient and Persistent States (JPA Entity States)

Definition / Concept:
JPA entities go through **different states** depending on their interaction with the persistence context:

- **Transient:** Object exists in memory but **not associated** with the database.
- **Persistent:** Object is **managed by JPA** and changes are synchronized with the database.

Key Points / Features:

State	Description	Example Method
Transient	Newly created object, not yet saved in DB, not tracked by EntityManager.	new User()
Persistent	Object associated with EntityManager; changes automatically saved.	em.persist(user)

State	Description	Example Method
Detached	Was persistent, but EntityManager is closed or object is removed from context.	em.detach(user)

Purpose / Importance:

- Helps JPA **manage entity lifecycle** efficiently.
- Freshers should understand state changes to avoid **unexpected database behavior**.

Real-life Example / Analogy:

- **Transient:** Writing a document on a notepad but not saving it.
- **Persistent:** Saving the document in Google Drive, automatically updated.
- **Detached:** Document saved but you closed Google Drive; changes are no longer tracked.

Interview Tip / Common Question:
Q: ***What happens if you modify a transient object?***
A: Changes are **not reflected in the database** until persisted.

Q: ***How to convert a detached entity back to persistent?***
A: Use merge() to attach it back to the persistence context.

Detached Objects (JPA Entity State)

Definition / Concept:
A **detached object** is an entity that was once **persistent (managed by EntityManager)** but is now **disconnected from the persistence context**.

Key Points / Features:

- Changes to a detached object are **not automatically synchronized** with the database.
- Can be **reattached** to the persistence context using merge().
- Often occurs when **EntityManager is closed** or detach() is called explicitly.
- Useful to **transfer entities across layers** (like from service to UI) without keeping EntityManager open.

Purpose / Importance:

- Helps **manage entity lifecycle** and control when database updates happen.
- Avoids keeping **EntityManager open for long periods**, improving performance.

Real-life Example / Analogy:
Like a **checked-out library book**: it’s no longer in the library system (EntityManager), but you can return it (merge) to synchronize it.

Example:

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

User user = em.find(User.class, 1L); // Persistent
em.getTransaction().commit();
em.close(); // User is now detached
```

```
user.setName("Updated Name"); // Changes not tracked automatically

em = emf.createEntityManager();
em.getTransaction().begin();
em.merge(user); // Reattach and persist changes
em.getTransaction().commit();
em.close();
```

Interview Tip / Common Question:
Q: ***What is the difference between detached and persistent objects?***
A: Persistent objects are **managed** by JPA; detached objects are **not tracked.**

Q: ***How do you save changes to a detached object?***
A: Use merge() to **reattach it to the persistence context**.

Removed State (JPA Entity State)

Definition / Concept:
A **removed object** is an entity that is **marked for deletion** from the database. It is still in the persistence context until the transaction is committed.

Key Points / Features:

- Entity is **scheduled for deletion** using em.remove(entity).
- Still exists in memory until the transaction **commits or flushes.**
- Cannot be modified meaningfully after removal; changes won’t be saved.
- Transition from **Persistent** → **Removed** when remove() is called.

Purpose / Importance:

- Provides a **controlled way to delete entities** while maintaining transactional integrity.
- Ensures JPA manages **entity lifecycle** properly and avoids inconsistent deletes.

Real-life Example / Analogy:
Like **marking a book for removal from a library catalog**: it’s scheduled to be deleted but still on the shelf until processed.

Example:

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

User user = em.find(User.class, 1L); // Persistent
em.remove(user); // Marked as removed

em.getTransaction().commit(); // Actually deleted from DB
em.close();
```

Interview Tip / Common Question:
Q: ***What happens if you modify a removed entity?***
A: Changes **won’t be saved**; the entity is scheduled for deletion.
Q: ***Which state comes before Removed?***
A: Persistent state; only persistent entities can be removed.

Persistent State & EntityManager Methods

Persistent State (Definition / Concept):

- An entity is **persistent** when it is **managed by the EntityManager**.
- Any changes to the entity are **automatically synchronized** with the database at commit or flush.

Common EntityManager Methods with Persistent Entities:

Method	Purpose / Description
persist()	Makes a **transient object persistent**; inserts it into the database.
merge()	Updates a **detached object** or existing entity in the persistence context.
remove()	Marks a persistent entity for **deletion** from the database.
find()	Retrieves an entity from the database by its **primary key**.

Purpose / Importance:

- Ensures **CRUD operations** can be performed efficiently.
- Manages entity lifecycle automatically and maintains **database consistency**.

Real-life Example / Analogy:

- **persist():** Adding a new book to a library catalog.
- **merge():** Updating a previously checked-out book and returning it.
- **remove():** Marking a book for deletion from the catalog.
- **find():** Searching the library catalog for a book by its ID.

Example Code:

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

// Persist new object
User user = new User();
user.setName("Shaik Gaffoor");
em.persist(user); // Transient -> Persistent

// Find an object
User existingUser = em.find(User.class, 1L);

// Merge detached object
existingUser.setName("Updated Name");
em.merge(existingUser);

// Remove an object
em.remove(existingUser);

em.getTransaction().commit();
em.close();
```

Interview Tip / Common Question:

Q: ***What's the difference between persist() and merge()?***
A: persist() = new entity; merge() = detached/existing entity.

Q: ***Do you need transactions for these operations?***
A: Yes, **all persist, merge, remove operations must be inside a transaction.**

JPQL (Java Persistence Query Language)

Definition / Concept:
JPQL is a **query language provided by JPA** to perform database operations on **entities**, not directly on database tables.

- It is **similar to SQL** but works with **Java classes and their attributes**.

Key Points / Features:

- Operates on **entities and their relationships**, not tables.
- Supports SELECT, UPDATE, DELETE queries.
- Uses **entity names and field names**, not table or column names.
- Can include **WHERE, JOIN, ORDER BY, GROUP BY** clauses.

Basic Syntax / Examples:
Select all records:
```
SELECT u FROM User u
```
Select with condition:
```
SELECT u FROM User u WHERE u.name = 'Shaik'
```
Delete query:
```
DELETE FROM User u WHERE u.id = 1
```
Update query:
```
UPDATE User u SET u.name = 'Gaffoor' WHERE u.id = 1
```
Join query:
```
SELECT o FROM Order o JOIN o.customer c WHERE c.name = 'Shaik'
```

Purpose / Importance:

- Allows **database-independent queries** using **entity objects**, making code portable.
- Essential for fresher-level interviews on **JPA and ORM concepts.**

Real-life Example / Analogy:
Like **asking questions about a list of objects in Java** instead of manually querying the database tables.

Interview Tip / Common Question:
Q: ***What is the difference between JPQL and SQL?***
A: JPQL works on **entities (classes/fields)**; SQL works on **tables/columns**.

Q: ***Can JPQL handle relationships?***
A: Yes, JPQL supports **joins and navigation of entity relationships**.

Named Queries (JPA)

Definition / Concept:

- A **Named Query** is a **predefined, reusable JPQL query** defined at the entity level using annotations.
- Helps in **centralizing queries** for better readability, maintainability, and performance.

Key Points / Features:

- Declared using @NamedQuery (single) or @NamedQueries (multiple) at the **entity class level.**

- Can be **referenced by name** in the code, avoiding repeated query strings.
- Supports **JPQL syntax** including SELECT, UPDATE, DELETE, and joins.
- Improves **performance** because queries are **precompiled** at startup.

Purpose / Importance:

- Makes queries **reusable and easy to maintain.**
- Reduces **runtime query errors** and improves code clarity.

Real-life Example / Analogy:
Like **creating a shortcut** for a frequently used query: you name it once and use it anywhere without rewriting.
Example:

```
@Entity
@NamedQuery(
    name = "User.findByName",
    query = "SELECT u FROM User u WHERE u.name = :name"
)
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
}
```

Using the named query:

```
User user = em.createNamedQuery("User.findByName", User.class)
             .setParameter("name", "Shaik")
             .getSingleResult();
```

Interview Tip / Common Question:
Q: ***What is the difference between Named Query and dynamic query?***
A: Named Query is **predefined and reusable**, dynamic query is **built at runtime** using createQuery().

Q: ***Where are Named Queries defined?***
A: At the **entity class level** using @NamedQuery or @NamedQueries.

Custom Query Methods (Spring Data JPA)

Definition / Concept:

- Custom Query Methods allow you to **define queries directly in repository interfaces** by **following naming conventions** or using @Query.
- Eliminates the need to write **implementation code** for simple queries.

Key Points / Features:

- **Method Name Query:** Spring Data parses the method name and generates the query automatically.
    - Examples: findByName(String name), findByPriceGreaterThan(Double price)
- **@Query Annotation:** Allows **writing JPQL or SQL** queries explicitly.
    - Example: @Query("SELECT u FROM User u WHERE u.email = ?1")
- Supports **pagination, sorting, and dynamic queries.**
- Can return **entity objects, lists, or projections.**

Purpose / Importance:

- Reduces boilerplate code and speeds up development.
- Ensures queries are **type-safe and easier to maintain**.

Real-life Example / Analogy:
Like **asking a question by its exact pattern** instead of manually searching through a large database.

Example using method name:

```
public interface UserRepository extends JpaRepository<User, Long> {
       List<User> findByName(String name);
}
```

Example using @Query:

```
@Query("SELECT u FROM User u WHERE u.email = :email")
User findUserByEmail(@Param("email") String email);
```

Interview Tip / Common Question:
Q: ***What is the difference between method name query and @Query?***
A: Method name query **derives query from method name automatically**, @Query allows **explicit custom queries**.

Q: ***Can you use joins in custom query methods?***
A: Yes, using @Query with JPQL.

@Query Annotation (Spring Data JPA)

Definition / Concept:

- @Query allows you to **define custom JPQL or SQL queries directly on repository methods**.
- Useful when **method name derivation is not enough** for complex queries.

Key Points / Features:

- Can write **JPQL** (entity-based) or **native SQL** queries.
- Supports **named parameters** (:param) or **positional parameters** (?1).
- Can be used with **update/delete operations** using @Modifying.
- Works with **List, Optional, single entity, projections** as return types.

Purpose / Importance:

- Handles **complex queries** not achievable by method name conventions.
- Improves code readability and **centralizes queries in repository interfaces**.

Real-life Example / Analogy:
Like **writing a specific instruction** for retrieving exactly what you want from a large dataset.

Examples:
JPQL Query:

```
@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmail(@Param("email") String email);
```

Native SQL Query:

```
@Query(value = "SELECT * FROM users u WHERE u.email = ?1", nativeQuery = true)
```

```
User findByEmailNative(String email);
```

Update/Delete with @Modifying:

```
@Modifying
@Query("UPDATE User u SET u.status = :status WHERE u.id = :id")
int updateUserStatus(@Param("id") Long id, @Param("status") String status);
```

Interview Tip / Common Questions:

Q: ***Difference between @Query and method name query?***
A: @Query = explicit, can handle **complex JPQL/SQL**; method name = automatic derivation.

Q: ***What is @Modifying used for?***
A: To perform **update/delete operations** in custom queries.

Q: ***Can @Query return custom DTOs?***
A: Yes, by using **constructor expressions** in JPQL.

Native Queries (Spring Data JPA)

Definition / Concept:

- A **Native Query** is a **SQL query written directly for the database** instead of JPQL.
- Useful when JPQL cannot handle **complex database-specific operations.**

Key Points / Features:

- Use @Query with nativeQuery = true.
- Can return **entities, lists, projections, or scalar values**.
- Supports **positional (?1) or named (:param) parameters**.
- Works with all SQL capabilities, including joins, unions, database functions, etc.

Purpose / Importance:

- Allows execution of **database-specific SQL** while still using Spring Data repositories.
- Useful for **performance optimization** or when JPQL is limited.

Real-life Example / Analogy:
Like **writing a direct SQL command** in the database instead of using higher-level abstractions.

Example:

```
public interface UserRepository extends JpaRepository<User, Long> {

    // Native query with named parameter
    @Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)
    User findByEmailNative(@Param("email") String email);

    // Native query with positional parameter
    @Query(value = "SELECT * FROM users WHERE status = ?1", nativeQuery = true)
    List<User> findByStatusNative(String status);
}
```

Interview Tip / Common Questions:

Q: ***Difference between JPQL and Native Query?***

- JPQL = database-independent, entity-based.
- Native Query = **database-specific SQL**.

Q: ***Can Native Queries return DTOs?***
A: Yes, using @SqlResultSetMapping or constructor expressions in JPA.

Q: ***Why use Native Query over JPQL?***
A: For **complex joins, database functions, or performance optimization** not supported by JPQL.

Specifications (Spring Data JPA)

Definition / Concept:

- **Specifications** are used to create **dynamic queries** in Spring Data JPA using the **Criteria API**.
- Allows building queries **programmatically**, especially when query conditions are **optional or variable**.

Key Points / Features:

- Implement Specification<T> interface for the entity.
- Combines multiple conditions using **and(), or(), not()**.
- Works seamlessly with JpaSpecificationExecutor<T> repository interface.
- Returns **type-safe queries** that are database-independent.

Purpose / Importance:

- Ideal for **search filters**, **dynamic query building**, and **complex criteria queries**.
- Avoids **hardcoding JPQL or SQL strings**, improving maintainability.

Real-life Example / Analogy:
Like **filtering products on an e-commerce site**: price range, category, brand, availability—conditions vary dynamically.

Example:

```
// Entity: User
public class UserSpecification {
    public static Specification<User> hasName(String name) {
        return (root, query, cb) -> cb.equal(root.get("name"), name);
    }

    public static Specification<User> hasStatus(String status) {
        return (root, query, cb) -> cb.equal(root.get("status"), status);
    }
}

// Repository
public interface UserRepository extends JpaRepository<User, Long>,
JpaSpecificationExecutor<User> {}

// Using Specification
```

```
Specification<User> spec = Specification.where(UserSpecification.hasName("Shaik"))
                                        .and(UserSpecification.hasStatus("Active"));
List<User> users = userRepository.findAll(spec);
```

Interview Tip / Common Questions:

Q: ***When to use Specification over JPQL/@Query?***
A: When **query conditions are dynamic** and **cannot be known at compile time**.

Q: ***Can Specifications be combined?***
A: Yes, using .and(), .or() for multiple conditions.

Q: ***What interface is needed to support Specifications?***
A: JpaSpecificationExecutor<T>

Advanced JPA Concepts

Definition / Concept:
Advanced JPA covers **features beyond basic CRUD**, enabling efficient **entity management, performance optimization, and complex queries**.

Key Topics / Features:

Topic	Description
JPQL (Java Persistence Query Language)	Entity-based query language for database-independent queries.
Named Queries	Predefined, reusable JPQL queries defined at entity level.
Native Queries	Direct SQL queries executed on the database; bypass JPQL limitations.
Custom Query Methods	Spring Data feature allowing query creation from **method names** or @Query.
Fetch Types (LAZY/EAGER)	Controls how related entities are loaded (on demand vs immediate).
Cascade Types	Defines **how operations propagate** from parent to child entities (ALL, PERSIST, REMOVE, etc.).
Entity States	Lifecycle states of entities: **Transient, Persistent, Detached, Removed.**
Optimistic & Pessimistic Locking	Handles concurrent access: @Version for optimistic, LockModeType for pessimistic.
Embeddable / @Embedded	Reusable value objects embedded in entities, e.g., Address object.
Inheritance Mapping	Map **class hierarchy** to tables: Single Table, Joined, Table per Class strategies.
Specifications / Criteria API	Build **dynamic, type-safe queries** programmatically.
Second-level Cache	Store entities in **shared cache** to improve performance (e.g., Hibernate cache).
Entity Graphs	Define **which associations to fetch** dynamically to avoid N+1 problem.

Purpose / Importance:

- Makes JPA **scalable, efficient, and maintainable.**
- Essential for **enterprise-level applications** where performance, complex queries, and transactions matter.

Real-life Example / Analogy:
Like **upgrading from basic Excel operations to macros, pivot tables, and dashboards**: basic CRUD works, but advanced features make handling large, complex datasets efficient.

Interview Tip / Common Questions:
Q: ***What are Cascade Types in JPA?***
A: Define **how parent entity operations affect child entities** (PERSIST, MERGE, REMOVE, REFRESH, DETACH, ALL).

Q: ***Difference between LAZY and EAGER fetch?***
A: LAZY = load on access; EAGER = load immediately.

Q: ***What is Optimistic Locking?***
A: Prevents concurrent updates using a **version field** (@Version).

Pagination & Sorting (Spring Data JPA)

Definition / Concept:

- **Pagination**: Retrieve data in **chunks/pages** instead of loading all records at once.
- **Sorting**: Retrieve data in a **specific order** (ascending or descending).
- Both are important for **performance and user experience** in large datasets.

Key Points / Features:

- Use Pageable for pagination and Sort for sorting.
- Spring Data repositories like JpaRepository and PagingAndSortingRepository support these features.
- Can combine **pagination + sorting** in a single query.
- Helps **reduce memory usage** and **improves query efficiency**.

Purpose / Importance:

- Efficiently handle **large data tables**.
- Essential for **enterprise applications** with potentially thousands of records.

Real-life Example / Analogy:
Like browsing **Google search results**: 10 results per page (pagination), sorted by relevance (sorting).

Examples:
Pagination Only:

```
Pageable pageable = PageRequest.of(0, 5); // Page 0, 5 records per page
Page<User> usersPage = userRepository.findAll(pageable);
List<User> users = usersPage.getContent();
```

Sorting Only:

```
List<User> users = userRepository.findAll(Sort.by(Sort.Direction.ASC, "name"));
```

Pagination + Sorting:

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("name").ascending());
Page<User> usersPage = userRepository.findAll(pageable);
```

Interview Tip / Common Questions:
Q: ***Which repository interface supports pagination and sorting?***
A: PagingAndSortingRepository or JpaRepository (extends it).

Q: ***Difference between Page and List?***
A: Page contains **metadata** (total pages, total elements) + content; List is just the data.

Q: ***Can you sort by multiple fields?***
A: Yes, e.g., Sort.by("name").ascending().and(Sort.by("id").descending()).

Projections (Spring Data JPA)

Definition / Concept:

- **Projections** allow fetching **only specific fields** from an entity instead of the whole entity.
- Useful to **improve performance** and **reduce data transfer**.

Key Points / Features:

- Types of projections:
  - **Interface-based projection** – define an interface with getter methods.
  - **Class-based (DTO) projection** – use a **constructor expression** to map selected fields.
  - **Dynamic projections** – return different types based on method parameters.
- Can be used with **JPQL, @Query, or method name queries**.
- Reduces memory overhead when you don’t need full entity data.

Purpose / Importance:

- Optimizes queries for **read-heavy applications**.
- Makes code cleaner by **returning only relevant data**.

Real-life Example / Analogy:
Like **selecting only Name and Email from a user table** instead of fetching all columns including password, address, etc.

Examples:
Interface-based Projection:

```
public interface UserNameEmail {
String getName();
String getEmail();
}

public interface UserRepository extends JpaRepository<User, Long> {
List<UserNameEmail> findByStatus(String status);
}
```

DTO / Class-based Projection:

```
public class UserDTO {
private String name;
private String email;

public UserDTO(String name, String email) {
      this.name = name;
      this.email = email;
}
}

@Query("SELECT new com.example.UserDTO(u.name, u.email) FROM User u WHERE u.status =
:status")
List<UserDTO> findUserDTOByStatus(@Param("status") String status);
```

Interview Tip / Common Questions:
Q: ***Why use projections instead of fetching full entities?***
A: To **improve performance** and fetch only required fields.

Q: ***Can projections support nested objects?***
A: Yes, you can use **interface projections with nested getters** or **DTOs**.

Q: ***Difference between interface and DTO projection?***
A: Interface = simpler, read-only; DTO = can include **custom constructors and logic**.

Auditing (Spring Data JPA)

Definition / Concept:

- **Auditing** automatically tracks **who created/modified an entity and when.**
- Helps maintain **history and accountability** for database records.

Key Points / Features:

- Uses annotations like:
  - @CreatedDate – stores creation timestamp
  - @LastModifiedDate – stores last update timestamp
  - @CreatedBy – stores creator information
  - @LastModifiedBy – stores last modifier information
- Requires enabling auditing via @EnableJpaAuditing in the configuration class.
- Works with **entities implementing Auditable fields**.

Purpose / Importance:

- Useful in **enterprise applications** for compliance, debugging, and tracking changes.
- Reduces **manual effort** to maintain timestamps and user tracking.

Real-life Example / Analogy:
Like **Git commits**: shows who created/updated a file and when.

Example:
Entity with Auditing:

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {

@Id
@GeneratedValue
private Long id;
private String name;

@CreatedDate
private LocalDateTime createdDate;

@LastModifiedDate
private LocalDateTime lastModifiedDate;

@CreatedBy
private String createdBy;

@LastModifiedBy
private String lastModifiedBy;
}
```

Enable Auditing in Configuration:

```
@Configuration
@EnableJpaAuditing
public class JpaConfig {}
```

Interview Tip / Common Questions:
Q: ***How do you enable JPA Auditing?***
A: Using @EnableJpaAuditing and @EntityListeners(AuditingEntityListener.class).

Q: ***Which fields are commonly audited?***
A: Created/modified date (@CreatedDate, @LastModifiedDate) and user (@CreatedBy, @LastModifiedBy).

Q: ***Why use auditing instead of manually setting fields?***
A: Automates tracking, reduces errors, ensures **consistency and maintainability.**

Caching in Spring Data JPA

Definition / Concept:

- **Caching** stores frequently accessed data in memory to **reduce database calls** and improve performance.

Key Points / Features:

Two levels of caching in JPA/Hibernate:

- **First-level Cache**:
  - Default cache in **EntityManager.**
  - Scope: **per transaction/session.**
  - Automatically enabled; no configuration needed.
- **Second-level Cache**:
  - Shared cache for multiple sessions.
  - Requires configuration (e.g., **EhCache, Hazelcast**).
  - Improves **application-wide performance.**

Supports caching for **entities, collections, and queries.**

Real-life Analogy:

- Like **keeping frequently used books on your desk** instead of fetching them from the library every time.

Example:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
}
```

Interview Tip:

Q: *Difference between 1st-level and 2nd-level cache?*

- 1st-level: **EntityManager scoped**, auto-enabled.
- 2nd-level: **shared across sessions**, needs configuration.

Transaction Management in Spring Data JPA

Definition / Concept:

- A **transaction** is a **unit of work** where multiple database operations are executed **atomically**.

Key Points / Features:

- Spring provides @Transactional annotation to manage transactions.
- Supports **rollback** on exceptions.
- Transactions can be **read-only** for optimization.
- Can be applied at **method or class level**.

Purpose / Importance:

- Ensures **data consistency** (ACID properties).
- Prevents **partial updates** in case of errors.

Real-life Analogy:

Like **money transfer between bank accounts**: debit and credit must happen together, otherwise rollback.

Example:

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void updateUserStatus(Long id, String status) {
        User user = userRepository.findById(id).orElseThrow();
        user.setStatus(status);
        // changes automatically committed at the end of transaction
    }
}
```

Interview Tip:
Q: ***What happens if an exception occurs in a @Transactional method?***
A: Transaction **rolls back automatically** by default for unchecked exceptions.

Q: ***Difference between programmatic and declarative transaction management?***

- Declarative: using @Transactional (preferred).
- Programmatic: using TransactionTemplate or PlatformTransactionManager.

JPA & Spring Data Review Cheat Sheet

Concept / Topic	**Definition / Purpose**	**Key Points / Features**	**Example / Analogy**	**Interview Tip**
ORM	Object-Relational Mapping – maps Java objects to DB tables	Reduces SQL writing, improves productivity, ensures consistency	Like mapping a **Java User object to a users table**	Q: Advantages of ORM? A: Productivity, maintainability, DB-independent
JPA	Java Persistence API – standard for ORM in Java	Works with entities, provides EntityManager for DB ops	Like **JPA is a toolkit** to manage objects in DB	Q: Difference JPA vs Hibernate? JPA = spec, Hibernate = implementation
Entity & Table (@Entity, @Table)	Marks a class as DB entity and maps to table	@Table(name="users")	User class mapped to users table	Q: Can you skip @Table? Yes, defaults to class name
Primary Key (@Id, @GeneratedValue)	Identifies unique entity records	GenerationType.IDENTITY, SEQUENCE	ID of a user	Q: Difference between ID strategies?
Column (@Column)	Maps field to table column	nullable, length, unique	User email field → email column	Q: Default column name? Field name
Relationships (@OneToOne, @OneToMany, @ManyToOne, @ManyToMany)	Define entity associations	Use mappedBy, cascade, fetch	Library → Books	Q: Difference between LAZY and EAGER?
Entity States	Lifecycle of entities	Transient, Persistent, Detached, Removed	Like **object in memory vs DB**	Q: How to reattach detached? merge()
EntityManager Methods	CRUD & lifecycle operations	persist(), merge(), remove(), find()	Saving or deleting user	Q: Which require transaction? All write operations
JPQL	Entity-based query language	SELECT, UPDATE, DELETE	Like **SQL for objects**	Q: JPQL vs SQL? JPQL = entity-based
Named Queries	Predefined reusable queries	@NamedQuery at entity level	Shortcut for common queries	Q: Where are they defined? Entity class

Custom Query Methods	Derived queries from method names	findByName, findByPriceGreaterThan	Like **asking by pattern**	Q: When use @Query? Complex queries
@Query Annotation	Custom JPQL/SQL queries on repo methods	Named/positional params, @Modifying	Specific instructions for DB	Q: Difference method name vs @Query?
Native Queries	Direct SQL queries	nativeQuery=true	Database-specific query	Q: Why use native? For complex DB ops
Specifications	Dynamic, type-safe queries	Implement Specification<T>	Filtering like e-commerce search	Q: Interface needed? JpaSpecificationExecutor
Projections	Fetch only selected fields	Interface or DTO	Like selecting only name/email	Q: Interface vs DTO? DTO = constructor, Interface = getter-only
Pagination & Sorting	Retrieve pages of data & order results	Pageable, Sort	Google search results	Q: Page vs List? Page = content + metadata
Auditing	Track created/modified info	@CreatedDate, @LastModifiedDate, @CreatedBy, @LastModifiedBy	Like Git commits	Q: How enabled? @EnableJpaAuditing
Caching	Store frequently accessed entities in memory	1st-level (EntityManager), 2nd-level (shared)	Books on desk vs library	Q: 1st vs 2nd level cache? Scope & config
Transaction Management	Ensure atomic DB operations	@Transactional, rollback, read-only	Bank transfer analogy	Q: What happens on exception? Rollback

---

Purpose for Freshers:

- Understand **how JPA manages objects**, queries, and database operations.
- Learn **Spring Data features** to reduce boilerplate code.
- Prepare for **interview questions on lifecycle, queries, and advanced features.**
- Recognize **performance improvements**: caching, pagination, projections, auditing.
