✅ JPA & Hibernate Roadmap (with Topic Numbers)

No.	Topic
1	What is ORM, JPA, and Hibernate? Difference and Need
2	Setting up JPA with Spring Boot (starter, config, dependencies)
3	Entity, Table, and Primary Key Annotations (@Entity, @Table, @Id)
4	Field Mapping Annotations: @Column, @GeneratedValue, etc.
5	JPA Repository and CRUD Operations
6	Derived Query Methods (findBy, countBy, existsBy)
7	Custom Queries with @Query and Native SQL
8	JPQL (Java Persistence Query Language) Basics
9	Pagination and Sorting
10	Entity Relationships (OneToOne, OneToMany, ManyToOne, ManyToMany)
11	Cascade Types and fetch Strategies (EAGER, LAZY)
12	Embedded and Embeddable objects (@Embeddable)
13	Audit Fields (@CreatedDate, @LastModifiedDate)
14	Soft Delete (@SQLDelete, @Where)
15	Transactions with @Transactional
16	Difference between JPA, Hibernate, Spring Data JPA
17	Common Interview Questions (Grouped Summary)

✅ JPA & Hibernate – Topic 1: What is ORM, JPA, and Hibernate?

🔷 1. What is ORM?

✅ Definition:

ORM stands for Object Relational Mapping.

It is a technique to map Java objects to database tables and vice versa.

🧠 Why Do We Need ORM?

Without ORM:

java

Copy

Edit

// Old traditional way (JDBC)

Connection con = DriverManager.getConnection(...);

PreparedStatement ps = con.prepareStatement("INSERT INTO users VALUES (?, ?)");

ps.setString(1, "Viraj");

With ORM (JPA/Hibernate):

java

Copy

Edit

User user = new User("Viraj");

userRepository.save(user);

✅ ORM automates SQL handling — no need to write raw queries every time.

✅ Benefits of ORM:

Benefit Explanation

🔁 Automates SQL You don't have to write boilerplate SQL code

🧩 Cleaner Code Code is more readable and closer to object-oriented principles

🧪 Testable Easier to mock and test

🔄 Portable Works across databases (MySQL, Oracle, PostgreSQL)

💾 Caching & Performance Hibernate adds powerful caching features

🔷 2. What is JPA?

✅ Definition:

JPA = Java Persistence API (specification)

JPA is a standard (interface) given by Oracle to manage relational data in Java using ORM.

JPA doesn't provide an implementation — it just provides a set of interfaces and annotations.

🧩 Examples of JPA Providers:

Hibernate ✅ (most popular)

EclipseLink

OpenJPA

✅ What does JPA provide?

Annotations like @Entity, @Table, @Id, etc.

EntityManager interface for managing data

Query language (JPQL)

🔷 3. What is Hibernate?

✅ Definition:

Hibernate is the most commonly used implementation of JPA.

You can use Hibernate alone or with Spring Data JPA.

Hibernate adds extra features on top of JPA:

Automatic table generation

Caching

Dirty checking

Better performance tuning

🔄 JPA vs Hibernate vs Spring Data JPA

Feature JPA Hibernate Spring Data JPA

✅ What is it? Specification (interfaces) Implementation of JPA Spring module built on top of JPA

🛠️ Provides? Interfaces only Full working code (engine) Simplifies all JPA + Repository logic

📦 Need to write repo? Yes Yes No – Spring does it for you

🔶 Real-Life Example

Think of it like this:

Concept	Example
JPA	Car Manual
Hibernate	Car Engine (based on manual)
Spring Data JPA	Car with Auto-Driver

So instead of driving the car yourself (writing SQL), Spring does it automatically with @Repository interfaces.

🚫 Drawbacks of ORM

Drawback	Explanation
🐌 Performance	Slightly slower than native SQL if not optimized
❌ Complex Joins	For very complex queries, raw SQL may still be better
📚 Learning Curve	Learning annotations and configuration is required

🎯 Interview Questions

What is ORM?

What is JPA? How is it different from Hibernate?

Can you use Hibernate without JPA?

What is Spring Data JPA?

Why use ORM instead of JDBC?

✅ JPA & Hibernate – Topic 2: Setting Up JPA with Spring Boot

🔧 What Are We Doing?

To use **JPA + Hibernate** in Spring Boot, we must:

1. Add dependencies
2. Configure database connection
3. Create an entity
4. Create a repository
5. Test basic operations

✅ Step-by-Step Setup

1️⃣ Add Maven Dependencies

In pom.xml, add:

xml

CopyEdit

```
<dependencies>
  <!-- Spring Boot JPA Starter -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- MySQL Connector (or your DB) -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
  </dependency>
```

</dependencies>

✅ spring-boot-starter-data-jpa includes Hibernate (as default JPA provider)

2️⃣ Database Configuration in application.properties

properties

CopyEdit

Database config

spring.datasource.url=jdbc:mysql://localhost:3306/studentdb

spring.datasource.username=root

spring.datasource.password=your_password

JPA config

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

---

🔍 What is spring.jpa.hibernate.ddl-auto?

Value	Meaning
none	No schema changes
update	Updates table schema (non-destructive) ✅
create	Creates tables (deletes old data)
create-drop	Creates tables and drops them on shutdown
validate	Validates schema against entity mappings

⚠️ In real projects, use update for dev, but NEVER in production.

3️⃣ Create an Entity

java

CopyEdit

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Getters and setters
}
```

4️⃣ Create Repository Interface

java

CopyEdit

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

public interface StudentRepository extends JpaRepository<Student, Long> {
    // You can add custom query methods here
}
```

Spring Boot auto-creates the implementation at runtime.

5️⃣ Use Repository in a Service or Controller

java

CopyEdit

```
@Service
public class StudentService {

  @Autowired
  private StudentRepository repo;

  public List<Student> getAllStudents() {
    return repo.findAll();
  }

  public Student saveStudent(Student student) {
    return repo.save(student);
  }
}
```

✅ **How It Works**

1. Spring Boot **auto-configures Hibernate** based on dependencies.
2. JPA scans @Entity classes and maps them to tables.
3. You use JpaRepository to avoid writing SQL.
4. Spring handles database CRUD operations behind the scenes.

✅ **Benefits of Using JPA in Spring Boot**

Benefit	Description
📦 No DAO/Boilerplate	Just create interface, Spring does the rest
⚙️ Auto config	Spring Boot configures Hibernate & DB

Benefit	Description
📋 Easy testing	You can mock repos/services easily
🔄 Portable	Easy to switch between MySQL, PostgreSQL, etc.

🧠 Interview Questions

1. How do you integrate JPA in Spring Boot?
2. What is spring.jpa.hibernate.ddl-auto used for?
3. What does @Entity do?
4. What is the role of JpaRepository?
5. Can you explain how Spring Boot handles ORM setup?

✅ JPA & Hibernate – Topic 3: Entity, Table, and Primary Key Annotations

These annotations are the **foundation of JPA**, as they help Java classes map to database tables.

🔹 1. @Entity Annotation

✅ What is it?

Marks a class as a **JPA entity**, meaning the class is mapped to a table in the database.

java

CopyEdit

@Entity

public class Student {

  // fields

}

🧠 How It Works:

- JPA automatically creates a table for this class (if ddl-auto=update or create).
- Each instance of this class becomes a row in the table.

❗ Rules:

- Must have a no-arg constructor (public or protected).
- Must have at least one field with @Id.

🔹 2. @Table Annotation

✅ What is it?

Maps an entity to a specific **table name** in the database.

java

CopyEdit

@Entity

@Table(name = "students")

```
public class Student {
  // fields
}
```

🔍 **If You Skip It?**

JPA will use the class name as the table name (Student → student).

🔸 **Use cases:**

- If the table name in DB is different from the class name
- To define schema, catalog, unique constraints (if needed)

🔹 **3. @Id Annotation**

✅ **What is it?**

Marks the **primary key field** in the entity.

java

CopyEdit

```
@Id
private Long id;
```

Every entity **must have exactly one** @Id field.

🔹 **4. @GeneratedValue**

✅ **What is it?**

Tells JPA how to **generate the primary key** automatically.

java

CopyEdit

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

🔢 **Common Strategies:**

Strategy	Description
AUTO	JPA chooses the strategy
IDENTITY	Uses DB auto-increment (MySQL) ✅
SEQUENCE	Uses sequence object (Oracle/PostgreSQL)
TABLE	Uses a table to store key values (rare)

🧠 **Example Code**

java

CopyEdit

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "student_name", nullable = false)
    private String name;

    private String email;

    // Getters, setters, constructors
}
```

✅ Benefits of These Annotations

Feature	Benefit
@Entity	Auto-maps Java class to table
@Table	Custom table names, schema
@Id	Sets primary key
@GeneratedValue	Avoids manual ID generation

❗ Drawbacks / Notes

Limitation	Reason
Must follow conventions	Needs no-arg constructor, one @Id
Can throw runtime errors	If annotations are missing/misused
Primary key generation	Some DBs behave differently (e.g., Oracle prefers SEQUENCE)

🎯 Interview Questions

1. What is the use of @Entity?
2. Can a class work without @Entity in JPA?
3. Why is @Id required in every entity?
4. Difference between @Table and @Entity?
5. What does @GeneratedValue(strategy = GenerationType.IDENTITY) mean?

✅ JPA & Hibernate – Topic 4: Field Mapping Annotations

🔹 1. @Column Annotation

✅ What is it?

Specifies the **column** that a field should map to in the database.

java

CopyEdit

@Column(name = "student_name", nullable = false, length = 50)

private String name;

🔍 Common Attributes:

Attribute	Description
name	Column name in DB
nullable	Whether the column allows null
length	Max length (used for VARCHAR)
unique	Adds a unique constraint
updatable	Prevents updates if false
insertable	Exclude from insert queries if false

🧠 Example:

java

CopyEdit

@Column(name = "email", nullable = false, unique = true)

private String email;

Creates a **NOT NULL and UNIQUE email** column.

🔹 2. @Transient

✅ **What is it?**

Tells JPA **NOT to store this field** in the database.

java

CopyEdit

@Transient

private String tempPassword;

🧠 **Use Case:**

Useful for fields like passwords, computed values, or helper fields you don't want persisted.

🔹 **3. @Temporal (For legacy java.util.Date)**

✅ Use LocalDate, LocalDateTime from java.time.* in modern Java.

For older Java versions:

java

CopyEdit

@Temporal(TemporalType.DATE)

private Date birthDate;

Type	**Stores in DB as**
TemporalType.DATE	Only date (yyyy-mm-dd)
TemporalType.TIME	Only time (hh:mm:ss)
TemporalType.TIMESTAMP	Full date + time

🔹 **4. @Enumerated**

✅ **What is it?**

Used to map enum fields in your entity to the DB.

java

CopyEdit

public enum Gender { MALE, FEMALE }

@Enumerated(EnumType.STRING)

private Gender gender;

Type	Stored as
EnumType.ORDINAL	Integer (index) like 0, 1
EnumType.STRING	String (e.g., MALE) ✅

💡 Always prefer STRING for clarity and safety.

🔹 **5. @Lob**

✅ **What is it?**

Maps large content like **text** or **binary files**.

java

CopyEdit

@Lob

private String bio; // large text

@Lob

private byte[] image; // binary/image

✅ **Example with All:**

java

CopyEdit

@Entity

@Table(name = "students")

public class Student {

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "student_name", nullable = false, length = 50)
    private String name;

    @Column(unique = true)
    private String email;

    @Enumerated(EnumType.STRING)
    private Gender gender;

    @Transient
    private String tempField;

    @Lob
    private String longDescription;
}
```

✅ **Benefits of Field Mapping**

Feature	Benefit
Fine control	Choose column names, lengths, constraints
Customization	You control how each field maps
Optional fields	Mark with nullable, insertable, updatable
Prevent accidental saves	Use @Transient

❌ Drawbacks

Issue	Description
Manual work	Have to annotate each field
Misconfigurations	Wrong mappings can cause runtime exceptions
Schema mismatch	Can cause startup failures if column type/length mismatches DB

🎯 Interview Questions

1. What is the purpose of @Column?
2. Difference between @Column(nullable = false) and allowing null?
3. When to use @Transient?
4. How to store enums in JPA?
5. What is the difference between @Temporal and Java 8 DateTime API?

✅ JPA & Hibernate – Topic 5: Relationships in JPA

In real applications, entities are connected — like a **Student** having multiple **Courses**, or an **Order** having one **Customer**. JPA provides relationship annotations to map these connections.

🧩 Types of Relationships in JPA

Relationship Type	Example
One-to-One	A User has one Profile
One-to-Many	A Customer has many Orders
Many-to-One	Many Orders belong to one Customer
Many-to-Many	Students enrolled in many Courses and vice versa

---

🔹 1. @OneToOne Mapping

✅ Use Case:

One entity is associated with exactly one entity.

🔧 Example:

java

CopyEdit

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

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

- @JoinColumn defines the **foreign key column** (profile_id) in the User table.

🔹 **2. @OneToMany and @ManyToOne**

✅ **Use Case:**

- A Customer has many Orders → @OneToMany
- Each Order belongs to one Customer → @ManyToOne

🔧 **Example:**

java

CopyEdit

```
@Entity
public class Customer {
  @Id
  private Long id;

  @OneToMany(mappedBy = "customer")
  private List<Order> orders;
}

@Entity
public class Order {
  @Id
  private Long id;

  @ManyToOne
  @JoinColumn(name = "customer_id")
  private Customer customer;
}
```

- mappedBy = "customer" tells JPA that the foreign key lives in Order.

- @JoinColumn creates the customer_id foreign key in the Order table.

🔹 3. @ManyToMany Mapping

✅ Use Case:

Students and Courses — a student can join many courses, and each course can have many students.

🔧 Example:

java

CopyEdit

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

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

- Creates a **join table**: student_course
- Contains foreign keys: student_id, course_id

🔍 Lazy vs Eager Fetching

Fetch Type	Description
Lazy (default)	Related data is loaded **on-demand**
Eager	Related data is loaded **immediately**

java

CopyEdit

@OneToMany(fetch = FetchType.LAZY) // or EAGER

private List<Order> orders;

✅ Use **Lazy** loading to improve performance and avoid unnecessary queries.

✅ Benefits of Relationship Mapping

Feature	Benefit
Models real-world structure	Mirrors natural entity relationships
Cleaner SQL management	No need to write joins manually
Reduces boilerplate	Auto-handled via JPA
Improves reusability	Fetching nested data becomes easy

❌ Drawbacks

Issue	Description
Circular loading	Can cause infinite loops in JSON serialization
Lazy loading traps	May trigger extra queries or LazyInitializationException
Complex mappings	Many-to-many joins require intermediate tables and careful management

---

🎯 Interview Questions

1. What are the different types of relationships in JPA?
2. How does @OneToMany differ from @ManyToOne?

3. What is the purpose of mappedBy?
4. How does @JoinTable work in @ManyToMany?
5. What is lazy vs eager fetching?

✅ JPA & Hibernate – Topic 6: Cascade Types & Orphan Removal

In JPA, when two entities are related, we may want **operations** (like save, delete, update) to **automatically propagate** from one entity to its related entities.

That's where **Cascade Types** and **Orphan Removal** come in.

🔹 1. What is Cascade in JPA?

✅ Definition:

Cascade allows operations on a parent entity to automatically apply to **child or associated entities**.

java

CopyEdit

@OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)

private List<Order> orders;

💡 If you persist the Customer, the Order objects are also persisted automatically.

🔹 2. Cascade Types in JPA

Cascade Type	Description
PERSIST	Save child when parent is saved
MERGE	Update child when parent is updated
REMOVE	Delete child when parent is deleted
REFRESH	Refresh child from DB when parent is refreshed
DETACH	Detach child when parent is detached
ALL	Applies **all of the above** ✅

🔧 Example:

java

CopyEdit

```
@Entity
public class Customer {
   @Id
   private Long id;

   @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
   private List<Order> orders;
}
```

Now, saving a Customer will **also save its Orders** automatically.

🖌️ 3. What is Orphan Removal?

✅ **Definition:**

Automatically deletes **child entities** that are no longer referenced by the parent.

java

CopyEdit

```
@OneToMany(mappedBy = "customer", orphanRemoval = true)
private List<Order> orders;
```

❌ If an order is **removed from the list**, it will also be **deleted from the database**.

🔍 Example:

java

CopyEdit

```
customer.getOrders().remove(0); // also removes from DB if orphanRemoval = true
```

✅ When to Use Cascade & Orphan Removal?

Feature	Use Case
CascadeType.ALL	Parent-child lifecycle is tightly coupled (like Customer-Orders)
orphanRemoval = true	You want DB cleanup when child is removed from collection

---

❌ Drawbacks / Caution

Issue	Description
Too much cascading	Can accidentally delete too much (ex: ALL removes everything)
Performance issues	Large cascades can slow down operations
Orphan removal mistakes	Can delete records unintentionally if collection modified

---

🧠 Real-life Example

- Customer and their Orders: Cascade ALL + Orphan Removal
- User and Profile: OneToOne with cascade
- Post and Comments: OneToMany + orphan removal

🎯 Interview Questions

1. What is cascading in JPA?
2. Difference between CascadeType.ALL and CascadeType.PERSIST?
3. What does orphanRemoval = true do?
4. What are the dangers of using cascade incorrectly?
5. Can we use cascade and orphanRemoval together?

✅ JPA & Hibernate – Topic 7: JPQL & Native Queries

In JPA, we need to retrieve or manipulate data using queries. JPA provides two main ways:

1. **JPQL** – Java Persistence Query Language (Object-Oriented)
2. **Native SQL** – Actual SQL queries (Database-Oriented)

🔹 1. What is JPQL?

✅ Definition:

JPQL is like SQL, but it works with **entity objects** and their **fields**, not tables and columns.

java

CopyEdit

SELECT s FROM Student s WHERE s.name = 'John'

💡 Student is the Entity name, not the table name.

🔹 2. Syntax Example – JPQL

java

CopyEdit

@Query("SELECT s FROM Student s WHERE s.email = :email")

Student findByEmail(@Param("email") String email);

✅ Works on Entity and its field
❌ Not tied to actual DB table or column names

🔍 Common JPQL Clauses

Clause	Example
SELECT	SELECT u FROM User u
WHERE	WHERE u.age > 18

Clause	Example
ORDER BY	ORDER BY u.name ASC
JOIN	JOIN u.profile p
GROUP BY	GROUP BY u.role

🔹 3. Native Queries

✅ **Definition:**

Native queries are plain SQL queries executed **directly** on the database.

java

CopyEdit

@Query(value = "SELECT * FROM students WHERE email = ?1", nativeQuery = true)

Student getStudentByEmail(String email);

✅ Full control over SQL
✅ Use DB-specific features (like functions, joins)
❌ Not portable across databases

🔧 When to Use Which?

Type	When to Use
JPQL	When using standard object-based queries
Native SQL	When query is complex, uses DB features
Criteria API	When dynamic query building is needed

🔹 4. Named Queries (Optional)

You can define reusable JPQL queries on the entity:

java

CopyEdit

@Entity

```
@NamedQuery(
 name = "Student.findByName",
 query = "SELECT s FROM Student s WHERE s.name = :name"
)
public class Student { ... }
```

✅ Benefits of JPQL

Feature	Benefit
Object-focused	Works with Entities, not tables
Cleaner syntax	Short and readable
Portable	DB independent
Integrates well	Works perfectly with Spring Data JPA

❌ Drawbacks

Issue	Description
Limited power	Can't use complex SQL joins/functions
Slower	May not be optimized like hand-written SQL
Native-only features	Not usable in JPQL (like JSON columns, full-text search)

🎯 Interview Questions

1. What is the difference between JPQL and SQL?
2. How to write a native query in Spring Data JPA?
3. When should I prefer JPQL over native queries?
4. What is the benefit of using named queries?
5. What happens if entity field names are renamed but JPQL is not updated?

✅ JPA & Hibernate – Topic 8: Criteria API & Dynamic Queries

In many applications, you need to build **queries dynamically** based on user input. That's where the **Criteria API** comes in.

🔹 1. What is Criteria API?

✅ Definition:

The **Criteria API** is a Java-based API for building **type-safe**, **dynamic queries** using Java objects instead of strings.

💡 Useful when query parameters or conditions change at runtime.

🔧 Example (Basic Query):

java

CopyEdit

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Student> cq = cb.createQuery(Student.class);

Root<Student> root = cq.from(Student.class);

cq.select(root).where(cb.equal(root.get("name"), "John"));

List<Student> students = entityManager.createQuery(cq).getResultList();

- CriteriaBuilder – Used to build conditions
- CriteriaQuery – Holds the query
- Root – Refers to the Entity (table)

🔍 Example (Multiple Conditions):

java

CopyEdit

Predicate p1 = cb.equal(root.get("gender"), "Male");

Predicate p2 = cb.greaterThan(root.get("age"), 18);

cq.where(cb.and(p1, p2));

You can combine conditions with .and() or .or().

🔹 2. Why Use Criteria API?

Reason	Description
Type-safe	Catches errors at compile time (unlike JPQL strings)
Dynamic query building	Create flexible queries based on runtime input
Ideal for complex filters	Useful in admin dashboards, search panels, etc.

🔹 3. Comparison: JPQL vs Criteria API

Feature	JPQL	Criteria API
Syntax	SQL-like string	Java-based object query
Dynamic support	Hard to modify	Easy to build dynamically
Readability	More readable	Verbose / harder to read
Safety	Runtime errors (typos)	Compile-time errors (safe)

✅ Benefits of Criteria API

Benefit	Why it matters
Type safety	Reduces runtime errors
Reusable logic	Can modularize and reuse predicates
Complex queries	Allows joins, group by, subqueries, etc.
IDE support	Easy to refactor due to field references

❌ Drawbacks

Issue	Explanation
Verbose	A lot more code for a simple query
Harder to read	Compared to JPQL or SQL
Slower to write	Takes more effort and boilerplate

🧠 Real-life Use Case

- Building an **admin search filter** where the user can search:
    - By name, gender, age, date
    - By status or multiple criteria combinations
      → Use Criteria API to dynamically build those queries.

🎯 Interview Questions

1. What is the Criteria API in JPA?
2. When should you use Criteria API over JPQL?
3. What are the key classes used in Criteria API?
4. How can you build multiple dynamic conditions using Criteria?
5. What are the pros and cons of using Criteria API?

✅ JPA & Hibernate – Topic 9: JPA Annotations Deep Dive

JPA uses annotations to map Java classes and fields to database tables and columns. Understanding these annotations is essential for configuring how your data is stored and retrieved.

🔹 1. Basic Entity-Level Annotations

✅ @Entity

Marks a class as a JPA Entity (a table in the database).

java

CopyEdit

@Entity

public class Student { ... }

📌 **Purpose** **Maps class to table**

💡 Note Class must have a no-arg constructor and an @Id field

✅ @Table(name = "student_table")

Specifies custom table name and constraints.

java

CopyEdit

@Entity

@Table(name = "student_table", uniqueConstraints = {@UniqueConstraint(columnNames = "email")})

public class Student { ... }

| Benefit | Customize DB table name or define constraints |

🔹 2. Primary Key Annotations

✅ @Id

Marks the field as the primary key.

java

CopyEdit

@Id

private Long id;

✅ **@GeneratedValue**

Generates primary key values automatically.

java

CopyEdit

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

Strategy	Description
AUTO	Default, chooses strategy automatically
IDENTITY	Uses DB auto-increment (like MySQL)
SEQUENCE	Uses DB sequence (mostly in PostgreSQL/Oracle)
TABLE	Uses a table to generate values

🔹 **3. Column-Level Annotations**

✅ **@Column**

Maps a field to a specific DB column with options.

java

CopyEdit

@Column(name = "full_name", nullable = false, length = 50)

private String name;

Attribute	Description
name	Custom column name
nullable	Allow NULL?
length	Max length (for strings)
unique	Set uniqueness constraint

✅ **@Transient**

Tells JPA to **ignore** a field (not persisted in DB).

java

CopyEdit

@Transient

private String tempData;

🔹 **4. Timestamps & Audit Fields**

✅ **@Temporal**

Used with Date to specify format.

java

CopyEdit

@Temporal(TemporalType.TIMESTAMP)

private Date createdDate;

✅ **@CreationTimestamp and @UpdateTimestamp (Hibernate only)**

Automatically fills created/updated dates.

java

CopyEdit

@CreationTimestamp

private LocalDateTime createdOn;

@UpdateTimestamp

private LocalDateTime updatedOn;

🔹 5. Relationship Annotations

Annotation	Description
@OneToOne	One-to-One mapping
@OneToMany	One-to-Many (e.g., Customer → Orders)
@ManyToOne	Many-to-One (e.g., Order → Customer)
@ManyToMany	Many-to-Many
@JoinColumn	Specifies the foreign key column
@JoinTable	Used for mapping join table in Many-to-Many

✅ Example: OneToMany

java

CopyEdit

@OneToMany(mappedBy = "student", cascade = CascadeType.ALL)

private List<Course> courses;

✅ Benefits of JPA Annotations

Feature	Benefit
Declarative	No XML, just use annotations
Easy mapping	Map class/field → table/column easily
Cleaner code	Code remains close to data model

❌ Drawbacks

Issue	Description
Verbosity	Need to remember many annotations
Tied to JPA	Not usable with non-JPA solutions
Performance	Misused annotations can affect performance (e.g., lazy loading)

🎯 Interview Questions

1. What is the use of @Entity and @Table annotations?
2. How does @GeneratedValue work in JPA?
3. What is the difference between @Column and @JoinColumn?
4. How do you map a OneToMany relationship using annotations?
5. What is @Transient used for?

✅ JPA & Hibernate – Topic 10: Lazy vs Eager Loading

When working with entity relationships (like @OneToMany, @ManyToOne, etc.), **JPA** provides two fetching strategies:

- LAZY (default in most relationships)
- EAGER (loads immediately)

🔹 1. What is Fetching in JPA?

Fetching decides **when related entities are loaded** from the database:

Type	Meaning
EAGER	Load immediately with the main entity
LAZY	Load only when accessed (on demand)

🔹 2. LAZY Fetching (Recommended)

java

CopyEdit

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

private List<Course> courses;

✅ Courses are **not loaded** with Student.
They're fetched **only when you access getCourses()**.

💡 Saves memory and performance when related data is not always needed.

🔹 3. EAGER Fetching

java

CopyEdit

@ManyToOne(fetch = FetchType.EAGER)

private Department department;

✅ Department is **immediately loaded** with Student.

💡 Can lead to performance issues if many relationships are EAGER.

🔍 How They Work Internally

Strategy	Action
LAZY	Uses a proxy object. Actual SQL is triggered only on access.
EAGER	Executes a JOIN query or multiple SELECTs immediately.

🔹 4. Example: Lazy vs Eager in Action

java

CopyEdit

```
Student student = studentRepository.findById(1L).get();

student.getCourses(); // If LAZY, query runs here. If EAGER, query ran earlier.
```

🔹 5. Performance Comparison

Feature	LAZY (Recommended)	EAGER
Load Time	Fast (initial load)	Slower (loads more data)
Memory Use	Low	High (loads all related objects)
Flexibility	You control when data is fetched	Always loads related data

---

✅ When to Use

Use Case	Preferred Strategy
Lists or sets (e.g. @OneToMany)	LAZY
Mandatory single relation (e.g. @ManyToOne)	EAGER if always needed

---

⚠️ Common Pitfall – LazyInitializationException

You may get this error:

less

CopyEdit

org.hibernate.LazyInitializationException: failed to lazily initialize a collection

Happens when you access LAZY-loaded data **outside a transaction** (like in the controller after session is closed).

✅ **Solutions:**

1. Use @Transactional in the service layer
2. Use DTOs with JOIN FETCH
3. Fetch required relations using custom JPQL queries

🎯 **Interview Questions**

1. What is the difference between LAZY and EAGER loading?
2. Which is the default fetch type for @OneToMany and @ManyToOne?
3. Why do we get LazyInitializationException?
4. How can we solve lazy loading issues in Spring Boot?
5. Which is better – LAZY or EAGER? Why?

✅ **Summary**

Aspect	**LAZY**	**EAGER**
Timing	On demand	Immediately
Resource usage	Light	Heavy
Use case	Large collections or optional fields	Required relationships

✅ JPA & Hibernate – Topic 11: Entity Relationships

In real applications, entities are connected. JPA provides annotations to define **relationships between entities** like:

- OneToOne
- OneToMany
- ManyToOne
- ManyToMany

These map **real-world relationships** into tables using **foreign keys and join tables**.

🔹 1. OneToOne Relationship

✅ Example: A Student has one Address

java

CopyEdit

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

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "address_id")
   private Address address;
}
```

java

CopyEdit

```
@Entity
public class Address {
   @Id
   private Long id;
```

```
    private String city;
}
```

🔍 **Notes**

@JoinColumn adds a **foreign key** in Student table referencing Address ID.

🔹 **2. OneToMany & ManyToOne Relationship**

✅ **Example: One Student has Many Courses**

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

    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
    private List<Course> courses;
}
java
CopyEdit
@Entity
public class Course {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;
```

}

🔍 **Notes**

mappedBy indicates the **owning side** is Course (not Student).

@JoinColumn goes on the owning side.

🔹 **3. ManyToMany Relationship**

✅ **Example: Students can enroll in many Courses & vice versa**

java

CopyEdit

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

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

java

CopyEdit

```
@Entity
public class Course {
  @Id
```

```
    private Long id;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

📘 **Note**

JPA will create an intermediate **join table** called student_course automatically.

✅ **Cascade Types**

Type	Description
ALL	Applies all cascades
PERSIST	Saves child automatically
REMOVE	Deletes child when parent is deleted
MERGE	Propagates merge
DETACH	Detaches child when parent is detached
REFRESH	Refreshes child entity too

🔁 **Relationship Ownership**

Rule

Always use @JoinColumn on the **owning side**.

Use mappedBy on the **inverse side**.

✅ **Real-Life Mapping Example**

Relationship	Real World Example
OneToOne	User ↔ Profile

Relationship	Real World Example
OneToMany / ManyToOne	Customer ↔ Orders
ManyToMany	Students ↔ Courses

🧠 Best Practices

- Use LAZY fetching for collections.
- Avoid CascadeType.ALL in @ManyToMany (can cause issues).
- DTOs are recommended for transferring related data.

❌ Common Mistakes

Mistake	Solution
Missing mappedBy	Always define owning side properly
Using EAGER for lists	Can cause performance issues
Infinite loop in JSON (Spring Boot)	Use @JsonManagedReference, @JsonBackReference

---

🎯 Interview Questions

1. What is the difference between OneToOne and OneToMany?
2. How is ManyToMany handled in JPA?
3. What is the purpose of mappedBy?
4. What happens if both sides have @JoinColumn?
5. What are cascade types in JPA and how are they used?

✅ JPA & Hibernate – Topic 12: Spring Data JPA Repositories

Spring Data JPA is a part of Spring that **simplifies database operations**. Instead of writing boilerplate DAO code, it gives us powerful interfaces for **CRUD and query operations** on entities.

🔹 1. What is Spring Data JPA?

Spring Data JPA is a **wrapper over JPA** that provides:

- Pre-defined repository interfaces (like JpaRepository)
- Powerful query generation using method names
- Pagination and sorting
- Support for custom queries using JPQL/native SQL

🔹 2. Main Interfaces

Interface	Extends	Description
CrudRepository	-	Basic CRUD (save, findById, delete)
PagingAndSortingRepository	CrudRepository	Adds pagination & sorting
JpaRepository	PagingAndSortingRepository	Full features: CRUD, pagination, custom queries

✅ Most commonly used: JpaRepository

🔹 3. Creating a Repository

Assume we have an Entity called Student.

java

CopyEdit

```
@Entity
public class Student {
  @Id
```

```
    private Long id;
    private String name;
    private int age;
}
```

Now create a repository:

java

CopyEdit

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    // custom queries here
}
```

🔹 4. Common Methods Available

Method	Description
findAll()	Get all records
findById(id)	Find record by ID
save(entity)	Insert/update entity
delete(entity)	Delete entity
existsById(id)	Check if entity exists
count()	Total count
findAll(Sort sort)	Sort results
findAll(Pageable page)	Paginate results

🔹 5. Custom Query Methods (Derived Queries)

You can write queries **just by method names**:

java

CopyEdit

List<Student> findByName(String name);

List<Student> findByAgeGreaterThan(int age);

List<Student> findByNameAndAge(String name, int age);

Spring Data parses the method and builds the query behind the scenes.

🔹 6. Custom JPQL or Native Queries

You can also define queries manually:

java

CopyEdit

@Query("SELECT s FROM Student s WHERE s.name = ?1")

List<Student> getStudentsByName(String name);

@Query(value = "SELECT * FROM student WHERE age > ?1", nativeQuery = true)

List<Student> getStudentsAboveAge(int age);

🔹 7. Pagination and Sorting

java

CopyEdit

Page<Student> findAll(Pageable pageable);

java

CopyEdit

Pageable pageable = PageRequest.of(0, 10, Sort.by("name"));

Page<Student> page = studentRepository.findAll(pageable);

✅ Real Life Use Case

Let's say you're building a Student Management System.

- StudentRepository handles all DB logic.
- You don't need to write a single SQL or HQL to fetch records.

- Easily implement search, pagination, and filter options.

🎯 Interview Questions

1. What is Spring Data JPA?
2. What is the difference between CrudRepository and JpaRepository?
3. How does Spring Data JPA generate queries from method names?
4. How do you write custom queries in Spring Data JPA?
5. How to perform pagination in Spring Data JPA?

✅ Benefits

- Saves 70–80% boilerplate code
- Easy to use and integrate
- Provides pagination and sorting out-of-the-box
- Supports derived, JPQL, and native queries

⚠️ Drawbacks / Cautions

- For very complex queries, native SQL might still be better
- Derived query method names can get long and hard to read
- Overuse can lead to large repository interfaces

✅ JPA & Hibernate – Topic 13: JPQL vs Native SQL in Spring Data JPA

JPA provides two ways to write custom queries:

1. **JPQL** – Java Persistence Query Language (object-oriented)
2. **Native SQL** – Traditional SQL (table-oriented)

Understanding the difference helps you choose the right one based on your needs.

🔹 1. What is JPQL?

- JPQL stands for **Java Persistence Query Language**.
- It works with **Java entities and fields**, not database tables/columns.
- It's **database independent**.
- Syntax is similar to SQL but uses **class and field names**.

✅ Example

java

CopyEdit

@Query("SELECT s FROM Student s WHERE s.name = ?1")

List<Student> getStudentsByName(String name);

Here:

- Student is the Entity class.
- s.name is a field in the entity, **not a table column**.

🔹 2. What is Native SQL?

- Native SQL is **actual SQL**, executed directly on the database.
- It uses **table names and column names**.
- Not portable (DB-specific features like LIMIT, ROWNUM, etc.).
- Sometimes necessary for complex queries or performance tuning.

✅ Example

java

CopyEdit

```
@Query(value = "SELECT * FROM student WHERE name = ?1", nativeQuery = true)
List<Student> getStudentsByName(String name);
```

Here:

- student is the actual table.
- name is a table column.

🔄 JPQL vs Native SQL: Side-by-Side Comparison

Feature	JPQL	Native SQL
Based On	Java Entities	Database Tables
Portability	Yes (DB-independent)	No (DB-specific)
Syntax	Java-style	SQL-style
Use Cases	Simple/medium queries	Complex queries, joins, DB-specific tasks
Return Type	Entity objects	Entities or DTOs with @SqlResultSetMapping
Performance	Slower for complex queries	Better optimization options
Named Parameters	Supported	Supported

🔹 3. When to Use Which?

Scenario	Recommendation
Simple select with filters	Use JPQL
Join fetch with mapped entities	Use JPQL
Complex joins not mapped in entities	Use Native SQL
Use of DB-specific functions (e.g. LIMIT)	Use Native SQL

Scenario	Recommendation
Full control over raw query	Use Native SQL

✅ **Real-Life Example**

Let's say you want to fetch top 3 students by marks.

- **JPQL:**

java

CopyEdit

@Query("SELECT s FROM Student s ORDER BY s.marks DESC")

List<Student> findTopStudents(Pageable pageable);

Call with:

java

CopyEdit

PageRequest.of(0, 3)

- **Native SQL:**

java

CopyEdit

@Query(value = "SELECT * FROM student ORDER BY marks DESC LIMIT 3", nativeQuery = true)

List<Student> findTop3Students();

❗ **Drawbacks**

JPQL	Native SQL
Cannot use database-specific features	Not portable, breaks on DB change
Difficult for very complex joins	Hard to map results to objects sometimes

---

🎯 **Interview Questions**

1. What is JPQL?
2. What is the difference between JPQL and SQL?
3. When would you use native query instead of JPQL?
4. Can JPQL support joins?
5. What are the limitations of JPQL?

✅ **Summary**

- **Use JPQL** when possible for clean, maintainable, and portable code.
- **Use Native SQL** when you need raw power and full control.

✅ JPA & Hibernate – Topic 14: Named Queries & @NamedQuery Annotation

---

🔹 1. What is a Named Query?

A **Named Query** is a **static, pre-defined JPQL query** associated with an entity.
It's defined using the @NamedQuery annotation and allows **reuse of JPQL queries** without rewriting them.

🔹 2. Why Use Named Queries?

- Reusability: Define once, use anywhere.
- Maintainability: Easy to update queries in one place.
- Performance: Parsed at startup (can be slightly faster).
- Clarity: Avoids dynamic query creation in code.

🔹 3. How to Define Named Queries

✅ Syntax

You define it **above the entity class**:

java

CopyEdit

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

You can also define **multiple queries** using @NamedQueries:

java

CopyEdit

```
@NamedQueries({
  @NamedQuery(name = "Student.findByName", query = "SELECT s FROM Student s WHERE s.name = :name"),
  @NamedQuery(name = "Student.findByAge", query = "SELECT s FROM Student s WHERE s.age = :age")
})
```

🔹 4. How to Use Named Queries in Repository

java

CopyEdit

```
@Query(name = "Student.findByName")
List<Student> findByName(@Param("name") String name);
```

OR with EntityManager:

java

CopyEdit

```
List<Student> students = entityManager
  .createNamedQuery("Student.findByName", Student.class)
  .setParameter("name", "Viraj")
  .getResultList();
```

🔹 5. Real-Life Example

Let's say you need a query to get students by name in many places in the app.
Instead of repeating @Query(...), just define a **named query once** in the Student entity.

Then reuse it wherever needed.

🔁 Named Queries vs Dynamic Queries

Feature	Named Query	Dynamic Query (@Query)
Defined in	Entity class	Repository interface
Dynamic	No	Yes
Reusable	Yes	Not easily
Readability	High	Medium
Flexibility	Low	High

🎯 Interview Questions

1. What is a Named Query in JPA?
2. How is a Named Query different from @Query?
3. Where are Named Queries defined?
4. What are the advantages of using Named Queries?
5. Can you use Named Queries with native SQL?

✅ Benefits

- Easy to reuse
- Helps organize queries centrally
- Precompiled by JPA provider (slight performance benefit)
- Reduces duplication

⚠️ Drawbacks

- Not flexible — can't modify query at runtime
- Difficult to manage if many queries are in one entity
- Harder to read if entity becomes too cluttered

✅ JPA & Hibernate – Topic 15: Relationships in JPA (OneToOne, OneToMany, ManyToOne, ManyToMany)

🔹 1. What are JPA Relationships?

In real-world applications, entities (like Student, Course, Department) are **connected**.

JPA allows us to **map those relationships** using annotations like:

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

This makes it easier to model and navigate relational data in an object-oriented way.

🔹 2. Types of Relationships

🔸 a. @OneToOne

✅ **Meaning**: One entity has exactly one related entity.

📌 **Example**: A student has one address.

java

CopyEdit

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

   @OneToOne
   private Address address;
}
```

🔸 **b. @OneToMany**

✅ **Meaning**: One entity has multiple related entities.

📌 **Example**: One department has many students.

java

CopyEdit

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

   @OneToMany
   private List<Student> students;
}
```

🔸 **c. @ManyToOne**

✅ **Meaning**: Many entities share one related entity.

📌 **Example**: Many students belong to one department.

java

CopyEdit

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

   @ManyToOne
   private Department department;
}
```

🔸 **d. @ManyToMany**

✅ **Meaning**: Many entities relate to many others.

📌 **Example**: Students can enroll in many courses, and each course can have many students.

java

CopyEdit

@Entity

public class Student {

  @Id

  private Long id;

  @ManyToMany

  private List<Course> courses;

}

🔹 **3. Owning vs Inverse Side**

- The **owning side** is the entity that manages the relationship.
- You define the relationship on both sides using mappedBy on the inverse side.

✅ Example:

java

CopyEdit

@Entity

public class Student {

  @Id

  private Long id;

  @ManyToOne

```
    private Department department;
}

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

    @OneToMany(mappedBy = "department")
    private List<Student> students;
}
```

🔹 4. Cascade Types

JPA allows operations to be **cascaded** from parent to child:

CascadeType	Description
PERSIST	Saves child when parent is saved
REMOVE	Deletes child when parent is deleted
MERGE	Updates child when parent is updated
ALL	All of the above

```
java
CopyEdit
@OneToMany(cascade = CascadeType.ALL)
private List<Student> students;
```

🔹 5. Fetch Types

Type	Description
LAZY	Loads related entity only when needed
EAGER	Loads related entity immediately

java

CopyEdit

@OneToMany(fetch = FetchType.LAZY)

✅ Real-Life Example

In a Student-Course System:

- Each student belongs to one department → @ManyToOne
- Each department has many students → @OneToMany
- A student has one ID card → @OneToOne
- A student can take many courses, and a course has many students → @ManyToMany

---

🎯 Interview Questions

1. What are the different types of relationships in JPA?
2. What is the difference between @OneToMany and @ManyToOne?
3. What is the owning side in a relationship?
4. What does cascade do in JPA?
5. Difference between LAZY and EAGER fetching?

✅ Benefits

- Helps model real-world relationships clearly
- Automatically handles joins in the background
- Saves time and code for navigating related data

⚠️ Drawbacks

- Misusing EAGER fetching can lead to performance issues (N+1 problem)
- Bidirectional relationships need careful design to avoid infinite loops
- Managing cascade operations carelessly may cause unwanted deletes/updates

✅ JPA & Hibernate – Topic 16: EntityManager and Persistence Context

---

🔹 1. What is EntityManager in JPA?

EntityManager is the **central interface** in JPA to interact with the **persistence context** and the database.

You can think of it as the **bridge between your Java code and the database**.

It is responsible for:

- CRUD operations (Create, Read, Update, Delete)
- Query execution
- Managing transactions
- Managing entities' lifecycle

✅ Basic Example:

java

CopyEdit

@Autowired

private EntityManager entityManager;

Student student = entityManager.find(Student.class, 1L);

entityManager.persist(new Student(...));

entityManager.remove(student);

🔹 2. What is a Persistence Context?

The **Persistence Context** is the **first-level cache** in JPA.

It keeps track of all **managed entities** (Java objects) in memory during a transaction. This allows JPA to:

- Avoid unnecessary DB calls
- Automatically detect changes and sync them to DB

✅ **Example:**

java

CopyEdit

Student s1 = entityManager.find(Student.class, 1L); // DB hit

Student s2 = entityManager.find(Student.class, 1L); // No DB hit, comes from cache

Both s1 and s2 are the same object (from persistence context).

🔹 **3. Common EntityManager Methods**

Method	Description
persist(entity)	Adds a new entity to the database
find(Class, id)	Finds an entity by ID
remove(entity)	Deletes an entity
merge(entity)	Updates an existing entity
createQuery(...)	Runs JPQL queries
createNativeQuery(...)	Runs raw SQL queries
flush()	Forces synchronization with the database
clear()	Detaches all managed entities (clears cache)

🔹 **4. Lifecycle of an Entity (with Persistence Context)**

State	Description
New	Just created, not linked to DB
Managed	Attached to persistence context
Detached	Removed from context, not tracked anymore
Removed	Scheduled for deletion

🔹 5. How EntityManager Helps

✅ Auto Update

If an object is managed (in persistence context), any change is auto-saved on commit.

java

CopyEdit

```
Student s = entityManager.find(Student.class, 1L);

s.setName("Updated Name");

// No need to call update – it's automatic on commit
```

✅ Query Example

java

CopyEdit

```
List<Student> students = entityManager

  .createQuery("SELECT s FROM Student s WHERE s.name = :name", Student.class)

  .setParameter("name", "Viraj")

  .getResultList();
```

🔹 6. Real-Life Example

Imagine you're building a student management system.
Instead of writing raw SQL, you use EntityManager to:

- Fetch student details
- Register new students
- Automatically track and update changes
- Improve performance by caching in the same transaction

🎯 Interview Questions

1. What is the purpose of EntityManager?
2. What is a persistence context in JPA?

3. How does JPA avoid multiple DB hits for the same object?
4. What is the difference between persist and merge?
5. What is the lifecycle of a JPA entity?

✅ **Benefits**

- Clean and object-oriented way to handle database operations
- First-level cache boosts performance
- Automatically tracks entity changes
- Reduces boilerplate SQL code

⚠️ **Drawbacks**

- Can consume more memory if too many entities are managed
- Needs careful handling in large or long-running transactions
- Improper use may cause stale data or exceptions

✅ JPA & Hibernate – Topic 17: Caching in JPA (First-Level & Second-Level Cache)

🔹 1. What is Caching in JPA?

Caching helps improve performance by reducing the number of times JPA needs to hit the **database**.
Instead of querying the database repeatedly, data is stored temporarily in **memory**.

JPA offers two levels of caching:

Cache Level	Scope	Managed By
First-Level Cache	Per EntityManager	JPA (Always ON)
Second-Level Cache	Across sessions/managers	Optional (Hibernate or other providers)

🔹 2. First-Level Cache (L1 Cache)

✅ **Default cache in JPA**
✅ Automatically enabled
✅ Exists **per EntityManager**

✅ **Example:**

java

CopyEdit

Student s1 = entityManager.find(Student.class, 1L); // DB Hit

Student s2 = entityManager.find(Student.class, 1L); // No DB Hit – comes from cache

🧠 Both s1 and s2 refer to the **same object**, fetched only **once** from DB.

🔹 How It Works:

- When you call find() or getReference(), the result is cached.
- If you call the same query again **within the same transaction**, JPA returns the cached object.
- Cache is **cleared when EntityManager is closed or cleared** manually.

🔹 **3. Second-Level Cache (L2 Cache)**

✅ **Optional cache**
✅ Works **across EntityManager sessions**
✅ Managed by **Hibernate**, not JPA itself
✅ Needs manual configuration

✅ **When to Use:**

- You want to **share cached entities across multiple sessions**.
- Your app is **read-heavy**, and hitting DB repeatedly is a bottleneck.

🔸 **Technologies Used for L2 Cache:**

- **EhCache**
- **Hazelcast**
- **Infinispan**
- **Redis**

✅ **Configuration (Hibernate + EhCache Example):**

properties

CopyEdit

spring.jpa.properties.hibernate.cache.use_second_level_cache=true

spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory

spring.cache.jcache.config=classpath:ehcache.xml

🔹 **4. Cache Clearing**

✅ **Manually Clear L1 Cache:**

java

CopyEdit

entityManager.clear();

✅ **Manually Evict L2 Cache:**

java

CopyEdit

Cache cache = entityManagerFactory.getCache();

cache.evict(Student.class);

🔹 **5. Real-Life Example**

Let’s say you're building a course registration system with thousands of students.

- Using L1 cache: A request that fetches the same student twice in the same transaction will only query the DB once.
- Using L2 cache: The same student info fetched by another session won’t need to hit the DB if cached.

🎯 **Interview Questions**

1. What is the difference between First-Level and Second-Level cache in JPA?
2. Is caching enabled by default in JPA?
3. How do you configure Second-Level cache in Hibernate?
4. What are the benefits and risks of caching?
5. How do you clear the EntityManager cache?

✅ **Benefits**

- Reduces number of DB calls
- Improves application performance
- Provides faster data access

⚠️ **Drawbacks**

- L2 cache must be carefully managed to avoid **stale data**

- Inconsistent cache across clusters if not handled properly
- Overuse can lead to **memory issues**