Project interview questions

Great! Here's a **complete set of answers** to all the **interview questions** mentioned above, organized section-wise based on your CategoryMaster entity:

**🔹 1. What is an entity in JPA?**

An entity in JPA is a lightweight, persistent domain object that represents a table in a relational database. Each instance of the entity class corresponds to a row in the table. Entities are annotated with @Entity.

**🔹 2. How does Hibernate know which class to persist?**

Hibernate uses the @Entity annotation to identify persistent classes. You must also configure the class in persistence.xml or Spring Boot will automatically scan based on your package structure.

**🔹 3. Can a JPA entity exist without a primary key?**

❌ No.  
Every JPA entity **must have a primary key**. It is used by the persistence context to track and manage entity instances.

**🔹 4. What are the different @GeneratedValue strategies in JPA?**

| **Strategy** | **Description** |
| --- | --- |
| AUTO | Default; lets JPA decide based on DB |
| IDENTITY | Uses DB auto-increment |
| SEQUENCE | Uses DB sequence (PostgreSQL/Oracle) |
| TABLE | Uses a special table to generate IDs |

**🔹 5. When to use IDENTITY vs SEQUENCE?**

* Use **IDENTITY** if you're using MySQL or H2 and want the DB to handle auto-increment.
* Use **SEQUENCE** with PostgreSQL or Oracle where sequences are supported.
* SEQUENCE is often more flexible and performs better in batch inserts.

**🔹 6. What is the use of the @Column annotation?**

@Column maps a class field to a table column and allows customization such as name, length, nullable, etc.

Example:

@Column(name = "category\_name", length = 100, nullable = false)

**🔹 7. What happens if you don’t use @Column?**

JPA will use the **field name as the column name** by default, and apply default settings like nullable=true and no length restriction for Strings.

**🔹 8. How do you enforce non-null constraints in JPA?**

Use @Column(nullable = false) to ensure that the field is required. It reflects in the DB as a NOT NULL constraint.

**🔹 9. How would you define a foreign key?**

You can define foreign keys using @ManyToOne and @JoinColumn annotations:

@ManyToOne

@JoinColumn(name = "subcategoryid")

private SubCategoryMaster subCategory;

**🔹 10. How can you control string size in JPA?**

Use the length attribute in @Column(length = 100) to define max character limit for a VARCHAR.

**🔹 11. How is data validation handled in JPA entities?**

You can use:

* @Column(nullable = false)
* Bean Validation API like @NotNull, @Size, etc.  
  Example:

@NotNull

@Size(min = 2, max = 100)

@Column(length = 100)

private String categoryName;

**🔹 12. Should images be stored in DB or filesystem?**

✅ **Recommended: Store image path in DB and image file in filesystem or cloud (S3, etc.)**  
❌ Storing actual image (as BLOB) makes DB heavy, backups slow, and queries inefficient.

**🔹 13. Why use path instead of storing image in BLOB?**

Storing **file path** reduces DB size, improves performance, and makes image serving easy via static file servers or CDN.

**🔹 14. What is soft delete and how is it implemented?**

**Soft delete** means marking data as inactive instead of physically deleting it from DB.  
✅ Use a flag like isDeleted or isActive (Boolean):

@Column(nullable = false)

private Boolean isActive;

Then query like:

SELECT \* FROM CategoryMaster WHERE isActive = true;

**🔹 15. Why use a Boolean flag instead of deleting rows?**

Advantages of Boolean Flag (Soft Delete):

* Keeps historical data
* Avoids foreign key constraint violations
* Easy to recover accidentally deleted data
* Supports audit trails

**🔹 16. Can we use composite keys in JPA?**

✅ Yes, by using:

* @EmbeddedId
* @IdClass

But it's more complex and usually avoided unless necessary.

Let me know if you want:

* Mock interview questions based on this class
* Real-world use case diagram (ERD)
* CRUD repository or service layer for this entity

**📦 1. import com.fasterxml.jackson.annotation.JsonIgnore;**

**🔹 Purpose:**

This is part of the **Jackson** library used for **JSON serialization/deserialization** in Java (especially in Spring Boot REST APIs).

**✅ What does @JsonIgnore do?**

It tells Jackson **not to include this field** when converting Java objects to JSON (i.e., during response output).

**🧠 Example:**

public class User {

private String username;

@JsonIgnore

private String password; // 👈 This will NOT appear in JSON response

}

📤 JSON Output:

{

"username": "vikas"

}

Converting Java objects to JSON is a common task in web applications, especially when building REST APIs using Spring Boot. The most popular library for this is **Jackson**, which is usually **auto-configured** in Spring Boot.

**✅ 1. What is JSON Serialization?**

**Serialization** = Converting Java object → JSON  
**Deserialization** = Converting JSON → Java object

Example:

**Java Object:**

public class User {

private String name;

private int age;

}

**JSON:**

{

"name": "Vikas",

"age": 25

}

**🛠️ 2. Using Jackson to Convert Java Object to JSON**

**👉 Using ObjectMapper**

import com.fasterxml.jackson.databind.ObjectMapper;

public class JsonExample {

public static void main(String[] args) throws Exception {

ObjectMapper mapper = new ObjectMapper();

User user = new User("Vikas", 25);

// Convert Java object to JSON string

String json = mapper.writeValueAsString(user);

System.out.println(json);

}

}

**Output:**

{"name":"Vikas","age":25}

**🔁 3. Convert JSON back to Java (Deserialization)**

String json = "{\"name\":\"Vikas\",\"age\":25}";

User userObj = mapper.readValue(json, User.class);

System.out.println(userObj.getName()); // Vikas

**🧩 4. Spring Boot Example (Automatic)**

In Spring Boot, if you return an object from a REST controller, it is automatically converted to JSON using Jackson.

@RestController

public class UserController {

@GetMapping("/user")

public User getUser() {

return new User("Vikas", 25); // 🔁 Auto converted to JSON

}

}

**Response:**

{

"name": "Vikas",

"age": 25

}

**📦 5. Controlling JSON Output with Annotations**

| **Annotation** | **Purpose** |
| --- | --- |
| @JsonIgnore | Ignore field from JSON |
| @JsonProperty("custom\_name") | Rename field in JSON |
| @JsonInclude(Include.NON\_NULL) | Exclude null fields |
| @JsonFormat | Format date/time values |

**Example:**

public class User {

private String name;

@JsonIgnore

private String password;

@JsonProperty("user\_age")

private int age;

}

**🧠 Interview Questions on Java to JSON Conversion**

| **Question** | **Answer** |
| --- | --- |
| Which library is used for Java-JSON conversion? | Jackson (com.fasterxml.jackson) |
| What is ObjectMapper? | A Jackson class used to convert Java objects to/from JSON |
| How do you ignore a field in JSON output? | Use @JsonIgnore |
| How does Spring Boot convert Java to JSON automatically? | Spring uses Jackson’s HttpMessageConverters under the hood |
| What if you want pretty-printed JSON? | Use mapper.writerWithDefaultPrettyPrinter().writeValueAsString(obj) |

You're working with JPA's **entity relationship mapping**, and this line:

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "catMasterId", nullable = false)

@JsonIgnore

private CategoryMaster categoryMaster;

contains **three core concepts** used in enterprise Java:

* JPA **entity relationship mapping** (@ManyToOne, @JoinColumn)
* JSON **serialization control** (@JsonIgnore)
* **Lazy fetching** optimization (FetchType.LAZY)

Let’s **explain each part in detail**, along with **background theory**, **real-world meaning**, and **interview Q&A** 👇

**🔹 @ManyToOne(fetch = FetchType.LAZY)**

**✅ Meaning:**

Defines a **Many-to-One relationship** in JPA between the current entity and the CategoryMaster entity.

**🔍 Background:**

* This means **"Many subcategories (or records in this table) belong to one CategoryMaster."**
* Used in database relationships like **foreign keys**.

🧠 Example:  
If you have 5 cities like Mumbai, Pune, Nagpur, etc., and they all belong to 1 state (Maharashtra), then:

* Cities = Many
* State = One  
  ➡️ ManyToOne relationship.

**🔧 FetchType.LAZY:**

* **Lazy loading** means: the categoryMaster object will **not be fetched immediately** when this object is retrieved.
* It will be fetched **only when you access it** (on-demand).

➡️ Helps improve performance and avoid unnecessary joins.

**🧠 Interview Questions:**

| **Question** | **Answer** |
| --- | --- |
| What is @ManyToOne in JPA? | It defines a relationship where many entities refer to one parent. |
| What is lazy vs eager loading? | Lazy loads related data only when accessed. Eager loads immediately. |
| Which is default fetch type for @ManyToOne? | FetchType.EAGER is default, but can be overridden to LAZY. |
| Why is lazy loading preferred? | It improves performance by avoiding unnecessary DB joins. |

**🔹 @JoinColumn(name = "catMasterId", nullable = false)**

**✅ Meaning:**

Specifies the **foreign key column** in the current table that **links to CategoryMaster**.

**🔍 Background:**

* catMasterId is the foreign key in this table that references the categoryid (usually primary key) of CategoryMaster.
* nullable = false means: this relationship **must exist**. i.e., no orphaned child without parent category.

🧠 Behind the scenes:

ALTER TABLE SubCategory

ADD CONSTRAINT fk\_cat FOREIGN KEY (catMasterId)

REFERENCES CategoryMaster(categoryid);

**🧠 Interview Questions:**

| **Question** | **Answer** |
| --- | --- |
| What is @JoinColumn? | Maps a foreign key column for entity relationships. |
| What does nullable = false do? | Prevents null foreign keys; ensures integrity. |
| Can the foreign key name be different from the field name? | Yes, using @JoinColumn(name="..."). |

**🔹 @JsonIgnore**

**✅ Meaning:**

Tells **Jackson (used for JSON)** to **ignore this field** when serializing to JSON.

**🔍 Background:**

* Useful to **prevent infinite recursion** in bidirectional relationships (like parent → child → parent…).
* Also hides sensitive or unnecessary data from API response.

🧠 Why needed here:

* If CategoryMaster has a List<SubCategory> and SubCategory again has a reference to CategoryMaster, it creates a loop.
* JSON conversion fails with:  
  com.fasterxml.jackson.databind.JsonMappingException: Infinite recursion

✅ Solution: Use @JsonIgnore to break the loop.

**🧠 Interview Questions:**

| **Question** | **Answer** |
| --- | --- |
| Why use @JsonIgnore in JPA entities? | To prevent circular reference issues during JSON serialization. |
| What is infinite recursion in Jackson? | When two entities reference each other and JSON keeps nesting. |
| Can you ignore only during serialization but not deserialization? | Yes, using @JsonIgnoreProperties or advanced Jackson configs. |

**📌 Example Scenario (Real-World)**

**Entities:**

**🟦 CategoryMaster**

@Entity

public class CategoryMaster {

@Id

private int id;

private String name;

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

private List<SubCategory> subCategories;

}

**🟨 SubCategory**

@Entity

public class SubCategory {

@Id

private int id;

private String name;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "catMasterId", nullable = false)

@JsonIgnore

private CategoryMaster categoryMaster;

}

Now if /api/subcategory is called:

* JSON will show SubCategory details
* But it **won’t show the CategoryMaster object** because of @JsonIgnore → avoiding infinite loop

**🧠 Bonus Interview Tips:**

| **Concept** | **Tips** |
| --- | --- |
| Bidirectional Relationships | Always control JSON output using @JsonIgnore or DTOs |
| LAZY vs EAGER | LAZY is better unless the child is always required |
| Serialization Issue | Use DTOs or @JsonManagedReference/@JsonBackReference for complex mappings |

Let’s break down and **fully explain** this line:

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

private List<SubCategory> subCategories;

This line is from the CategoryMaster entity and it defines a **one-to-many** relationship with the SubCategory entity.

**🔍 Full Explanation**

**✅ @OneToMany**

* Declares a **one-to-many** relationship in JPA.
* Means: **One CategoryMaster** can have **many SubCategories**.
* It's the **parent** side of the relationship.

**✅ mappedBy = "categoryMaster"**

* mappedBy tells JPA that this side **is not the owner** of the relationship.
* The owner is the SubCategory class (child entity), where the foreign key actually resides.

**In SubCategory:**

@ManyToOne

@JoinColumn(name = "catMasterId")

private CategoryMaster categoryMaster;

This means:

* categoryMaster in SubCategory owns the foreign key.
* So in CategoryMaster, JPA knows it just needs to **map** and not **create a new column**.

**✅ cascade = CascadeType.ALL**

* All operations (persist, merge, remove, refresh) will **cascade** from CategoryMaster to SubCategory.

👉 Example:  
If you save or delete a CategoryMaster, its subCategories are also:

* Saved
* Deleted
* Updated automatically

🔁 Types of CascadeType:

| **Type** | **Description** |
| --- | --- |
| PERSIST | Save child when parent is saved |
| MERGE | Update child when parent is updated |
| REMOVE | Delete child when parent is deleted |
| ALL | All of the above |

**✅ List<SubCategory> subCategories**

* It stores all related subcategories of a particular category.

For example:

CategoryMaster cm = new CategoryMaster();

cm.setCategoryName("Travel");

SubCategory s1 = new SubCategory("Flights", cm);

SubCategory s2 = new SubCategory("Hotels", cm);

cm.setSubCategories(Arrays.asList(s1, s2));

**🧠 Behind the scenes — Table Relationship**

It means:

* In DB, there is a **foreign key column in SubCategory table** (catMasterId)
* But **no foreign key in CategoryMaster** table

**💬 Common Interview Questions & Answers**

| **Question** | **Answer** |
| --- | --- |
| What is @OneToMany in JPA? | It maps a parent entity to a list of related child entities. |
| What does mappedBy mean? | It shows which entity owns the relationship (the foreign key). |
| Why use CascadeType.ALL? | To apply parent operations to children automatically. |
| Will a column be created for subCategories in the database? | No. The relationship is stored in the SubCategory table (via catMasterId). |
| What happens if you delete a CategoryMaster? | All related SubCategory entries will be deleted (due to CascadeType.REMOVE). |

**🔄 Bidirectional Mapping Summary**

**In CategoryMaster (parent):**

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

private List<SubCategory> subCategories;

**In SubCategory (child):**

@ManyToOne

@JoinColumn(name = "catMasterId", nullable = false)

private CategoryMaster categoryMaster;

**🧠 When to Use It?**

* When parent-child structure exists (like Category → SubCategory)
* When you want automatic cascading of data (save/delete/update children with parent)
* When you need to access all children from parent side (ex: showing category with subcategories in API)

Let's break this line down in **complete detail**, including real-world meaning, JPA behavior, background theory, and common **interview questions**.

**🔍 Code:**

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

private List<Passenger> passengers;

**✅ Full Meaning & Detailed Explanation:**

**1. @OneToMany**

This annotation defines a **One-to-Many** relationship:

* **One Customer** → can have **many Passenger** entries.
* This is used in **ORM** (Object-Relational Mapping) to map Java relationships to SQL relationships.

**2. mappedBy = "customer"**

This tells JPA:

* **This side is inverse (non-owning side)**.
* The **foreign key is in the Passenger entity**, inside the field named customer.

In Passenger entity, it looks like:

@ManyToOne

@JoinColumn(name = "customer\_id", nullable = false)

private Customer customer;

**Why mappedBy is important:**

* Prevents **extra join table** from being created.
* Tells Hibernate: **don’t create a new column**, use the one in the child entity.

**3. cascade = CascadeType.ALL**

All changes in Customer are **cascaded** to Passenger.

| **Parent (Customer) Action** | **What happens to Passenger** |
| --- | --- |
| persist() | Save all passengers too |
| merge() | Update all passengers |
| remove() | Delete all passengers |
| detach() | Detach all passengers |
| refresh() | Refresh all passengers |

👉 **Example:**

Customer c = new Customer();

Passenger p1 = new Passenger("John", c);

Passenger p2 = new Passenger("Doe", c);

c.setPassengers(Arrays.asList(p1, p2));

entityManager.persist(c); // Passengers will be saved automatically

**4. orphanRemoval = true**

**Key Point**: This is often missed in interviews and is very important.

**🔴 Meaning:**

If a Passenger is removed from the list (passengers.remove(p1)), it will be:

* **Automatically deleted from the database** as well.
* Considered an "orphan" because it no longer belongs to the parent.

👉 Without orphanRemoval = true, it would just **break the relationship**, but **not delete** the record from the DB.

**5. private List<Passenger> passengers;**

This is a collection of all passengers related to the customer.

**6. Getter:**

public List<Passenger> getPassengers() {

return passengers;

}

This is the standard Java getter to access the passenger list.

**🧠 Table Design Behind the Scene:**

* **Customer Table** (parent):
  + customer\_id (PK)
  + name, email, etc.
* **Passenger Table** (child):
  + passenger\_id (PK)
  + customer\_id (FK referencing customer table)

**🔍 Sample JSON (Bi-directional Mapping May Cause Infinite Loop)**

* Avoid this with @JsonIgnore or DTOs.

**Customer:**

{

"id": 1,

"name": "Vikas",

"passengers": [

{ "id": 101, "name": "John" },

{ "id": 102, "name": "Doe" }

]

}

**💬 Common Interview Questions**

| **Question** | **Answer** |
| --- | --- |
| What does @OneToMany mean in JPA? | It maps a parent to a list of child entities. |
| What is mappedBy used for? | Specifies the field in the child that owns the relationship. |
| What is the effect of CascadeType.ALL? | Operations on parent are cascaded to children. |
| What is orphanRemoval = true? | Automatically deletes children removed from collection. |
| Will this create a new table in DB? | No, it uses the foreign key in the Passenger table. |
| What happens if I remove a passenger from the list? | If orphanRemoval=true, it's deleted from DB. |
| Is this unidirectional or bidirectional? | Bidirectional, because both entities refer to each other. |

**✅ Summary**

| **Annotation** | **Purpose** |
| --- | --- |
| @OneToMany | Relationship from Customer → Passenger |
| mappedBy | Tells which field owns the relation (in Passenger) |
| cascade = CascadeType.ALL | Automatically persist, delete, update children |
| orphanRemoval = true | Deletes child if it's removed from the list |
| Getter | Standard POJO method |

Let's fully **break down** and **explain** this code:

@Column(name = "bookingDate", nullable = false, updatable = false)

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'")

private LocalDateTime bookingDate;

@Enumerated(EnumType.STRING)

@Column(name = "bookingStatus", nullable = false)

private BookingStatus bookingStatus = BookingStatus.PENDING;

**✅ Section-by-Section Detailed Explanation**

**🔷 1. @Column(name = "bookingDate", nullable = false, updatable = false)**

**🎯 Purpose:**

* Maps bookingDate field to the bookingDate column in the DB.

| **Attribute** | **Meaning** |
| --- | --- |
| name | Column name in DB: bookingDate |
| nullable=false | Cannot be NULL in DB |
| updatable=false | Once inserted, value **cannot be updated** via JPA |

**🧠 When to use updatable=false?**

* For timestamps that should not be changed later.
* Example: bookingDate, createdAt, etc.

**📌 Interview Tip:**

Q: What happens if you try to change a column marked as updatable = false?  
A: JPA will **ignore the update** for that field. DB value remains unchanged.

**🔷 2. @JsonFormat(...)**

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'")

**🎯 Purpose:**

* **Used by Jackson** (JSON parser in Spring Boot) to format the date in a readable way when returning JSON in REST APIs.

**✅ Format:**

* yyyy-MM-dd'T'HH:mm:ss.SSS'Z' → ISO 8601 format  
  Example:  
  2025-07-28T11:30:45.000Z

**🧠 Interview Tip:**

Q: How to ensure LocalDateTime serializes in correct format in Spring Boot JSON response?  
A: Use @JsonFormat or configure ObjectMapper globally.

**🔷 3. private LocalDateTime bookingDate;**

* Java 8 LocalDateTime represents date + time **without timezone**.
* More modern and safer than Date class.

**🔷 4. @Enumerated(EnumType.STRING)**

**🎯 Purpose:**

* Tells JPA to store **Enum as String** in DB instead of its ordinal (index).

public enum BookingStatus {

PENDING,

CONFIRMED,

CANCELED

}

This stores values like 'PENDING', 'CONFIRMED', not 0, 1, 2.

**🧠 Background:**

* Safer to store as **String**:
  + Prevents issues if order of enum constants change.
  + Readable in DB.

Default = EnumType.ORDINAL, which stores as 0, 1, 2, which is risky.

**📌 Interview Tip:**

Q: Why prefer EnumType.STRING over ORDINAL?  
A: Because it is **readable** and **order-independent**.

**🔷 5. @Column(name = "bookingStatus", nullable = false)**

* Maps bookingStatus field to bookingStatus column.
* Cannot be null in DB.

**🔷 6. private BookingStatus bookingStatus = BookingStatus.PENDING;**

* Enum field with default value.
* When object is created, bookingStatus will automatically be set to PENDING if not assigned.

**💬 Real-World Usage**

{

"bookingDate": "2025-07-28T15:33:45.000Z",

"bookingStatus": "PENDING"

}

In DB:

| **bookingDate** | **bookingStatus** |
| --- | --- |
| 2025-07-28 15:33:45.000 | PENDING |

**🧠 Common Interview Questions & Answers**

| **Question** | **Answer** |
| --- | --- |
| What does @Column(nullable = false) do? | Prevents null values in DB column. |
| Why use updatable = false? | Prevents JPA from changing the value once inserted. |
| What is the use of @JsonFormat? | Formats Java date fields into specific JSON date format. |
| Why use EnumType.STRING in @Enumerated? | Safer, more readable, avoids bugs if enum order changes. |
| What if @Enumerated is not used? | JPA defaults to ORDINAL, storing enum index as integer. |
| What are alternatives to LocalDateTime? | Date, LocalDate, ZonedDateTime. |
| Can we assign default enum value? | Yes, like bookingStatus = BookingStatus.PENDING;. |

**✅ Summary**

| **Feature** | **Purpose** |
| --- | --- |
| @Column | Maps Java field to DB column |
| nullable = false | Prevents null |
| updatable = false | Makes field immutable post-insert |
| @JsonFormat | Controls JSON format of date |
| @Enumerated(EnumType.STRING) | Stores enum as string |
| LocalDateTime | Java 8+ class for timestamp |
| Default enum | Prevents null in status field |

| **Type** | **Description** |
| --- | --- |
| Uni-directional | Only BookingHeader knows about Tours |
| Bi-directional | Both entities reference each other |

Great question! Understanding why @ManyToOne is **EAGER** by default and @OneToMany is **LAZY** helps you write better and more performant JPA/Hibernate code.

**✅ First, the Table Again for Reference**

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

**🔍 Why @ManyToOne is EAGER by Default**

**➤ Reason:**

* Usually, when you load a child entity, you **almost always** need the parent.

**✅ Example:**

@Entity

class Passenger {

@ManyToOne

private Booking booking; // Parent

}

If you're loading a Passenger, it makes sense to also get its Booking, because a passenger doesn't make sense without the context of the booking.

**📌 Analogy:**

Think of a **Bank Transaction**:

* A transaction always belongs to an **account**.
* If you're viewing a transaction, you likely want to know which account it's from.

**🔍 Why @OneToMany is LAZY by Default**

**➤ Reason:**

* A parent can have **thousands of children**.
* Fetching all children **every time you load the parent** would be **very slow and memory-heavy**.

**✅ Example:**

@Entity

class Booking {

@OneToMany(mappedBy = "booking")

private List<Passenger> passengers;

}

If you load 1000 bookings and passengers are eagerly fetched, it could load **millions of records**, most of which you don’t need immediately.

**📌 Analogy:**

* When you open a **YouTube Channel**, it doesn't load **all 1000 videos** at once.
* Instead, it loads basic info first, and **more only when you scroll** (Lazy Load).

**✅ Summary**

| **Relationship** | **Why Default FetchType?** |
| --- | --- |
| @ManyToOne | Parent is usually small and needed immediately |
| @OneToMany | Child list can be huge, so avoid loading unless needed |

**💡 Interview Questions and Answers**

| **❓ Question** | **✅ Answer** |
| --- | --- |
| What is the default fetch type for @ManyToOne and why? | EAGER – because parent data is usually needed immediately. |
| Why is @OneToMany default LAZY? | To avoid performance issues by loading large child collections only when required. |
| How can you change the fetch strategy? | Using fetch = FetchType.LAZY or fetch = FetchType.EAGER explicitly in the annotation. |
| What is LazyInitializationException in Hibernate? | Happens when you try to access a lazy-loaded entity **outside of a transaction or session**. |

**🛠 Example Fix (If Lazy Load Fails):**

@OneToMany(fetch = FetchType.EAGER)

private List<Passenger> passengers;

But only do this when you're sure you **always need** the passengers.

Let me know if you want:

* Diagram/ERD explaining it visually
* Example REST API call with LAZY vs EAGER loading behavior
* How to use DTO to avoid loading unnecessary data in large collections

I'm here to help!