Spring Data JPA - Interview Questions & Answers

1. Introduction to JPA

Q1: What are the key differences between JDBC and Spring Data JPA?

- **JDBC** is low-level, requiring manual connection management, SQL query writing, and result set parsing.
- **Spring Data JPA** abstracts all of this; it provides CRUD operations, pagination, and custom queries without boilerplate code.

Q2: How is Spring Data JPA different from Hibernate?

- Hibernate is a **JPA provider**; it implements JPA specifications.
- Spring Data JPA is a **framework built on top of JPA (and Hibernate)** to reduce boilerplate and simplify repository development.
- With Spring Data JPA, developers can define **interfaces** and get query methods auto-implemented.
- Hibernate requires more **manual coding** for query logic and session handling.

Q3: What are the advantages of using Spring Data JPA?

- Reduces boilerplate code using CrudRepository / JpaRepository.
- Automatic implementation of query methods based on method names.
- Built-in support for pagination and sorting.
- Easy integration with Spring Boot and REST.
- Declarative transaction management via annotations (@Transactional).
- Works seamlessly with popular databases and JPA providers.

4. What are JPA repositories in Spring Data JPA?

Q: Explain the purpose of interfaces like CrudRepository, JpaRepository, and PagingAndSortingRepository.

- CrudRepository: Basic CRUD operations.
- PagingAndSortingRepository: Adds pagination and sorting capabilities.

- JpaRepository: Extends both above; adds JPA-specific methods like batch operations, flushing, etc.

Q5: What are query methods and how does Spring Data JPA resolve them?

- Methods like findByName, findByAgeGreaterThan follow naming conventions.
- Spring parses the method name and auto-generates the corresponding SQL/JPQL.
- No need to write custom queries unless needed.

Q6: When would you use @Query annotation over derived query methods?

Use **derived methods** when the query is simple and can be expressed in method names.

- Use @Query for:
    - Complex queries
    - Joins
    - Native SQL
    - When using projection or dynamic parameters

Q7: How are transactions managed in Spring Data JPA?

- Spring uses @Transactional to manage transactions.
- Read-only operations can be marked as @Transactional(readOnly = true).
- Transactions are propagated based on Spring's propagation types (REQUIRED, REQUIRES_NEW, etc.).

Q8: What are lazy and eager fetching strategies in JPA?

- **Lazy**: Related entities are loaded only when accessed (default for @OneToMany, @ManyToMany).
- **Eager**: Related entities are loaded immediately with the parent (default for @ManyToOne, @OneToOne).
- Lazy loading is more efficient but may cause LazyInitializationException outside a transaction.

Q9: How can you define custom queries that aren't supported by method names?

- Use @Query("SELECT u FROM User u WHERE u.name = ?1")

- Or native SQL with @Query(value = "SELECT * FROM users WHERE name = ?", nativeQuery = true)
- Also use @Modifying for update/delete queries.

Q10. What is an Entity in JPA?

A POJO annotated with @Entity, representing a table in the database.

Q11. What is the use of @Id and @GeneratedValue?

Defines the primary key and generation strategy (e.g., AUTO, IDENTITY, SEQUENCE).

Q13. How do you handle One-to-Many and Many-to-One relationships in JPA?

Using @OneToMany, @ManyToOne with @JoinColumn or mappedBy for bidirectional mapping.

Q14. What are DTO projections and how are they used in Spring Data JPA?

Interfaces or classes that define a subset of fields returned by queries. Useful for performance and API responses.

1. If you're asked to switch the database from MySQL to PostgreSQL in a large Spring Boot app, how would JPA help minimize the effort involved?

Answer: JPA abstracts data access logic, allowing developers to switch between databases by only changing connection properties and dialect. Entity mappings and repository interfaces stay unchanged.

2. You're working in a microservice that exposes REST APIs. Why would JPA still be useful even if the API isn't directly coupled to DB schemas?

Answer: JPA acts as an ORM abstraction, helping translate DB rows to Java objects. This helps decouple services from low-level SQL logic while allowing flexible querying and validation.

3. If your entities are designed for JPA, can they be reused in a Hibernate-only app? Why?

Answer: Yes. JPA is just an API. Hibernate is a JPA provider, so JPA-annotated entities work with Hibernate. But Hibernate-specific annotations might not be portable to other JPA providers.

2. Benefits of Spring Data JPA

4. How would you justify avoiding boilerplate DAO code using Spring Data JPA? Can you explain this with an example?

Answer: Spring Data JPA eliminates DAO boilerplate by auto-generating queries via method names. E.g., findTop3ByOrderByRatingDesc() directly fetches data without writing queries:

```
List<Restaurant> findTop3ByOrderByRatingDesc();
```

5. Your project requires rapid prototyping with minimal database complexity. Why is Spring Data JPA preferred over Hibernate alone?

Answer: Spring Data JPA integrates with Spring Boot for autoconfiguration, simplifies CRUD operations, and supports query derivation—offering a productivity boost over plain Hibernate.

6. Why does Spring Data JPA reduce error rates in queries compared to string-based native SQL in JDBC?

Answer: Derived queries in Spring Data JPA generate queries based on method names, reducing risk of syntax errors and ensuring compile-time checks.

3. JPA vs Hibernate vs JDBC

7. Imagine you're joining a legacy project using plain JDBC. The team is struggling with connections and result sets. What would your migration approach be, and where would JPA fit in?

Answer: I'd recommend migrating to JPA for ORM capabilities and using Spring Data JPA for repository abstraction. It reduces boilerplate and simplifies transaction, mapping, and query handling.

8. You need direct control over SQL but want the benefits of entity mapping. Which JPA features allow this?

Answer: Use @Query with nativeQuery = true to write native SQL while still mapping results to entity objects.

4. Setting Up Spring Data JPA

9. Why would you get "No qualifying bean of type JpaRepository" error in a Spring Boot project?

Answer: Reasons could be:

- Repository interface is not in a scanned package.
- @EnableJpaRepositories points to the wrong package.
- Entity classes aren't detected (@EntityScan issue).

10. You've set up JPA but encounter table-not-found errors. Yet your entity class is annotated with `@Entity`. What could be wrong?

Answer: @Table annotation might be missing or misconfigured, or spring.jpa.hibernate.ddl-auto might be set to none preventing schema generation.

5. Entity Mapping Annotations

11. You have a field in your entity that should be calculated at runtime but not stored in the DB. How would you annotate it? Can it still be used in projections?

Answer: Annotate with @Transient. It's excluded from persistence but can be included in custom DTOs if mapped manually.

```
@Transient
private double distanceFromUser;
```

12. You forgot to annotate the primary key in an entity. Does it throw any exceptions? If yes Why?

Answer: Yes, it will throw a runtime exception during application startup. JPA requires every entity to have a field annotated with @Id to uniquely identify each instance for persistence operations. If it's missing, Hibernate (as the JPA provider) cannot manage the entity. There's no way for the persistence context to distinguish or track objects without a primary key.

6. Basic CRUD and Repository

13. What happens if you define a repository interface but forget to extend JpaRepository?

Answer: Spring won't recognize it as a repository bean, and CRUD methods like save() or findAll() won't be available.

14. You use @GeneratedValue on a non-primary key field. What issue might arise?

Answer: The @GeneratedValue annotation is specifically intended for use with fields annotated with @Id, as it instructs the persistence provider to auto-generate the value for the primary key. If you apply @GeneratedValue to a non-@Id field, JPA will throw a mapping exception during startup.

7. Derived Query Methods

15. You define a derived query method like findByRatingGreaterThan(Double rating). What happens if rating is null?

Answer: The method will throw a QueryMethodParameterMismatchException at runtime. Spring Data JPA does not handle null values in derived query methods gracefully for comparison operators like GreaterThan, LessThan, etc. Since it tries to translate the method name into a concrete query, passing null leads to an invalid predicate, which results in the exception.

16. How would you write a method to fetch the top 5 restaurants by location sorted by rating using derived query feature of Spring Data JPA?

Answer:

```
List<Restaurant> findTop5ByLocationOrderByRatingDesc(String location);
```

8. Custom Queries with @Query

17. You need a report of restaurant names along with their respective order counts. How would you implement this using JPQL in Spring Data JPA?

Answer: You can use a custom JPQL query with a JOIN and GROUP BY clause:

```
@Query("SELECT r.restaurantName, COUNT(o) FROM Restaurant r JOIN
r.ordersList o GROUP BY r.restaurantName")
List<Object[]> getRestaurantOrderCounts();
```

18. You want to implement a flexible search that filters restaurants by cuisine and, optionally, by rating (if provided). What kind of query would allow you to handle nullable parameters effectively in Spring Data JPA?

Answer: Use a custom JPQL query with conditional filtering using IS NULL OR to handle optional parameters:

```
@Query("SELECT r FROM Restaurant r WHERE r.cuisine = :cuisine AND
(:rating IS NULL OR r.rating > :rating)")
List<Restaurant> search(@Param("cuisine") String cuisine,
@Param("rating") Double rating);
```

19. You want to fetch restaurants that have more than 5 associated orders. How can you achieve this using JPQL in Spring Data JPA?

Answer: You can use JPQL's SIZE() function to filter based on the number of elements in a collection:

```
@Query("SELECT r FROM Restaurant r WHERE SIZE(r.ordersList) > 5")
List<Restaurant> findRestaurantsWithMoreThanFiveOrders();
```

20. You mistakenly pass a string like "high" to a Spring Data JPA query method that expects a Double. What would happen?

Answer: Spring Data JPA will throw a QueryMethodParameterMismatchException at runtime. This occurs because Spring Data attempts to bind the parameter "high" to a Double field (e.g., rating), and the conversion fails. Since derived queries and @Query methods are type-sensitive, any mismatch between the declared parameter type and the actual input causes an exception.

21. What happens if you have orphanRemoval = true on @OneToMany and you clear the collection?

Answer: All associated child records are deleted from the DB when the parent is saved.

9. One-to-One Mapping (@OneToOne)

22. In a @OneToOne relationship, the foreign key isn't being saved in the database. What might be missing?

Answer: The @JoinColumn annotation is likely missing on the owning side. Hibernate requires it to determine where the foreign key should be stored.

23. While serializing a User entity that references a UserProfile via @OneToOne, infinite recursion occurs. How can this be resolved?

Answer: Apply @JsonManagedReference on the parent side and @JsonBackReference on the child to prevent cyclic serialization:

```
// User.java
@OneToOne
@JsonManagedReference
private UserProfile profile;

// UserProfile.java
@OneToOne(mappedBy = "profile")
@JsonBackReference
private User user;
```

24. Deleting an entity (e.g., User) doesn't cascade to its associated UserProfile, despite using cascade settings. What's going wrong?

Answer: Cascade settings must be applied on the owning side (i.e., where @JoinColumn is present). Applying cascade on the inverse (mappedBy) side has no effect.

25. Despite setting fetch = FetchType.LAZY on a @OneToOne mapping, join queries are still executed immediately. Why?

Answer: By default, Hibernate treats @OneToOne relationships as eagerly fetched. Enforcing lazy loading requires bytecode enhancement or specific provider-level configurations.

10. One-to-Many / Many-to-One Mapping

26. A @OneToMany relationship results in the creation of a join table in the database. What could explain this?

Answer: This typically happens when mappedBy is not specified. Without it, JPA treats the relationship as unidirectional and generates a separate join table.

27. Removing a parent entity doesn't delete its associated children in a @OneToMany relationship. What configuration might be missing?

Answer: Ensure that either cascade = CascadeType.REMOVE or orphanRemoval = true is included in the @OneToMany mapping to propagate deletions to child entities.

28. A List<Comment> inside a Post entity results in excessive queries (N+1 issue). What likely causes this behavior?

Answer: Using FetchType.EAGER with @OneToMany can cause this. Switch to FetchType.LAZY and consider using fetch joins or DTO projections to optimize performance.

29. After associating a child (Order) with a parent (Customer), the relationship isn't saved in the database. What might be missing?

Answer: Both sides of the relationship need to be set. Example:

```
order.setCustomer(customer);
customer.getOrders().add(order);
```

30. Why is it discouraged to use cascade = ALL on large @OneToMany collections?

Answer: This can cause all child elements to be processed during every persistence operation, leading to performance bottlenecks even for unchanged records.

11. Many-to-Many Mapping (@ManyToMany, @JoinTable)

31. In a bidirectional @ManyToMany mapping between Student and Course, saving only one side doesn't update the join table. Why?

Answer: Both sides of the relationship must be updated explicitly:

```
student.getCourses().add(course);
course.getStudents().add(student);
```

32. A @ManyToMany relationship needs to store additional data like enrollmentDate. How can this be modeled?

Answer: Introduce an intermediate entity (e.g., Enrollment) with two @ManyToOne associations and the additional field:

```
@Entity
public class Enrollment {
    @ManyToOne private Student student;
    @ManyToOne private Course course;
    private LocalDate enrollmentDate;
}
```

33. What's the best way to retrieve students of a course, sorted by enrollment date (stored in the join entity)?

Answer: Query the Enrollment entity using JPQL:

```
@Query("SELECT e.student FROM Enrollment e WHERE e.course.id = :cid
ORDER BY e.enrollmentDate DESC")
```

34. A @ManyToMany is defined without a @JoinTable. What will JPA do by default?

Answer: Hibernate automatically creates a join table using the alphabetical order of entity names and default foreign key column names.

35. How would you modify a query to fetch employees who have not submitted any reports yet?
Answer*:* Knowledge of `LEFT JOIN` with `IS NULL` to fetch unlinked data, e.g.,

@Query("SELECT e FROM Employee e LEFT JOIN e.reports r WHERE r IS NULL")

36. A list of addresses is not being fetched for a Customer entity, even though the mapping is correct. What would you check first?
Answer*:* Look at fetch type (lazy vs eager), if the session is open, and if data exists in the child table.

37. What's the difference in DB schema generated for `@OneToMany` with and without `mappedBy`?
Answer*:* Without `mappedBy`, a join table is created. With it, the foreign key is managed by the child side.

38. You change a collection in a parent entity and call `save()`, but no updates are seen in the DB. Why?
Answer*:* The collection is not managed (e.g., no `orphanRemoval`, or `@Transactional` missing), or change detection did not pick up the difference.

39. What are the risks of having a large collection with `cascade = CascadeType.ALL` in a OneToMany mapping?
Answer*:* Performance overhead, unintended mass updates or deletions.

40. What is the key behavioral difference between `@JsonIgnore` and `@JsonBackReference`? When would you prefer one over the other?
Answer:

- `@JsonIgnore` hides the field from both serialization and deserialization.
- `@JsonBackReference` allows deserialization but suppresses serialization in a bidirectional setup.
  Prefer `@JsonBackReference` for avoiding loops in bidirectional mappings, and `@JsonIgnore` for hiding sensitive or redundant data.

41. After adding `@JsonIgnore` to a field, you find JPA still tries to persist it. Why is that, and is it a problem?
Answer*:* Jackson annotations affect only JSON serialization; persistence is controlled by JPA annotations like `@Transient`.

42. What is the difference between @JsonIgnore and @JsonIgnoreProperties in Spring Boot (Jackson)?

Answer:

@JsonIgnore is used when I want to ignore a **specific field** during JSON serialization or deserialization. I usually apply it directly on the field or its getter method. For example, if a class has a password field, and I don't want that to appear in the JSON response, I simply annotate it with @JsonIgnore.

On the other hand, @JsonIgnoreProperties is more flexible. It can be used at the **class level** when I want to ignore **multiple properties** at once. I can pass an array of field names to it. For example, if I want to exclude both password and email, I'd use @JsonIgnoreProperties({"password", "email"}) at the class level.

@JsonIgnoreProperties can also be used at the **field level**, especially for nested objects. For instance, if I have an Employee class with a Department object, and I want to ignore some fields from the department during serialization—like location—I can annotate the department field with @JsonIgnoreProperties({"location"}). This is useful when I don't want to modify the Department class itself.

Q43: Can we use @JsonIgnore and @JsonIgnoreProperties together?

Answer:

Yes, we can use both annotations together if needed. While it's not very common, there are cases where we might use @JsonIgnoreProperties at the class level to ignore multiple fields, and still use @JsonIgnore for a field that needs to be handled specifically — for example, if we want to exclude it not just from serialization, but also control how it's deserialized.

But generally, we try to choose one based on the use case — @JsonIgnore for single fields and @JsonIgnoreProperties for multiple or nested field control.

Q44: Can we use these annotations to solve circular reference issues?

Answer:

Not really. @JsonIgnore can sometimes help in breaking circular references — like in bidirectional relationships between entities — but it's not the ideal solution because it might hide important parts of the object graph from being serialized.

The better way to handle circular references in Spring Boot is by using Jackson annotations like:

- @JsonManagedReference on the parent side
- @JsonBackReference on the child side

These annotations tell Jackson to serialize only the parent side and ignore the back-reference during serialization, which prevents infinite recursion but still maintains the relationship.

Another option is to use @JsonIdentityInfo which assigns IDs to objects and refers back using the same ID, which is useful when I want to keep bidirectional relationships in JSON.

Q45: What happens if we don't handle circular references properly?

Answer:

If circular references like parent-child relationships aren't handled, Jackson will throw a StackOverflowError or hang during serialization because it keeps trying to expand the same objects again and again. That's why we either break the cycle with @JsonIgnore, or use @JsonManagedReference and @JsonBackReference, or @JsonIdentityInfo depending on the use case.