1. Basic Spring Boot Questions

1. **What is Spring Boot, and why is it used?**
    - Spring Boot simplifies Java application development by providing a framework that eliminates the need for detailed configuration and XML setup. It allows rapid development of production-ready Spring applications with built-in support for various features like embedded servers, auto-configuration, and dependency management.
2. **What are the key features of Spring Boot?**
    - Standalone applications
    - Embedded servers (Tomcat, Jetty, etc.)
    - Auto-configuration
    - Opinionated defaults
    - Production-ready features (metrics, health checks)
    - Spring Boot CLI
3. **What is the difference between Spring and Spring Boot?**
    - Spring is a comprehensive framework with extensive support for building Java-based applications. Spring Boot is a module of Spring that makes it easier to create Spring-based applications with reduced boilerplate code and configuration.
4. **What is Spring Boot Starter?**
    - Spring Boot Starters are dependency descriptors that simplify the inclusion of libraries and frameworks required for particular functionality (e.g., `spring-boot-starter-web`, `spring-boot-starter-data-jpa`).
5. **Explain auto-configuration in Spring Boot.**
    - Auto-configuration automatically configures Spring Beans based on the dependencies present in the classpath. For instance, if `spring-boot-starter-web` is added, Spring Boot auto-configures beans for embedded Tomcat, Jackson, etc.
6. **What is Spring Boot Actuator?**
    - Spring Boot Actuator provides built-in production-ready features like monitoring, metrics, health checks, and application info via endpoints (`/actuator/health`, `/actuator/metrics`).

2. Configuration & Annotations

1. **What is a Spring Boot starter parent?**
    - It's a special `pom.xml` file (`spring-boot-starter-parent`) that defines default configurations and dependency management settings, reducing the need for extra setup.
2. **How do you configure a Spring Boot application?**
    - Spring Boot applications can be configured using `application.properties` or `application.yml` files. Additionally, you can use annotations like `@Value`, `@ConfigurationProperties`, and profile-specific files for environment-based configuration.
3. **What is `@SpringBootApplication` annotation?**

  - It’s a convenience annotation that combines `@SpringBootConfiguration`, `@EnableAutoConfiguration`, and `@ComponentScan`. This is the starting point of any Spring Boot application.
4. **How do you enable a profile in Spring Boot?**
  - Profiles can be enabled via the command line (`--spring.profiles.active=dev`), in the `application.properties` file (`spring.profiles.active=dev`), or programmatically using `SpringApplication.setAdditionalProfiles()`.
5. **Explain the `@ConfigurationProperties` annotation.**
  - It is used to bind external properties (from `application.properties` or `application.yml`) into Java objects in a strongly-typed manner. You annotate a POJO and define the mapping.

3. Advanced Topics

1. **What is the use of embedded servers in Spring Boot?**
   - Spring Boot includes embedded servers like Tomcat or Jetty, allowing you to run web applications as standalone JARs, eliminating the need for deploying WAR files to external servers.
2. **What is a Spring Boot initializer?**
   - Spring Initializer (`start.spring.io`) is a web-based tool that helps to bootstrap a new Spring Boot project with necessary dependencies and configurations quickly.
3. **How does Spring Boot handle dependency management?**
   - Spring Boot uses the `spring-boot-starter-parent` to manage dependency versions automatically. It ensures that compatible versions of libraries are included and eliminates the need to specify individual library versions.
4. **How do you create a custom Spring Boot starter?**
   - To create a custom starter, you need to define a Maven or Gradle module with the necessary dependencies, create an auto-configuration class, and use the `spring.factories` file to register it.
5. **How do Spring Boot and Spring MVC integrate?**
   - Spring Boot simplifies Spring MVC configuration by auto-configuring beans like `DispatcherServlet`, `InternalResourceViewResolver`, etc., if `spring-boot-starter-web` is added to the classpath.
6. **What is the Spring Boot DevTools module?**
   - Spring Boot DevTools provides features like live reload, automatic restarts, and caching during development, improving the developer experience and reducing turnaround times during application changes.

4. Security & Production

1. **How do you secure a Spring Boot application?**
   - Spring Boot applications can be secured using `spring-boot-starter-security`. You can configure authentication and

    authorization rules using annotations like `@EnableWebSecurity`, `@PreAuthorize`, and `@Secured`.

2. **What is Spring Boot's production-ready feature set?**
    - Spring Boot provides production-ready features such as:
        - **Actuator** for health, metrics, and info.
        - **Metrics** integration with tools like Micrometer.
        - **Log configuration** with log levels.
        - **Externalized configuration** support for different environments.
3. **How do you deploy a Spring Boot application to a production environment?**
    - Spring Boot applications can be deployed as executable JARs or WARs. You can deploy them on platforms like AWS, Google Cloud, Heroku, or traditional application servers.
4. **What are some ways to improve the performance of Spring Boot applications?**
    - Use caching mechanisms (e.g., `@Cacheable`).
    - Use asynchronous processing (`@Async`).
    - Optimize database queries (use pagination, lazy loading, etc.).
    - Utilize profiling and monitoring tools like Spring Boot Actuator, Prometheus, Grafana.

5. Microservices & Cloud

1. **What is Spring Cloud, and how does it relate to Spring Boot?**
    - Spring Cloud builds on Spring Boot to provide infrastructure and tools for developing distributed, cloud-native applications. It includes components for service discovery (Eureka), distributed configuration (Config Server), circuit breakers (Hystrix), and more.
2. **Explain service discovery and registration in Spring Boot.**
    - Service discovery (like Eureka) is used to register microservices and allow other services to discover them without hardcoding IP addresses. Spring Boot applications integrate with Spring Cloud Netflix to leverage Eureka for service registration and discovery.
3. **What is a Spring Boot circuit breaker, and how does it work?**
    - Spring Boot Circuit Breaker (from the Resilience4J or Hystrix libraries) helps to prevent service overloads by wrapping remote service calls in a "circuit breaker". If a service is down, the breaker opens and returns a fallback instead of retrying failed requests.
4. **How does Spring Boot handle distributed tracing?**
    - Spring Boot, with Spring Cloud Sleuth and Zipkin, can be used to handle distributed tracing by tagging and tracing requests across different microservices for better debugging and performance monitoring.

6. Testing in Spring Boot

1. **How do you test a Spring Boot application?**
    - Spring Boot supports testing with `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`, and others, which allow you to test various layers (unit, integration, and web layers) of the application.

2. **How does Spring Boot handle MockMvc?**
    - `MockMvc` is used for testing Spring MVC controllers. It allows you to simulate HTTP requests and assert responses without starting an actual server.
3. **What is `@TestConfiguration` in Spring Boot?**
    - `@TestConfiguration` is a specialized `@Configuration` class used in tests to override beans or provide specific beans only for testing.
4. **How can you test a Spring Boot application's REST API?**
    - You can use `@WebMvcTest` to test REST controllers, along with `MockMvc` for simulating and asserting requests and responses.

By understanding these topics, you can prepare for a wide range of Spring Boot interview questions covering different aspects of the framework, from basic concepts to more advanced features like security, microservices, and testing.

4o

You said:
Explain @RestController annotation

ChatGPT said:
ChatGPT

The `@RestController` annotation in Spring Boot is a specialized version of the `@Controller` annotation, used to simplify the creation of RESTful web services. It combines two key annotations: `@Controller` and `@ResponseBody`. Here's a detailed explanation of how `@RestController` works:

1. Definition

- `@RestController` is used to define a controller in Spring that returns responses directly as JSON or XML (or other formats) rather than rendering views (like JSP, Thymeleaf).
- It is typically applied at the class level and indicates that the methods in this controller will handle HTTP requests and return data directly in the body of the response.

2. Combines `@Controller` and `@ResponseBody`

- **`@Controller`**: Marks the class as a Spring MVC controller, which is responsible for handling web requests.
- **`@ResponseBody`**: This annotation indicates that the return value of a method should be written directly to the HTTP response body, rather than being rendered as a view. This is commonly used in RESTful services where the response is typically JSON or XML.

By using `@RestController`, you don't need to explicitly add `@ResponseBody` to each method. This is automatically applied to all methods in the controller class.

3. Usage Example

java

Copy code

```java
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class MyRestController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable Long id) {
        // Assuming User is a class representing a user entity
        return new User(id, "John Doe", "john@example.com");
    }
}
```

Explanation:

- **`@RestController`**: Marks the class as a RESTful web service.
- **`@GetMapping("/hello")`**: Maps the `/hello` URL to the `sayHello()` method, which returns a plain string (`"Hello, World!"`).
- **`@GetMapping("/user/{id}")`**: Maps the `/user/{id}` URL to the `getUser()` method, which returns a `User` object as JSON (assuming default Jackson serialization).

4. Key Features

- **Simplicity**: By using `@RestController`, you avoid the need to annotate each method with `@ResponseBody`.
- **Data returned as JSON or XML**: The response body is automatically serialized into JSON (by default) or other formats based on the request's `Accept` header.
- **Integration with HTTP methods**: `@RestController` is often used with annotations like `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, etc., which are shortcuts for `@RequestMapping(method = ...)` to map specific HTTP methods to controller methods.

5. Comparison with `@Controller`

- In a traditional `@Controller`, you often use views (like JSP or Thymeleaf) to render HTML or UI components. You would explicitly use `@ResponseBody` for methods that need to return data (like JSON or XML).
- `@RestController` simplifies REST API creation by assuming that all methods will return data to the response body, eliminating the need for `@ResponseBody`.

6. When to Use `@RestController`?

- Use `@RestController` when building RESTful APIs where the controller's main responsibility is to handle web requests and respond with data (usually JSON).
- Use `@Controller` when building traditional web applications with HTML views or templates.

7. Serialization with `@RestController`

- The response data is typically serialized to **JSON** by default, and this is handled automatically by Spring Boot using **Jackson**. You can also configure your application to return **XML** by including the necessary dependencies and adjusting the content negotiation settings.

8. Error Handling in `@RestController`

- You can handle exceptions and return custom error responses using `@ExceptionHandler` or `@ControllerAdvice` annotations, which can be used globally or locally within the `@RestController`.

Example of Exception Handling:

java

Copy code

```
@RestController
@RequestMapping("/api")
public class MyRestController {

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable Long id) {
        if (id == null || id < 1) {
            throw new IllegalArgumentException("Invalid ID");
        }
        return new User(id, "John Doe", "john@example.com");
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String>
handleException(IllegalArgumentException e) {
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}
```

In this example, if the user passes an invalid ID, the exception handler will return a custom error message with a `400 Bad Request` status.

Summary:

- **`@RestController`** simplifies the development of RESTful services by automatically serializing data into the response body.
- It eliminates the need for using `@ResponseBody` on each method.
- Works seamlessly with HTTP request mappings like `@GetMapping`, `@PostMapping`, etc.
- Used to return data (usually JSON or XML) directly in the response, making it ideal for building APIs.

4o

You said:

How does Actuator work?

ChatGPT said:

ChatGPT

Spring Boot Actuator Overview

Spring Boot Actuator is a powerful feature that provides built-in tools to monitor and manage Spring Boot applications in production or development environments. It exposes various endpoints that allow you to interact with and monitor the application's internals, such as its health, metrics, environment, and configuration properties.

Here’s a detailed explanation of how **Spring Boot Actuator** works:

---

1. How to Enable Spring Boot Actuator

To enable Actuator in your Spring Boot project, you simply need to include the `spring-boot-starter-actuator` dependency in your `pom.xml` (Maven) or `build.gradle` (Gradle) file:

Maven:

xml
Copy code

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Gradle:

groovy
Copy code

```groovy
implementation
'org.springframework.boot:spring-boot-starter-actuator'
```

Once this is included, Actuator is automatically available in your application.

---

2. Default Endpoints

After enabling Actuator, several useful endpoints are available. These endpoints provide key information about the state of the application:

- **`/actuator/health`**: Reports the application's health status.
- **`/actuator/metrics`**: Provides various application metrics like memory usage, CPU usage, and HTTP request statistics.
- **`/actuator/info`**: Displays arbitrary application information such as version, description, and more (you can customize this).
- **`/actuator/env`**: Shows the environment properties.
- **`/actuator/loggers`**: Allows you to query and change the logging levels at runtime.
- **`/actuator/beans`**: Displays all Spring beans in the application context.
- **`/actuator/threaddump`**: Shows the thread dump of the application.
- **`/actuator/httptrace`**: Provides the last few HTTP request/response traces (disabled by default).

By default, only the `/actuator/health` and `/actuator/info` endpoints are exposed publicly (on the root `/actuator` endpoint). Other endpoints need to be explicitly enabled or configured.

---

3. Actuator Endpoint Configuration

You can customize which Actuator endpoints are exposed or secured by modifying the `application.properties` or `application.yml` configuration files.

Exposing Endpoints:

To expose all Actuator endpoints:

properties
Copy code

```
management.endpoints.web.exposure.include=*
```

To expose specific endpoints:

properties
Copy code

```
management.endpoints.web.exposure.include=health,info,metrics
```

Securing Endpoints:

You can secure Actuator endpoints by integrating Spring Security into your project. For example, you can restrict access to certain endpoints based on roles:

properties
Copy code

```
management.endpoints.web.exposure.include=health,metrics
management.endpoint.health.roles=ADMIN
management.endpoint.metrics.roles=ADMIN
```

---

4. Health Checks with `/health` Endpoint

The `/actuator/health` endpoint is used to check the health status of the application. By default, it returns a simple `UP` or `DOWN` status.

- **Custom Health Indicators**: You can create custom health indicators by implementing the `HealthIndicator` interface. This allows you to check the health of various components such as databases, message brokers, or external APIs.

Example:

java
Copy code

```
@Component
public class CustomHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        // Custom logic to determine if the service is healthy
        boolean isServiceHealthy = checkServiceHealth();
        if (isServiceHealthy) {
            return Health.up().build();
        } else {
            return Health.down().withDetail("Error", "Service not
available").build();
        }
    }
}
```

Health Groups:

You can also group various health indicators to separate critical from non-critical checks:

properties
Copy code

```
management.endpoint.health.group.liveness.include=diskSpace,db
management.endpoint.health.group.readiness.include=diskSpace
```

---

5. Metrics with `/metrics` Endpoint

The `/actuator/metrics` endpoint provides various system and application-level metrics, such as memory usage, CPU usage, HTTP request stats, and more.

- **Micrometer Integration**: Actuator uses **Micrometer**, a metrics collection library, to expose these metrics. Micrometer can also integrate with external monitoring systems such as Prometheus, Graphite, and New Relic.
  Some commonly used metrics include:
    - `jvm.memory.used`: Memory usage
    - `jvm.threads.live`: Live thread count
    - `http.server.requests`: HTTP request statistics
    - `system.cpu.usage`: CPU usage

You can view a specific metric by calling the metric name:

bash
Copy code

```
/actuator/metrics/jvm.memory.used
```

Custom Metrics:

You can define custom metrics in your application using the `MeterRegistry` provided by Micrometer:

java
Copy code
```
@Autowired
private MeterRegistry meterRegistry;

@PostConstruct
public void setupCustomMetrics() {
    meterRegistry.gauge("custom.metric", this, value ->
getCustomMetricValue());
}
```

---

6. Customizing `/info` Endpoint

The `/actuator/info` endpoint provides general application information. You can add custom info, like version or build information, using the `application.properties` file.

Example:

properties
Copy code
```
info.app.name=My Spring Boot Application
info.app.version=1.0.0
info.app.description=This is a custom description for my app.
```

The `/actuator/info` endpoint will now display this information:

json
Copy code
```
{
  "app": {
    "name": "My Spring Boot Application",
    "version": "1.0.0",
    "description": "This is a custom description for my app."
  }
}
```

You can also add dynamic information by using a custom `InfoContributor` class:

java
Copy code

```java
@Component
public class CustomInfoContributor implements InfoContributor {
    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("customDetail", "Custom Info Data");
    }
}
```

---

7. Loggers with `/loggers` Endpoint

The `/actuator/loggers` endpoint allows you to view and change logging levels at runtime. For instance, you can dynamically adjust the logging level of specific classes without restarting the application.

To change a logger's level:

bash
Copy code

```bash
POST /actuator/loggers/com.example.MyClass
{
  "configuredLevel": "DEBUG"
}
```

---

8. Tracing Requests with `/httptrace`

The `/actuator/httptrace` endpoint provides information about the last few HTTP requests and responses made to the application. This is useful for monitoring traffic and debugging issues.

It is disabled by default. To enable it:

properties
Copy code

```properties
management.trace.http.enabled=true
```

---

9. Integration with Monitoring Tools

Spring Boot Actuator can integrate with several external monitoring systems using Micrometer, such as:

- **Prometheus**: For gathering time-series metrics.
- **Grafana**: For visualizing Prometheus data.
- **New Relic, Datadog, StatsD**: For cloud-based monitoring.

To integrate with Prometheus, include the following dependency:

xml
Copy code
```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Afterward, the metrics will be available at the `/actuator/prometheus` endpoint.

Summary

- **Spring Boot Actuator** exposes operational information about the running application via a series of HTTP endpoints.
- It offers built-in health checks, metrics, environment details, and logging capabilities.
- You can customize which endpoints are exposed, who has access to them, and how they are secured.
- Actuator integrates with external monitoring systems (via Micrometer) and can be customized with custom health indicators, metrics, and info.

Actuator is a powerful tool for managing and monitoring Spring Boot applications, especially in production environments.

4o
Custom health checks?
Actuator with Prometheus?

Here are some commonly asked **Spring Boot JPA interview questions** along with answers, categorized into basic, advanced, and configuration-related topics:

1. Basic Spring Boot JPA Questions

1. **What is Spring Data JPA?**
    - **Answer**: Spring Data JPA is part of the larger Spring Data family, simplifying the data access layer by providing repositories and integrating with Java

Persistence API (JPA). It offers an abstraction over the traditional JPA APIs, making it easier to interact with databases and reducing boilerplate code.

2. **What is JPA? How does it differ from Hibernate?**
   - **Answer**: Java Persistence API (JPA) is a specification for object-relational mapping (ORM) and managing relational data in Java applications. Hibernate is an implementation of the JPA specification, providing additional features and options for ORM.
3. **What is the purpose of `@Entity` in JPA?**
   - **Answer**: The `@Entity` annotation is used to mark a class as a JPA entity, meaning that it is mapped to a database table. Each instance of the entity class corresponds to a row in the table.
4. **What is the role of `@Id` in a JPA entity?**
   - **Answer**: The `@Id` annotation is used to mark a field as the primary key of the entity. This field uniquely identifies each record in the corresponding database table.
5. **What is `@GeneratedValue` in JPA?**
   - **Answer**: `@GeneratedValue` is used to indicate that the value of the primary key will be automatically generated by the database. It can have different strategies like `AUTO`, `IDENTITY`, `SEQUENCE`, and `TABLE` for key generation.
6. **How do you define a one-to-many relationship in JPA?**
   - **Answer**: A one-to-many relationship is defined using the `@OneToMany` annotation. It indicates that one entity (parent) is related to many entities (children).

java

Copy code

```
@OneToMany(mappedBy = "parent")
private List<Child> children;
```

7.
8. **What is `@Repository` in Spring Data JPA?**
   - **Answer**: `@Repository` is a Spring annotation used to indicate that a class is a data repository and should be treated as a component. Spring Data JPA uses this annotation to provide exception translation (convert JPA exceptions into Spring DataAccessException).

---

2. Advanced Spring Boot JPA Questions

1. **What is `CrudRepository` and `JpaRepository` in Spring Data JPA?**
   - **Answer**: `CrudRepository` and `JpaRepository` are interfaces provided by Spring Data JPA to perform database operations.
     - **`CrudRepository`**: Provides basic CRUD operations (`save()`, `findById()`, `delete()`, etc.).

        - **`JpaRepository`**: Extends `CrudRepository` and adds more JPA-specific operations like pagination (`findAll(Pageable pageable)`), flushing (`flush()`), and batch operations.
2. **How do you create a custom query in Spring Data JPA?**
    - **Answer**: Custom queries can be created using:

Query method names: Define a method in the repository interface with a name that follows the Spring Data JPA query generation convention.
java
Copy code

```
List<User> findByLastName(String lastName);
```

        -

`@Query` annotation: You can write custom JPQL or native SQL queries using `@Query`.
java
Copy code

```
@Query("SELECT u FROM User u WHERE u.lastName = :lastName")
List<User> findUsersByLastName(@Param("lastName") String lastName);
```

        -
3. **What is the difference between `@Query` and `@NamedQuery`?**
    - **Answer**:
        - **`@Query`**: Used to define queries directly in the repository method using JPQL or SQL.

`@NamedQuery`: Predefined queries that are defined at the entity level and reused across multiple repositories.
java
Copy code

```
@NamedQuery(name = "User.findByLastName", query = "SELECT u FROM
User u WHERE u.lastName = :lastName")
```

        -
4. **Explain the `@Transactional` annotation in Spring Data JPA.**
    - **Answer**: `@Transactional` is used to manage transactions. By default, Spring Data JPA methods like `save()`, `delete()`, and `update()` are transactional. You can also use this annotation to define custom transactional behavior (e.g., read-only transactions).
5. **What is the difference between `fetch = FetchType.LAZY` and `fetch = FetchType.EAGER`?**
    - **Answer**:
        - **`FetchType.LAZY`**: The associated entities are loaded only when they are accessed (on-demand fetching). It improves performance by reducing the initial data loading.

    - ▪ `FetchType.EAGER`: The associated entities are loaded immediately when the parent entity is loaded. It can lead to performance issues if many associated entities are loaded unnecessarily.

6. **How do you implement pagination and sorting in Spring Data JPA?**
   - **Answer**: You can implement pagination using the `Pageable` interface and sorting using the `Sort` interface.

java
Copy code

```
Page<User> findByLastName(String lastName, Pageable pageable);
List<User> findAll(Sort sort);
```

Example of pagination:
java
Copy code

```
Pageable pageable = PageRequest.of(0, 10,
Sort.by("lastName").ascending());
Page<User> page = userRepository.findByLastName("Doe", pageable);
```

7.
8. **What are projections in Spring Data JPA?**
   - **Answer**: Projections are used to retrieve a subset of an entity's fields instead of the whole entity. This can be achieved using **interface-based projections** or **DTO-based projections**.

Interface-based projection:
java
Copy code

```
interface UserProjection {
    String getFirstName();
    String getLastName();
}
```

Repository method:
java
Copy code

```
List<UserProjection> findByLastName(String lastName);
```

    - ▪

---

3. Configuration and Performance Optimization

1. **What is the role of `application.properties` or `application.yml` in Spring Boot JPA?**

- **Answer**: Spring Boot uses `application.properties` or `application.yml` to configure database settings, Hibernate/JPA configurations, and other application-specific settings.
    - Example for configuring a database:

properties
Copy code

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

-

2. **What is the use of `spring.jpa.hibernate.ddl-auto`?**
    - **Answer**: This property controls how Hibernate handles database schema generation:
        - `none`: No action is taken.
        - `validate`: Validates the schema against the database, but does not modify it.
        - `update`: Automatically updates the schema.
        - `create`: Drops and creates the schema at startup.
        - `create-drop`: Drops the schema when the session ends.
3. **How do you optimize database performance in Spring Boot JPA?**
    - **Answer**: Common performance optimization techniques include:
        - Using **lazy loading** for associations to avoid unnecessary data fetching.
        - Implementing **pagination** and **batch processing** for large datasets.
        - **Caching** using second-level caches (like EhCache or Redis).
        - Reducing the number of SQL queries using **`@EntityGraph`** to fetch associated entities more efficiently.
        - Tuning **query performance** with proper indexing and query optimization.
4. **What is `@Modifying` in Spring Data JPA?**
    - **Answer**: The `@Modifying` annotation is used with `@Query` for write operations such as `INSERT`, `UPDATE`, or `DELETE`. It tells Spring Data JPA that the query modifies the data.

java
Copy code

```
@Modifying
@Query("UPDATE User u SET u.status = 'ACTIVE' WHERE u.id = :id")
int activateUser(@Param("id") Long id);
```

5.

6. **What is `EntityManager` in JPA?**

Answer: The `EntityManager` is the core interface in JPA that manages the lifecycle of entities. It allows you to perform CRUD operations, create queries, manage transactions, and interact with the persistence context.
java
Copy code

```
@PersistenceContext
private EntityManager entityManager;
```

○

7. **What is a persistence context in JPA?**
    - **Answer**: A persistence context is a set of entity instances that are managed by `EntityManager` within a specific transaction or session. It ensures that entity changes are tracked and automatically synchronized with the database at the transaction boundary.

---

4. Additional JPA Concepts

1. **What is the N+1 select problem in JPA?**
    - **Answer**: The N+1 select problem occurs when an application executes one query to retrieve a list of parent entities and then performs additional queries to fetch each child entity (one query for each parent). This leads to performance issues. To solve this, you can use **`fetch = FetchType.EAGER`** or **`JOIN FETCH`** to retrieve all associated entities in one query.
2. **Explain the concept of dirty checking in JPA.**

Explain the concept of dirty checking in JPA.

- **Answer**: Dirty checking is the process by which JPA tracks changes to entity objects in the persistence context. If any changes are detected, JPA automatically synchronizes the changes with the database during a transaction commit without explicit `save()` calls.