☕ Java & Spring Interview Questions for Experienced Developers

Ajit Gupta
Follow
10 min read · 1 day ago

. . .

🚀 Microservices with Spring Boot — Step-by-Step Beginner Guide

. . .

🧾 Example Scenario

Let's say your app has:

- `Customer` table
- `Order` table
- `Payment` table

We can break this into 3 microservices:

1. **Customer-Service**
2. **Order-Service**
3. **Payment-Service**

Each service:

- Runs on its own port
- Has separate responsibilities
- Can be deployed independently

. . .

✅ Benefits of Microservices

- Faster development & deployment
- Independent scaling
- Easy for new developers to understand
- Fault isolation (if one service fails, others keep working)
- Faster startup time
- Only the changed service needs to be redeployed

. . .

❌ Challenges

- More services = more complexity
- Harder integration and debugging
- Network latency, fault tolerance, and load balancing need to be handled
- Understanding the whole system may be difficult for new developers

. . .

🔨 What We'll Build

We'll build a simple microservice project with:

1. `Customer-Service`
2. `Order-Service` (connected to Customer)
3. `Service-Registry` (Eureka)
4. `API-Gateway`
5. `Config-Server`

. . .

1️⃣ Customer-Service

- Create Spring Boot app from https://start.spring.io
- Add dependencies:
- `spring-boot-starter-web`
- `spring-boot-starter-data-jpa`
- `h2`
- `lombok`

📁 Folder structure:

```
- controller
- entity
- repository
- service
```

📍 Important points:

- `@Entity` class for Customer
- `CustomerRepository` extends `JpaRepository`
- `CustomerService` contains business logic
- `CustomerController` with basic CRUD REST APIs

🔗 Sample APIs:

```
POST  http://localhost:9001/customer/
GET   http://localhost:9001/customer/{id}
```

. . .

2️⃣ Order-Service

Same setup as Customer-Service. Add an extra folder: `VO` (for Value Objects like `Customer`, `ResponseTemplateVO`)

📍 Integration with Customer-Service:

- Add `RestTemplate` bean (with `@LoadBalanced`)

- Use `restTemplate.getForObject(...)` to call Customer-Service

🔗 Sample APIs:

```
POST  http://localhost:9002/order/
GET   http://localhost:9002/order/{id}
GET   http://localhost:9002/order/withCustomer/{id}
```

. . .

3️⃣ Service-Registry (Eureka Server)

- Create Spring Boot app
- Add dependency: `spring-cloud-starter-netflix-eureka-server`
- Annotate main class with `@EnableEurekaServer`

🛠️ application.yml

```
server:
  port: 8761
```

```
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

📍 Register other services by:

- Adding `@EnableEurekaClient`
- Updating `application.yml` with Eureka config
- Naming the services using:

```
spring:
  application:
    name: CUSTOMER-SERVICE
```

📞 Eureka Dashboard: http://localhost:8761

. . .

4️⃣ API-Gateway

- Create Spring Boot app

Add:

- `spring-cloud-starter-gateway`
- `spring-cloud-starter-netflix-eureka-client`

🛠️ application.yml

```
server:
  port: 9191
```

```
spring:
  application:
    name: API-GATEWAY
  cloud:
    gateway:
      routes:
        - id: CUSTOMER-SERVICE
          uri: lb://CUSTOMER-SERVICE
          predicates:
            - Path=/customer/**
        - id: ORDER-SERVICE
          uri: lb://ORDER-SERVICE
          predicates:
            - Path=/order/**
```

🔗 Call APIs through the gateway:

```
http://localhost:9191/customer/
http://localhost:9191/order/
```

. . .

5️⃣ Config-Server

- Create config repo → https://github.com/fatihkirli/config-server

Create Spring Boot app

Add:

- `spring-cloud-config-server`
- `spring-cloud-starter-netflix-eureka-client`

🛠️ application.yml

```
server:
  port: 9296
```

```
spring:
  application:
    name: CONFIG-SERVER
  cloud:
    config:
      server:
        git:
          uri: https://github.com/fatihkirli/config-server
          clone-on-start: true
```

📍 For all other services:

- Add `bootstrap.yml` to load properties from config-server

```
spring:
  cloud:
    config:
      uri: http://localhost:9296
```

- Remove Eureka and port config from `application.yml`

Add:

- `spring-cloud-starter-config`
- `spring-cloud-starter-bootstrap`

. . .

✅ Order of Startup:

1. `service-registry`
2. `config-server`
3. `api-gateway`
4. `customer-service`
5. `order-service`

. . .

🎯 Final Thoughts

We just built a basic **Microservices Architecture** with Spring Boot.

This is just the beginning!

🧱 In the next steps, we'll add:

- Logging (with ELK or Sleuth)
- Security (with Keycloak, OAuth2)
- DTOs & Mapper Classes
- Global exception handling
- Monitoring (with Prometheus + Grafana)
- Docker & Kubernetes
- Frontend integration

. . .

☕ Java & Spring Interview Questions for Experienced Developers — Part 2

. . .

1️⃣ SOAP vs REST — What's the Difference?

SOAP (Simple Object Access Protocol):

- XML-based, strict with WSDL/XSD
- Both service provider and consumer must sync changes
- Supports only POST (other methods are hard or unsupported)
- More complex error handling (e.g. unmarshalling errors)

REST (Representational State Transfer):

- Uses JSON (lighter and faster)

- Flexible: supports POST, GET, PUT, DELETE
- You can keep the same URL and handle all CRUD operations
- Consumers can work with old request formats even after service updates

Real Project Insight: In modern systems, REST is preferred due to flexibility and performance. SOAP's tight coupling causes more production issues during contract updates.

. . .

2️⃣ What is a Spring Bean? What are `@Autowired`, Dependency Injection (DI), and IoC?

Basic Idea:

- **Spring Bean** = Any object managed by Spring
- **@Autowired** = Ask Spring to inject the bean automatically
- **Dependency Injection (DI)** = You don't manually create objects. Spring provides them.
- **Inversion of Control (IoC)** = Spring, not you, controls object creation and lifecycle

In short: If you want to use a class in another class, just use `@Autowired` and let Spring handle the rest (creating, injecting, managing, etc.).

. . .

3️⃣ Hibernate: One-to-One, Many-to-One, etc.

These define **relationships between entities** in your database.

Example: `Person` and `Address`

- `Person` has one `Address`
- Delete Person → Delete Address automatically (use `cascade = CascadeType.ALL`)

⚠️ **Be Careful:**

- If you update the parent, Hibernate may also update the child — even if no actual data changed.
- To avoid unwanted cascades, consider using just IDs instead of full object references in microservices.

Microservices Tip: Avoid tight entity relations. Use `customerId` instead of full `Customer` object, then fetch data via service layer (`getById()`).

. . .

4️⃣ Fetch Types in Hibernate: `EAGER` vs `LAZY`

- **EAGER**: Loads related data immediately
- **LAZY**: Loads only when needed

💡 **Best Practice:** Use `LAZY` loading to reduce memory and database overhead.

Common Issue: LazyInitializationException — when you try to access a lazy-loaded object **outside** of a transaction context.

. . .

5️⃣ What is `@Transactional` in Spring?

@Transactional handles:

- DB connection

- Commit
- Rollback
- Closing the session

🧠 **Tips:**

- Use `@Transactional` in lower layers (like Service)
- Don't put complex logic inside `@Transactional` methods
- Use appropriate propagation levels:
- `REQUIRED` (default): Reuses existing transaction
- `REQUIRES_NEW`: Suspends current, starts a new transaction

Batch Processing Tip:

- One transaction for all records = rollback all if one fails
- One transaction per record = allows partial success, use `REQUIRES_NEW`

Note: Batch jobs are risky if your test data doesn't reflect real PROD data. Test thoroughly.

. . .

6️⃣ JDBC Basics — (Is It Still Relevant?)

JDBC = Java Database Connectivity Used to directly connect to databases and run SQL queries.

⚠️ **Reality Check:**

- Today, most projects use Hibernate or Spring Data JPA
- Direct JDBC is rare unless performance tuning is needed

- If a company still uses JDBC — think twice!

💡 **Example Code:**

```
try (Connection conn = DriverManager.getConnection(...);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT * FROM Employees")) {
    while (rs.next()) {
        System.out.println(rs.getString("first_name"));
    }
}
```

🔴 *Unless required, skip detailed JDBC in interviews.*

. . .

7️⃣ Java Versioning Strategy

Java now follows a **time-based release cycle**:

- New version every 6 months
- LTS (Long-Term Support) versions: **Java 8, 11, 17**
- Oracle JDK (from Java 11 onward) is not free for commercial use
- **Use OpenJDK** for open-source development

🧩 Version Pattern:

```
$FEATURE.$INTERIM.$UPDATE.$PATCH
```

📝 Example:

- Java 17 = LTS
- Java 18, 19 = Non-LTS (short-lived)
- Use LTS in enterprise projects

. . .

📌 Summary

Topic	Key Takeaways
SOAP vs REST	REST is flexible, lightweight, easier to evolve
Spring Core	Use `@Autowired`, DI, and IoC for cleaner code
Hibernate Relations	Use IDs in microservices, be careful with `cascade`
Fetch Types	Use `LAZY` by default
Transactions	Use correct propagation, especially in batch jobs
JDBC	Mostly outdated — skip unless necessary
Java Versions	Prefer LTS releases, OpenJDK is free

. . .

✅ **Pro Tip:** Don't just memorize questions — *understand the logic behind them with real-world context.*

. . .

☕ Java & Spring Interview Questions for Experienced Developers — Part 3

. . .

1️⃣ What is AOP (Aspect-Oriented Programming) & Cross-Cutting Concerns?

AOP helps separate logic that is common across modules — such as:

- Logging
- Security
- Transaction management

These are called **cross-cutting concerns**.

Spring AOP supports:

- `@Before` – runs before the method
- `@AfterReturning` – runs after a method returns
- `@AfterThrowing` – runs after a method throws an exception
- `@Around` – wraps method execution (can run before and after)

⚠️ *Real-time Note:* In modern projects, I've rarely seen AOP directly used — because more advanced tools/libraries are available for logging, tracing, etc. But still, it's good to **understand the concept**.

. . .

2️⃣ What is Microservice Architecture?

Microservices = Break your large application into **independent, smaller services**.

Example Structure:

- `Customer-Service`: Handles customer data
- `Payment-Service`: Handles payment
- `Order-Service`: Handles orders

Each service:

- Has its own database
- Is deployed independently
- Can be developed and tested in isolation

. . .

3️⃣ Pros and Cons of Microservice Architecture

✅ **Pros:**

- Faster development and deployment
- Each team can work independently
- Easy to scale services individually
- Better fault isolation — one service fails, others keep running
- Easy onboarding for new team members
- CI/CD friendly (integration & deployment automation)

❌ **Cons:**

- More complex system architecture
- Difficult to manage as service count grows
- Requires handling:
- Service discovery
- Network latency
- Load balancing
- Data consistency
- Harder to understand the full application

📝 *Note:* Microservices work best with strong DevOps practices and a well-defined team structure.

. . .

4️⃣ Key Spring Annotations You Should Know

Here are **must-know annotations** in Spring and what they do:

Annotation	Purpose
`@Required`	Marks a setter method as required for dependency injection (mainly used with XML config)
`@Autowired`	Injects dependencies automatically
`@Bean`	Declares a bean in a `@Configuration` class
`@Configuration`	Declares a class that defines beans
`@Controller`	Marks a controller class (used in MVC)
`@ResponseBody`	Converts the response to JSON/XML
`@PathVariable`	Binds a URL value to a method parameter
`@RequestMapping`	General mapping for HTTP requests
`@GetMapping`, `@PostMapping`, etc.	Shortcuts for specific HTTP methods

. . .

5️⃣ What is a Spring Bean? What are Bean Scopes?

Spring Bean = An object managed by the Spring container.

Bean Scopes in Spring:

Scope	Description
`singleton`	One instance per Spring container (default)
`prototype`	A new instance every time you request it
`request`	One instance per HTTP request (used in web apps)
`session`	One instance per user session
`global-session`	One instance for global session (mainly in Portlet apps)

💡 *Tip:* In most real projects, **singleton** is used. But knowing other scopes is helpful when working with web apps or multithreading.

💡 *Tip:* In most real projects, **singleton** is used. But knowing other scopes is helpful when working with web apps or multithreading.

. . .

6️⃣ What is a Materialized View?

A **Materialized View** is like a snapshot of data from a table — stored physically.

Feature	View	Materialized View
Stored on disk	❌ (Virtual only)	✅ Yes
Always up-to-date	✅ Yes	❌ Only when refreshed
Faster performance	❌ No	✅ Yes (precomputed)
Uses memory space	❌ Minimal	✅ Takes disk space
Updates	Dynamic on use	Manual or trigger-based

📌 *Tip:* Use Materialized Views when performance is a concern and data doesn't change frequently.

📌 *Tip:* Use Materialized Views when performance is a concern and data doesn't change frequently.

. . .

✅ Final Thoughts

Topic	Key Takeaways
AOP	Helps separate cross-cutting concerns (logging, security, etc.)
Microservices	Divide app into smaller services for better scalability and fault isolation
Spring Annotations	Essential for clean, decoupled Spring apps
Bean Scopes	Mostly singleton, but others useful in web apps
Materialized View	Useful for performance optimization with precomputed data

. . .

☕ Java & Spring Interview Questions for Experienced Developers — Part 4

. . .

1️⃣ What are Design Patterns? Give Examples.

Design patterns are reusable solutions to common software design problems. They're not exact code — but a blueprint to solve problems that can appear again and again in software development.

🔧 **Three main categories:**

1. **Creational** — for object creation
2. **Structural** — for class/object composition
3. **Behavioral** — for communication between objects

📌 **Examples you should know:**

- **Singleton (Creational):** Only one instance of a class is allowed (used in caching, logging, thread pools). ✅ Constructor is made private, and the object is accessed via a static method.
- **Factory Method (Creational):** Used to create objects without exposing the creation logic to the client. ✅ Helps when the object creation logic is complex or needs to vary at runtime.
- **Adapter (Structural):** Converts one interface into another (like a translator). ✅ Helps when integrating legacy code with new systems.
- **Strategy (Behavioral):** Defines a family of algorithms, encapsulates each, and makes them interchangeable. ✅ Useful in scenarios where business rules change dynamically.

🎯 **Real Tip:** Know at least 3–4 well — Singleton, Factory, Strategy, and Adapter are good ones.

. . .

2️⃣ SOLID Principles

These are **5 object-oriented design principles** that help in building clean, scalable, and maintainable applications.

Principle	Meaning
S – Single Responsibility	One class should do one thing only
O – Open/Closed	Open for extension, closed for modification
L – Liskov Substitution	Subclass should replace parent class without breaking the code
I – Interface Segregation	Prefer small, specific interfaces
D – Dependency Inversion	Depend on abstractions, not concrete classes

🛠️ *Tip:* These principles are especially useful when building large, layered, or microservice-based systems.

🛠️ *Tip:* These principles are especially useful when building large, layered, or microservice-based systems.

. . .

3️⃣ Data Consistency & ACID in Microservices

In **monoliths**, ACID is handled easily by relational DB transactions.

ACID Property	Meaning
A – Atomicity	All or nothing
C – Consistency	DB is always valid post-transaction
I – Isolation	No interference between concurrent transactions
D – Durability	Data survives crashes or failures

But in **microservices**, data is distributed. So we need:

🔁 **Distributed Transaction Techniques**:

- **2-Phase Commit (2PC)** Coordinator waits for all services to agree before commit. ✅ Good for small systems ❌ Slower and blocking
- **SAGA Pattern** Break the transaction into smaller local transactions. ✅ Better scalability
- **Choreography-based:** Each service listens to events and triggers the next
- **Orchestration-based:** A central controller (Saga Orchestrator) handles the sequence and rollback (compensation)

⚠️ *Note:* Compensating transactions must be **idempotent** and carefully designed.

. . .

4️⃣ What is Big O Notation?

Big O measures algorithm performance — especially as data grows.

- `O(1)` – Constant time
- `O(n)` – Linear time
- `O(n²)` – Quadratic (nested loops)
- `O(log n)` – Binary search
- `O(n log n)` – Merge sort

🎯 Focus on writing code that minimizes time/space as input grows.

. . .

5️⃣ Database Design Basics

Steps to design a good database:

1. **Requirement Analysis** — Understand business needs
2. **Entity/Table Design** — Define tables
3. **Primary Keys & Foreign Keys** — Set relationships
4. **Normalization** — Remove redundancy, improve structure (1NF, 2NF, 3NF)

🧠 Tip: Normalize first, then denormalize if performance needs arise.

. . .

6️⃣ What is a Java Heap Dump?

A **heap dump** is a snapshot of all Java objects in memory.

Used for:

- Memory leak detection
- Finding unused objects
- Analyzing high memory usage

📂 Tools:

- `jhat`
- **JVisualVM**
- **Eclipse MAT (Memory Analyzer Tool)**

When analyzing a dump:

- Look for large objects

- Track object references and leaks

. . .

7️⃣ How to Improve Java Performance?

💡 **Real Tips from Real Projects:**

✅ Use **only required columns** in DB queries and REST responses

✅ Implement **pagination** for large datasets

✅ Avoid fetching unnecessary data from DB/WS

✅ Use **caching**, but manage it well to avoid memory issues

🚀 **Java-specific optimizations:**

- 🔄 **Use latest Java version** — Better memory and CPU handling
- 🧱 **Use primitive types** — Avoid boxing/unboxing (e.g., prefer `int` over `Integer`)
- 🧮 **Avoid BigDecimal unless needed** — It's slow and memory-heavy
- 🔡 **Use StringBuilder** — Avoid string concatenation with `+` in loops
- 🔄 **Use PreparedStatement over Statement** — Faster DB access and more secure
- 🔁 **Avoid compiling regex every time** — Cache compiled patterns
- 🔧 **Refactor large methods** — Break into smaller chunks for readability and optimization
- 🧠 **Use switch over multiple if-else blocks** — Especially when checking against constants

. . .

🗒️ Summary Table

Topic	Key Notes
Design Patterns	Singleton, Factory, Adapter, Strategy are essential
SOLID	Know all 5 principles — core of good design
ACID & Saga	Required for microservice data consistency
Big O	Know basic notations and optimize logic
DB Design	Normalize, use proper keys, analyze usage patterns
Heap Dump	Analyze memory issues, avoid leaks
Java Performance	Focus on DB, WS, primitives, efficient loops, and caching

Java Microservices Spring Boot Sql

16

Written by Ajit Gupta

165 followers · 208 following

Follow

No responses yet

Write a response

What are your thoughts?

More from Ajit Gupta

Ajit Gupta

🚀 Java Microservices Interview Cheat Sheet (20 Q&A)

Jun 16 46

Ajit Gupta

🚀 Interview Question : Java & Spring Boot Insights🌟 ✅ Q.

Jun 23 10 2

Ajit Gupta

All Oracle Senior Java Developer Interview Questions

Jun 13 11

Ajit Gupta

Java + Spring Boot Interview Questions Real Experience

Jul 4 5

See all from Ajit Gupta

Recommended from Medium

In Stackademic by Kavya's Programming Path

I Reviewed 500 Pull Requests—Here's What Every Java Dev Gets...

From Optional disasters to stream overkill, these are the Java sins haunting your code...

6d ago 131 6

Full Stack With Ram

Top 20 Java 8 Streams Coding Interview Questions Every...

Java 8 brought a paradigm shift to how we write code, and one of its most powerful...

Jul 3 1

In Java Programming by Pudari Madhavi

12 Multithreading Java Interview Questions

Break down complex Java multithreading questions into simple, interview-ready...

Jun 2 111 2

In Coding Odyssey by Shivam Srivastava

Qualcomm Java Interview Experience—2

Interview of Professional with 6+ Years of Experience

5d ago 13

Prince kumar Maurya

🧠 My Infosys Interview Experience (2025)—Full Stack...

Recently, I appeared for a Full Stack Java Developer interview at Infosys, and I'm...

Jul 5 3

Ajit Gupta

Java Developer Interview

Hey everyone!

May 20

See more recommendations