Springboot microservices guide

Below is **Set C (Questions 51–75) — Full Long-Form Expanded Answers**, continuing your 100-question Spring Boot Microservices Interview Guide.

✅ SET C — ADVANCED MICROSERVICES (Messaging, Data, Scalability, Transactions, API Patterns)

Questions 51–75 — FULL EXPANDED LONG ANSWERS

---

51. How do microservices communicate with each other? Explain sync vs async with real scenarios.

Microservices communicate in two major patterns:

1️⃣ Synchronous Communication (Request-Response)

This method requires **both services to be online at the same time**.

Common Protocols

- REST (HTTP)
- gRPC
- GraphQL

Pros

- Simple to understand
- Easy tracing
- Works well for real-time UI requests

Cons

- Tight coupling
- Failure cascades (if service B is down → service A also fails)
- Harder scalability

✔️ Real E-commerce Example

Checkout → Payment Service

```rust
Checkout Service → (HTTP Sync Call) → Payment Service
```

The user expects an immediate response → Payment must respond quickly.

2️⃣ Asynchronous Communication (Event-Driven)

Services communicate using **events**, messages, and queues.

Common Tools

- Kafka
- RabbitMQ
- AWS SNS/SQS
- Azure Service Bus

Pros

- Loose coupling
- Natural scalability
- No dependency on service uptime
- Great for heavy workloads

Cons

- Eventual consistency
- Harder debugging
- Message duplication handling needed

✔️ E-commerce Example

Order Placed → Order Service publishes event: `ORDER_CREATED`

Other services act independently:

```css
Order Service → Kafka → Payment Service
Order Service → Kafka → Inventory Service
Order Service → Kafka → Notification Service
```

User does not wait for these tasks; UI remains fast.

Summary Table

Feature	Sync	Async
Dependency	High	Low
Coupling	Tight	Loose
Failure impact	High	Low
Real-time	Yes	No
Use cases	Payment, UI calls	Inventory updates, notifications

---

52. What is Idempotency? Why is it critical in microservices?

✔️ Definition

Idempotency means **performing the same action multiple times produces the same effect as performing it once.**

✔️ Why Important?

Due to:

- retries
- message duplication
- network failure
- Kafka reprocessing

Microservices must avoid *double processing*.

Real-world example in e-commerce

Scenario: Payment API receives same request twice

原因:

- user double-clicks Pay
- gateway retries
- network timeout

Bad outcome:

❌ customer gets charged twice
❌ order created twice

Good implementation:

Add **Idempotency-Key** header.

```bash
POST /payments
Idempotency-Key: TXN_89348ASD
```

Payment service stores this key in DB → rejects duplicate.

Idempotent Methods

- GET → always safe
- PUT → replaces resource
- DELETE → repeated delete is still okay

---

53. Explain Eventual Consistency with real microservices examples.

✔️ Definition

In distributed systems, data may not be immediately consistent across all services.
But **eventually**, after some time, all systems reflect the correct state.

Real E-commerce Example

Step 1 — User Places Order

Order service marks: `OrderStatus = CREATED`

Step 2 — Inventory service receives event 2 seconds later

Updates stock: `Stock = Stock – 1`

Step 3 — Payment confirms after 4 seconds

Order updated: `OrderStatus = PAID`

❗ During these 4 seconds:

UI might temporarily show:
“Order pending inventory update”.

This is **eventual consistency**.

Why it is acceptable?

- Microservices run independently
- Systems prioritize availability over strict consistency
- User experience does not require immediate full consistency

---

---

54. What is Circuit Breaker Pattern? Why is it used? Implementation with Resilience4j?

Circuit breaker prevents cascading failures when a downstream service is failing.

Three States

1️⃣ Closed (Normal)

All requests allowed.
Failure rate monitored.

2️⃣ Open (Failure Detected)

Requests blocked for cool-down time.
Fallback logic used.

3️⃣ Half-Open

Allows limited trial requests to check if downstream recovered.

✔️ Real Example

Inventory service is down.
Order service calls keep failing.

Without circuit breaker:
❌ threads exhausted
❌ service A becomes slow
❌ cascading failure occurs

With circuit breaker:
✔️ fails fast
✔️ saves CPU
✔️ improves reliability

Resilience4j Example

```java
@CircuitBreaker(name="inventoryService", fallbackMethod="fallback")
public InventoryResponse checkStock(Long productId) {
   return restTemplate.getForObject(URL, InventoryResponse.class);
}

public InventoryResponse fallback(Long productId, Exception e){
   return new InventoryResponse(productId, false);
}
```

---

55. Explain Saga Pattern. Orchestration vs Choreography?

Saga solves **distributed transactions** in microservices.

Why needed?

SQL transactions cannot span across multiple microservices.

Types of Sagas

1️⃣ Choreography (Event-based)

- No central controller
- Services react to events

✔️ Good: simple, scalable
❌ Bad: hard to debug

Example: Order Workflow

```nginx
OrderCreated → Payment Service
PaymentCompleted → Inventory Service
InventoryReserved → Shipping Service
```

Each service listens to events.

2️⃣ Orchestration

- Central controller decides the workflow
- Clear control flow

✔️ Good: easy to manage
❌ Bad: central dependency

Example:

```nginx
```

```
Orchestrator → Payment Service
Orchestrator → Inventory
Orchestrator → Shipping
```

56. What is CQRS? How used in microservices?

CQRS = Command Query Responsibility Separation

- Read and write models are separated.

Benefits

✔️ Better scalability
✔️ Faster reads
✔️ Domain clarity
✔️ Works well with Event Sourcing

E-commerce Example

Commands:

- placeOrder
- cancelOrder
- updatePrice

Queries:

- getOrderHistory
- getOrderDetails
- listProducts

The read side uses denormalized data for fast UI responses.

57. What is Event Sourcing? How does it differ from state-based persistence?

✔️ Definition

Data is stored as **a sequence of events**, not as the final state.

State-Based (Normal DB)

You store only the latest state.

Example:

Order table:

```makefile
OrderId: 101
Status: SHIPPED
```

You lose history.

Event Sourcing Example

Events stored:

```markdown
1. ORDER_CREATED
2. PAYMENT_CONFIRMED
3. INVENTORY_RESERVED
4. ORDER_SHIPPED
```

Order state is reconstructed from events.

Pros

✔️ full audit

✔️ timeline replay

✔️ precise debugging
✔️ integrates with Kafka

Cons

❌ complex
❌ requires replay logic
❌ eventual consistency

---

58. What is API Gateway? Why do all microservices architectures use it?

API Gateway acts as **single entry point** for all clients.

Functions

- routing
- load balancing
- authentication
- rate limiting
- headers injection
- circuit breaker
- centralized logging

Real Example

Instead of:

```bash
/product-service
/order-service
/payment-service
/inventory-service
```

UI calls only:

```bash
/api/**
```

Gateway routes internally.

Popular Tools

- Spring Cloud Gateway
- Kong
- NGINX
- AWS API Gateway
- Azure APIM

---

59. Difference between API Gateway and Load Balancer?

API Gateway	Load Balancer
Application layer	Transport layer
Routes to specific services	Routes to identical nodes
Auth, rate limit, filters	Distributes traffic
used for microservices	used for scaling

---

60. What is Distributed Tracing? How implemented?

Distributed tracing tracks a request across multiple microservices.

Tools

- Zipkin
- Jaeger
- OpenTelemetry
- Sleuth

How it Works

Each request carries:

- TraceId
- SpanId

Services log these IDs → tracing tool shows complete flow.

Real Example

User places order:

The trace shows:

```nginx
OrderService → PaymentService → InventoryService → NotificationService
```

Helps in:
✔️ debugging
✔️ performance analysis
✔️ dependency mapping

---

61. What is Backpressure? How to handle in microservices?

When producer sends more load than consumer can process → backpressure occurs.

Solutions

- rate limiting
- buffering
- message throttling
- Kafka consumer groups scaling
- reactive programming (Project Reactor)

---

62. How do Kafka Consumer Groups scale microservices?

Consumer group = set of consumers reading from a topic.

Partition assignment:

```sql
1 partition → 1 consumer
```

So, with 10 partitions → you can scale to 10 parallel consumers.

Real Example

Inventory update events are heavy.
Add 5 instances of Inventory service.
Kafka partitions distribute work across all.

---

63. Why do microservices prefer NoSQL databases?

Reasons:

- schema flexibility
- horizontal scaling
- distributed by design
- denormalized fast reads

Common DBs:

- MongoDB
- Cassandra
- DynamoDB

---

64. What is Sharding? How is it used in distributed systems?

Sharding = splitting large dataset into smaller, independent chunks.

Example

User table:

```bash
Shard1 → users 1 to 1M
Shard2 → users 1M to 2M
```

Each shard on different node.

Benefits

✔️ horizontal scaling
✔️ reduced read/write latency
✔️ isolation

---

65. What is Cache Aside Pattern? Example with Redis.

Cache Aside:

1. check cache
2. if MISS → load from DB
3. store in cache

4. return response

Code Example

```sql
GET product/101
```

1. check Redis
2. if not exists → query MySQL
3. store in Redis
4. send response

---

66. Difference between Redis and Kafka?

Redis	Kafka
in-memory	disk-based distributed log
low latency	high throughput
cache	data streaming
short lived	persistent

---

67. How to handle database per service pattern? How services communicate?

Best practice:

Each microservice owns its DB.

Communication via:

- REST
- Kafka events
- Async messages

Example

Order does not read directly from Inventory DB.

It listens to `INVENTORY_UPDATED` events.

---

68. What is Two-Phase Commit? Why is it not recommended for microservices?

Distributed transaction algorithm:

1. Prepare phase
2. Commit phase

❌ Problems

- blocking
- coordinator single point of failure
- locks data across services
- slow
- kills performance

Thus → microservices prefer **Saga pattern**.

---

69. How do you secure asynchronous messages?

Secure messaging:

- SSL/TLS for Kafka brokers
- ACLs
- Kerberos
- SASL

- API authentication on producers

---

70. What is Fan-out/Fan-in Pattern?

Fan-out:

One event triggers multiple parallel tasks.

Example:
OrderPlaced event fans out to:

- Payment
- Inventory
- Notification

Fan-in:

Combine results.

Example:
Search results aggregated from:

- catalog service
- pricing service
- review service

---

71. How to prevent message duplication in Kafka consumers?

Techniques:

- idempotent consumer logic
- store processed offsets in DB
- enable exactly-once semantics (EOS)
- use `Idempotency-Key`

---

72. Explain exactly-once messaging in Kafka.

Kafka supports:
✔️ exactly once **delivery to topic**
✔️ exactly once **processing via transactions**
✔️ commit offset + write output atomically

---

73. How to handle schema evolution? (Backward/Forward compatibility)

Use **Schema Registry** (Avro/Protobuf).

Types:

- backward → old consumers can read new messages
- forward → new consumers read old messages
- full compatibility

---

74. What is Bulkhead Pattern?

Isolation technique to limit failure impact.

Example:

Separate thread pools:

- inventory calls
- payment calls
- shipping calls

If payment pool gets exhausted → inventory still works.

---

75. How do you scale microservices? All strategies explained.

Scaling techniques:

1. **Horizontal scaling** (add more instances)
2. **Auto-scaling** (CPU/memory triggers)
3. **Database sharding**
4. **Kafka partition scaling**
5. **Split monolith domains further**
6. **Caching with Redis**
7. **Async processing**
8. **Optimize JVM heap and GC**
9. **API Gateway load balancing**

✅ SET C COMPLETED Successfully (51–75)

If you want, I can generate:

✔️ Set D (76–100: Security, Logging, Monitoring, DevOps, Deployment)

✔️ Final Combined PDF of all 100 long-answer questions

✔️ A printable book-style layout with chapters & diagrams

Just say **Continue Set D** or **Generate PDF for all 100 Qs**.