Java Backend Interview Q&A

1. Your API works perfectly locally but becomes slow only in production. What would you check first?

First check environment differences: network latency, DB connection pool size, JVM GC, external service latency, thread pool saturation, logging level.
Example: Use Actuator metrics to compare latency.
```java
@RestController
public class HealthController {
    @GetMapping("/metrics")
    public ResponseEntity<Map<String, Object>> metrics() {
        Map<String, Object> m = new HashMap<>();
        m.put("timestamp", Instant.now());
        m.put("activeThreads", Thread.activeCount());
        return ResponseEntity.ok(m);
    }
}
```

2. Kafka consumers are running normally, but message lag keeps increasing. Why can this happen?

Common causes: consumer processing slower than producer rate, single partition bottleneck, consumer rebalance, GC pauses.
Example: Consumer with proper commit.
```java
@KafkaListener(topics = "orders", groupId = "order-group")
public void consume(String message) {
    try {
        process(message); // time-consuming
        ack.acknowledge(); // manual commit
    } catch (Exception e) {
        // handle and send to DLQ
    }
}
```

3. Database connections suddenly get exhausted during peak traffic. What could cause this?

Leaked connections, pool size too small, long-running transactions, connection not closed in finally block.
Example: Proper try-with-resources.
```java
try (Connection conn = dataSource.getConnection();
     PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?")) {
    stmt.setLong(1, id);
    ResultSet rs = stmt.executeQuery();
} // auto-close
```

4. Autoscaling creates more pods, but response time still keeps increasing.

Root cause often downstream dependency or shared resource like DB. Adding pods doesn't help if DB is bottleneck.
Check DB CPU, lock contention, connection pool.

5. Retry logic starts creating duplicate payment transactions during failures.

Idempotency key missing. Use idempotency token in DB or Redis.
```java
@Transactional
public PaymentResult processPayment(PaymentRequest req) {
    if (paymentRepository.existsByIdempotencyKey(req.getIdempotencyKey())) {
        return paymentRepository.findByIdempotencyKey(req.getIdempotencyKey());
    }
    Payment p = new Payment(req);
    return paymentRepository.save(p);
}
```

6. A scheduled job suddenly starts executing multiple times after scaling.

Without distributed lock, each pod runs job. Use ShedLock or Quartz with DB lock.
```java
@Scheduled(cron = "0 0 2 * * *")
@SchedulerLock(name = "dailyReport", lockAtMostFor = "10m")
public void generateReport() { ... }
```

7. One slow downstream service starts affecting the entire platform.

Thread pool exhaustion or blocking calls. Use circuit breaker + timeout + bulkhead.
```java
@CircuitBreaker(name = "paymentService", fallbackMethod = "fallback")
public Mono<Payment> callPayment() {
                                                                                return
webClient.get().uri("/pay").timeout(Duration.ofSeconds(2)).retrieve().bodyToMono(Payment.class);
}
public Mono<Payment> fallback(Throwable t) { return Mono.just(Payment.EMPTY); }
```

8. APIs randomly return 500 errors, but infrastructure looks healthy.

Check for transient issues: DB connection timeout, thread starvation, GC pause, rate limiting, exception in filter.
Enable stacktrace logging in dev env.

9. Health checks pass, but users still face failures.

Health check may only check container alive, not dependency. Use readiness probe for dependencies.
```yaml
readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: 8080
```

```

## 10. Cache improves performance initially, but later starts returning stale data.

TTL or cache invalidation missing. Use cache-aside pattern with TTL and event-driven invalidation.
```java
@Cacheable(value = "user", key = "#id", unless = "#result == null")
public User getUser(Long id) { return userRepo.findById(id).orElse(null); }

@CacheEvict(value = "user", key = "#user.id")
public void updateUser(User user) { userRepo.save(user); }
```

11. Logs exist everywhere, but debugging across services is still difficult.

Missing distributed tracing. Use OpenTelemetry + traceId propagation.
```java
@GetMapping("/order")
public Order getOrder(@RequestHeader("traceparent") String traceId) {
    log.info("Processing order with traceId={}", traceId);
    return orderService.get(traceId);
}
```

12. JVM memory usage slowly increases after every deployment.

Memory leak: static collections, ThreadLocal not cleared, non-closed resources.
Use heap dump and MAT to analyze.

13. APIs work in staging but fail behind the production gateway.

Gateway timeout, CORS, header size limit, TLS version, IP allowlist. Check gateway logs and timeout config.

14. Thread pools become exhausted even though CPU usage is stable.

Blocking IO in thread pool. Use async/reactive or dedicated thread pool for blocking tasks.
```java
ExecutorService blockingPool = Executors.newFixedThreadPool(50, new VirtualThreadFactory());
CompletableFuture.runAsync(() -> blockingCall(), blockingPool);
```

15. Circuit breakers are configured, but cascading failures still happen.

Circuit breaker threshold too high, fallback not implemented, or bulkhead missing. Combine with timeout and retry with backoff.
```java
@Retry(name = "payment", fallbackMethod = "fallback")
@TimeLimiter(name = "payment")
public CompletableFuture<Payment> callPayment() { ... }
```
```