# Java Backend Interview Q&A; — Production Debugging

*Real-world distributed systems questions with code examples*

*The shift in backend interviews: not “Can you explain annotations?” but “Can you debug distributed systems under production load?”*

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

**What to check first (in order):**

1. **Latency breakdown** — measure where time is spent: app, DB, downstream, network. Locally everything is on `localhost` (sub-millisecond); in prod each hop adds latency.

2. **N+1 queries** — small datasets locally hide them; prod data exposes them instantly.

3. **Connection pools (HikariCP, HTTP client, Redis)** — defaults are often too small for prod concurrency.

4. **GC pauses** — heap sized differently in prod; check `-Xlog:gc*` and pause times.

5. **TLS / DNS / proxy hops** — gateway, service mesh, sidecars add hidden overhead.

6. **Noisy neighbor / cold caches** — JIT not warm, OS page cache empty after deploy.

**Example: instrument with Micrometer to find the slow hop.**

```
@RestController
@RequiredArgsConstructor
public class OrderController {

    private final MeterRegistry registry;
    private final OrderService service;

    @GetMapping("/orders/{id}")
    public Order get(@PathVariable Long id) {
        Timer.Sample sample = Timer.start(registry);
        try {
            return service.findById(id);
        } finally {
            // Tag by outcome so prod dashboards split fast vs slow paths
            sample.stop(registry.timer("orders.get", "id", String.valueOf(id % 10)));
        }
    }
}
```

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

**Common root causes:**

- **Slow processing per record** — a downstream call, DB write, or heavy deserialization makes each `poll()` cycle longer than the production rate.
- **Too few partitions** — consumer parallelism is capped by partition count. 1 partition = 1 consumer thread, regardless of how many instances you run.
- **`max.poll.records` too high** combined with slow handler — broker thinks consumer is dead, triggers rebalance, lag grows.
- **Synchronous external calls** inside the consumer loop without batching.
- **Single-threaded listener** when records are independent and could be processed in parallel.

**Fix: bounded parallel processing inside the listener.**

```
@KafkaListener(topics = "payments", concurrency = "6")
public void consume(List<ConsumerRecord<String, Payment>> batch, Acknowledgment ack) {
    // Process batch in parallel but bounded — don't unbounded-fan-out to a thread pool
    List<CompletableFuture<Void>> futures = batch.stream()
        .map(rec -> CompletableFuture.runAsync(() -> handle(rec.value()), workerPool))
        .toList();

    CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();
    ack.acknowledge(); // commit only after all succeed; otherwise replay
}

private void handle(Payment p) {
    // keep this idempotent — see duplicate-payment Q below
    paymentService.process(p);
}
```

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

**Likely causes:**

- **Pool size too small** for peak concurrency (default Hikari = 10).
- **Long-running transactions** holding connections (e.g., `@Transactional` wrapping an external HTTP call).
- **Connection leaks** — code paths that don't return connections (missing try-with-resources, manual `getConnection()`).
- **Idle-in-transaction** sessions from a misconfigured ORM.
- **Cascade from a slow query** — every request queues for a connection, pool saturates.

**Fixes:** size pool from `threads × (1 + wait/compute)`, never call remote services inside a transaction, enable Hikari leak detection.

```
@Configuration
public class DataSourceConfig {

    @Bean
    public HikariDataSource dataSource() {
        HikariConfig cfg = new HikariConfig();
```

```
        cfg.setJdbcUrl(System.getenv("DB_URL"));
        cfg.setMaximumPoolSize(30);          // tune from real metrics, not guess
        cfg.setMinimumIdle(10);
        cfg.setConnectionTimeout(2_000);     // fail fast instead of queuing forever
        cfg.setLeakDetectionThreshold(10_000); // log stack trace if held > 10s
        cfg.setMaxLifetime(1_800_000);
        return new HikariDataSource(cfg);
    }
}

// ■ BAD: holds a DB connection for the entire HTTP round-trip
@Transactional
public void badCharge(Long id) {
    Order o = repo.findById(id).orElseThrow();
    paymentGateway.charge(o); // 2s remote call — connection held the whole time
    o.markPaid();
}

// ■ GOOD: do remote work outside the transaction
public void goodCharge(Long id) {
    Order o = readOnly(() -> repo.findById(id).orElseThrow());
    var receipt = paymentGateway.charge(o);            // no DB connection held
    txTemplate.executeWithoutResult(s -> repo.markPaid(id, receipt));
}
```

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

**The bottleneck is downstream, not the app.** More pods just multiply the pressure on the shared resource.

- **DB / cache / queue saturated** — adding pods adds connections, not capacity.
- **External API rate limit** — more pods = more 429s.
- **Shared lock / single-writer pattern** somewhere in the path.
- **Cold start** dominates for short-lived pods.
- **Scaling on the wrong metric** (CPU) when the real bottleneck is latency or queue depth.

**Fix: scale on a saturation signal AND protect the bottleneck with a bulkhead.**

```
// Resilience4j bulkhead — cap concurrent calls into a shared dependency,
// so adding pods cannot multiply pressure beyond what the DB can handle.
@Bean
public Bulkhead dbBulkhead() {
    return Bulkhead.of("db", BulkheadConfig.custom()
        .maxConcurrentCalls(20)
        .maxWaitDuration(Duration.ofMillis(50))
        .build());
}

@Service
@RequiredArgsConstructor
public class ReportService {
    private final Bulkhead dbBulkhead;
    private final ReportRepo repo;
```

```
    public Report build(Long id) {
        return Bulkhead.decorateSupplier(dbBulkhead, () -> repo.heavyQuery(id)).get();
    }
}
```

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

**Retries without idempotency = duplicates.** A timeout doesn't mean the call didn't happen — only that you didn't get the response.

**Fix:**

- Use an **idempotency key** per logical operation; the payment provider deduplicates.
- Persist the key + state **before** calling the provider.
- Retry only on idempotent failures (network/timeout), not on `200` + business error.

```
@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepo repo;
    private final StripeClient stripe;

    @Retryable(retryFor = IOException.class, maxAttempts = 3,
               backoff = @Backoff(delay = 200, multiplier = 2))
    public Receipt charge(ChargeRequest req) {
        // 1. Reserve a stable idempotency key tied to the business operation
        String key = "charge:" + req.orderId();
        PaymentRecord rec = repo.findByKey(key).orElseGet(() ->
            repo.save(new PaymentRecord(key, req.orderId(), Status.PENDING)));

        if (rec.getStatus() == Status.SUCCEEDED) return rec.toReceipt(); // short-circuit

        // 2. Provider sees the same key on retry and returns the original charge
        Receipt r = stripe.charge(req, key);

        repo.markSucceeded(rec.getId(), r.id());
        return r;
    }
}
```

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

**Every replica runs its own `@Scheduled`.** With N pods you get N executions.

**Fixes:**

- **Distributed lock** (ShedLock, Redisson) — only one node executes per tick.
- **External scheduler** (Quartz with JDBC store, K8s `CronJob`, Temporal) — scheduling is centralized.
- **Leader election** (Spring Cloud, K8s lease).

```
// ShedLock: backed by JDBC / Redis / Mongo — only one pod runs the job per window
@Configuration
```

```java
@EnableSchedulerLock(defaultLockAtMostFor = "PT5M")
public class SchedulerConfig {
    @Bean
    public LockProvider lockProvider(DataSource ds) {
        return new JdbcTemplateLockProvider(ds);
    }
}

@Component
public class ReportJob {
    @Scheduled(cron = "0 0 * * * *")
    @SchedulerLock(name = "hourlyReport", lockAtLeastFor = "PT1M", lockAtMostFor = "PT4M")
    public void run() {
        // guaranteed single execution cluster-wide
    }
}
```

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

**Threads pile up waiting on the slow dependency, then every endpoint stalls** — classic cascading failure.

**Isolate it:**

- **Timeouts everywhere** (connect + read).
- **Circuit breaker** — fail fast when error rate spikes.
- **Bulkhead** — cap concurrent calls so the slow path can't drain the shared thread pool.
- **Fallback** — return cached or degraded response.

```java
@Service
@RequiredArgsConstructor
public class RecommendationService {

    private final RecommendationClient client;

    @CircuitBreaker(name = "reco", fallbackMethod = "fallback")
    @TimeLimiter(name = "reco")
    @Bulkhead(name = "reco", type = Bulkhead.Type.THREADPOOL)
    public CompletableFuture<List<Item>> recommend(Long userId) {
        return CompletableFuture.supplyAsync(() -> client.fetch(userId));
    }

    @SuppressWarnings("unused")
    private CompletableFuture<List<Item>> fallback(Long userId, Throwable t) {
        // degraded but available — better than a cascading outage
        return CompletableFuture.completedFuture(List.of());
    }
}
```

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

**"Healthy infra" usually means CPU/RAM are fine — but 500s come from application-level issues:**

- **Connection pool exhaustion** intermittently (see Q3).

- **Race conditions** on shared mutable state.
- **Unhandled edge cases** in deserialization (nulls, missing fields).
- **Downstream returning malformed payloads** at certain hours.
- **Thread-local leaks** after async boundaries.
- **Bad pod in the set** — one replica is misconfigured.

**First move: correlate the 500s** — by pod, endpoint, payload shape, time window.

```
// Global handler that logs *everything* needed to correlate later
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorBody> handle(Exception ex, HttpServletRequest req) {
        String traceId = MDC.get("traceId"); // set by your tracing filter
        log.error("Unhandled: method={} uri={} pod={} traceId={}",
            req.getMethod(), req.getRequestURI(), System.getenv("HOSTNAME"), traceId, ex);
        return ResponseEntity.status(500)
            .body(new ErrorBody("INTERNAL", traceId)); // give clients the trace id
    }

    record ErrorBody(String code, String traceId) {}
}
```

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

**The health check is too shallow** — it returns 200 as long as the process is alive, but doesn't verify the dependencies users actually need.

**Use two probes:**

- **Liveness** — only checks the process is not deadlocked. Failing restarts the pod.
- **Readiness** — checks DB, cache, downstream auth are reachable. Failing removes the pod from the load balancer but doesn't restart it.

Never reuse the same endpoint for both.

```
@Component
public class DbReadinessIndicator implements HealthIndicator {
    private final JdbcTemplate jdbc;
    public DbReadinessIndicator(JdbcTemplate jdbc) { this.jdbc = jdbc; }

    @Override
    public Health health() {
        try {
            jdbc.queryForObject("SELECT 1", Integer.class);
            return Health.up().build();
        } catch (Exception e) {
            // Pod leaves load balancer rotation; users stop seeing failures from this pod
            return Health.down(e).build();
        }
    }
}
```

```
// application.yml
// management.endpoint.health.probes.enabled: true
// management.health.livenessstate.enabled: true
// management.health.readinessstate.enabled: true
```

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

**Caching strategy mismatched to the write pattern:**

- **No TTL** + no invalidation → stale forever.
- **TTL only** → window of inconsistency tolerated but unbounded for hot keys updated mid-window.
- **Write-through / invalidate-on-write** missing in some code paths (e.g., a batch job updates DB directly).
- **Cache stampede** when the key expires under load → many DB reads, then re-cached with old value due to race.

**Fix: invalidate on every write path + short TTL + single-flight refresh.**

```java
@Service
@RequiredArgsConstructor
public class ProductService {

    private final ProductRepo repo;

    @Cacheable(value = "product", key = "#id")
    public Product get(Long id) {
        return repo.findById(id).orElseThrow();
    }

    // Every mutation path MUST evict — including admin tools and batch jobs
    @CacheEvict(value = "product", key = "#p.id")
    @Transactional
    public Product update(Product p) {
        return repo.save(p);
    }

    @CacheEvict(value = "product", allEntries = true)
    public void onBulkImport() { /* called from batch job */ }
}
```

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

**Missing correlation.** Each service logs independently; without a shared identifier you cannot reconstruct a single request's path.

**Fix:**

- **Distributed tracing** (OpenTelemetry / Micrometer Tracing) — propagate `traceId` + `spanId` across HTTP and Kafka.
- **MDC** — put `traceId`, `userId`, `tenantId` into every log line.

- **Structured logs** (JSON) — searchable in Loki/ELK/Datadog.

```
// Filter that ensures every request has a traceId in MDC and logs
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class TraceIdFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res,
                                    FilterChain chain) throws IOException, ServletExceptio
n {
        String traceId = Optional.ofNullable(req.getHeader("X-Trace-Id"))
            .orElse(UUID.randomUUID().toString());
        MDC.put("traceId", traceId);
        res.setHeader("X-Trace-Id", traceId);
        try {
            chain.doFilter(req, res);
        } finally {
            MDC.clear(); // critical with thread pools — avoid leaking into next request
        }
    }
}

// logback pattern: %d %-5level [%X{traceId}] %logger - %msg%n
```

## 12. JVM memory usage slowly increases after every deployment.

**Classic memory leak signatures:**

- **`static` collections** growing unbounded (caches without eviction).
- **`ThreadLocal`** not cleared in pooled threads.
- **Listeners/observers** registered but never deregistered.
- **Classloader leak** with hot reloads (old class versions retained).
- **Connection / stream** not closed.

**Process:** take a heap dump (`jmap -dump:live`), open with Eclipse MAT, look at the dominator tree.

```
// ■ unbounded static cache — heap grows forever
public class BadCache {
    private static final Map<String, byte[]> CACHE = new HashMap<>();
    public static void put(String k, byte[] v) { CACHE.put(k, v); }
}

// ■ bounded with Caffeine — evicts by size and time
public class GoodCache {
    private static final Cache<String, byte[]> CACHE = Caffeine.newBuilder()
        .maximumWeight(64L * 1024 * 1024)              // 64MB cap
        .weigher((String k, byte[] v) -> v.length)
        .expireAfterWrite(Duration.ofMinutes(10))
        .recordStats()
        .build();
}

// ThreadLocal in a pooled thread — always clear in finally
ThreadLocal<User> CURRENT = new ThreadLocal<>();
```

```
try {
    CURRENT.set(user);
    handle();
} finally {
    CURRENT.remove(); // otherwise the next request inherits this user
}
```

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

**The gateway changes the request shape.** Things that pass in staging (direct calls) fail in prod:

- **Header rewriting / stripping** (`Authorization`, `Host`, custom headers).
- **Body size limits** smaller than the app's.
- **Timeouts** stricter than the app's processing time.
- **TLS termination** — `X-Forwarded-Proto` not honored, redirect loops.
- **Path rewriting** — `/api/v1/x` becomes `/x`, breaking route matching.
- **WAF rules** blocking certain payloads.

**Debug by capturing what the app actually receives.**

```
// Log the exact request the gateway delivers, once, to diff against staging
@Component
public class GatewayDiagnosticsFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res,
                                    FilterChain chain) throws IOException, ServletExceptio
n {
        if ("true".equalsIgnoreCase(req.getHeader("X-Diag"))) {
            Collections.list(req.getHeaderNames()).forEach(h ->
                log.info("hdr {}={}", h, req.getHeader(h)));
            log.info("uri={} query={} proto={} forwarded={}",
                req.getRequestURI(), req.getQueryString(),
                req.getHeader("X-Forwarded-Proto"),
                req.getHeader("Forwarded"));
        }
        chain.doFilter(req, res);
    }
}
```

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

**Threads are blocked on I/O, not burning CPU.** Low CPU + high latency = blocked threads.

**Causes:**

- Synchronous calls to a slow downstream — threads wait, pool drains.
- Lock contention (a `synchronized` block around a hot resource).
- Blocking calls inside reactive code (`block()` on a Reactor pipeline).
- DB connection wait — threads queued for a connection.

**Fix:** separate thread pools per dependency, async / non-blocking I/O, timeouts on every blocking call.

```
// Dedicated, bounded pool per downstream — slow service can't drain the shared pool
@Configuration
public class Pools {

    @Bean("paymentsPool")
    public ExecutorService paymentsPool() {
        return new ThreadPoolExecutor(
            10, 10,
            60, TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(100),                  // bounded queue
            new ThreadFactoryBuilder().setNameFormat("payments-%d").build(),
            new ThreadPoolExecutor.AbortPolicy()            // fail fast — don't queue for
ever
        );
    }
}

@Service
@RequiredArgsConstructor
public class PaymentsFacade {
    @Qualifier("paymentsPool") private final ExecutorService pool;
    private final PaymentsClient client;

    public CompletableFuture<Receipt> charge(ChargeRequest r) {
        return CompletableFuture.supplyAsync(() -> client.charge(r), pool)
            .orTimeout(2, TimeUnit.SECONDS); // every blocking call gets a timeout
    }
}
```

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

**A circuit breaker alone is not enough.** Common reasons it fails to stop a cascade:

- **No timeout** — the breaker waits forever for the slow call, never opens.
- **Thresholds too lenient** — opens only after damage is done.
- **No bulkhead** — even when open, queued threads are already exhausted.
- **Shared breaker across very different operations** — one bad endpoint poisons the metrics for the healthy ones.
- **Missing fallback** — open breaker throws, caller cascades anyway.
- **Retries inside the protected call** amplify load and keep the breaker closed.

**Fix: timeout + breaker + bulkhead + fallback, per dependency, with retries OUTSIDE.**

```
// Full resiliency stack — order matters: bulkhead → timeLimiter → circuitBreaker → retr
y
@Service
@RequiredArgsConstructor
public class InventoryFacade {

    private final InventoryClient client;

    @Bulkhead(name = "inventory", type = Bulkhead.Type.THREADPOOL)
```

```
    @TimeLimiter(name = "inventory")          // hard deadline so breaker can see failures
    @CircuitBreaker(name = "inventory", fallbackMethod = "fallback")
    @Retry(name = "inventory")                // retries only when breaker is closed
    public CompletableFuture<Stock> check(Long sku) {
        return CompletableFuture.supplyAsync(() -> client.stock(sku));
    }

    @SuppressWarnings("unused")
    private CompletableFuture<Stock> fallback(Long sku, Throwable t) {
        return CompletableFuture.completedFuture(Stock.unknown(sku)); // degraded, never t
hrow
    }
}

// application.yml
// resilience4j.circuitbreaker.instances.inventory:
//   slidingWindowSize: 50
//   failureRateThreshold: 30
//   waitDurationInOpenState: 10s
//   permittedNumberOfCallsInHalfOpenState: 5
```