Java Backend Interview Q&A: Production Debugging Scenarios

Modern backend interviews are moving away from "Can you explain annotations?" and toward "Can you debug distributed systems under production load?" These notes are written in that style: each answer starts with what to check first, explains why the issue happens, and shows a Java-oriented fix or diagnostic pattern.

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

Check the production path first, not the controller code first. Local traffic usually skips API gateways, TLS termination, service mesh sidecars, production databases, real network latency, real data volume, production logging volume, and downstream rate limits.

Start with latency breakdowns:

- Gateway and load balancer latency.
- Application request duration.
- Database query time and connection wait time.
- Downstream HTTP call duration.
- Queue, cache, DNS, and TLS timings.
- Thread pool saturation and garbage collection pauses.

The key interview point is that "slow in production" is usually a difference in environment, data, traffic shape, or dependency behavior. Do not guess from average latency only. Compare p95 and p99 latency by endpoint and by dependency.

```
import io.micrometer.core.instrument.Timer;
import org.springframework.stereotype.Service;

@Service
class ProductClient {
    private final RemoteCatalogApi api;
    private final Timer catalogTimer;

    ProductClient(RemoteCatalogApi api, Timer catalogTimer) {
        this.api = api;
        this.catalogTimer = catalogTimer;
    }

    ProductDto fetchProduct(String sku) {
        return catalogTimer.record(() -> api.getProduct(sku));
    }
}
```

In an interview, a strong answer is: "I would first prove where the time is spent with tracing and metrics. If the app time is low but gateway time is high, I look at the gateway. If app time is high, I split it into database, downstream calls, connection pool wait, CPU, GC, and thread pool wait."

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

Lag increases when the producer writes faster than the consumer group can process and commit offsets. The consumers may look "up" while still being too slow. Common reasons include slow processing logic, small consumer concurrency, too few partitions, long downstream calls, commit failures, repeated rebalances, poison messages, or consumers blocked on a database.

Check:

- Input rate versus processing rate.
- Number of partitions versus active consumers.
- Consumer group rebalances.
- Time spent inside message handling.
- Error and retry behavior.
- Whether offsets are committed only after successful processing.

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.Acknowledgment;
import org.springframework.stereotype.Component;

@Component
class PaymentEventConsumer {
    private final PaymentProjectionService projectionService;

    PaymentEventConsumer(PaymentProjectionService projectionService) {
        this.projectionService = projectionService;
    }

    @KafkaListener(topics = "payment-events", groupId = "payment-projection")
    void consume(ConsumerRecord<String, PaymentEvent> record, Acknowledgment ack) {
        projectionService.updateReadModel(record.value());
        ack.acknowledge();
    }
}
```

The important detail is that adding more consumers does not help if the topic has fewer partitions than consumers, or if all consumers block on the same database table, lock, or downstream dependency. Scaling consumers only helps when there is parallelizable work and enough partitions.

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

Connection exhaustion means requests are waiting for a database connection and eventually timing out. The database may not be down. The application may simply be borrowing connections faster than it returns them.

Common causes:

- Slow queries during peak data volume.
- Missing indexes.

- Transactions kept open while calling another service.

- Connection leaks.

- Pool size too small for traffic.

- Pool size too large across many pods, overwhelming the database.

- Long-running reports using the same pool as user APIs.

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
class CheckoutService {
    private final OrderRepository orders;
    private final PaymentGateway paymentGateway;

    CheckoutService(OrderRepository orders, PaymentGateway paymentGateway) {
        this.orders = orders;
        this.paymentGateway = paymentGateway;
    }

    @Transactional
    void badCheckout(Order order) {
        orders.save(order);
        paymentGateway.charge(order.paymentRequest());    // Holds DB connection too long.
    }
}
```

A better design is to keep the database transaction short and move slow network work outside it.

```
void checkout(Order order) {
    Order saved = orders.save(order);
    paymentGateway.charge(saved.paymentRequest());
    orders.markPaymentRequested(saved.id());
}
```

In production, check pool metrics such as active connections, idle connections, pending acquisition, borrow timeout, query latency, and transaction duration.

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

Autoscaling application pods only helps if the application layer is the bottleneck. If the bottleneck is the database, a shared cache, a downstream service, a queue partition count, or a global lock, more pods can make the problem worse by increasing pressure on the shared dependency.

Check whether the new pods are actually receiving traffic and whether they are ready before being added to the load balancer. Also check cold start time, JVM warmup, connection pool multiplication, and whether each pod creates background work.

```
import java.util.concurrent.Semaphore;
import org.springframework.stereotype.Service;

@Service
class InventoryClient {
    private final Semaphore bulkhead =  new Semaphore( 50);
    private final InventoryApi api;

    InventoryClient(InventoryApi api) {
        this.api = api;
    }

    Inventory get(String sku)   throws InterruptedException {
        if (!bulkhead.tryAcquire()) {
            throw new TooManyRequestsException(  "Inventory dependency is saturated"  );
        }
        try {
            return api.getInventory(sku);
        } finally {
            bulkhead.release();
        }
    }
}
```

A good interview answer: "I would identify the saturated resource. More pods reduce CPU pressure only if CPU was the bottleneck. If p99 rises while pods increase, I check shared dependencies, connection pools, queue partitioning, gateway limits, and whether autoscaling is reacting too late."

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

Retries are dangerous around non-idempotent operations. A timeout does not mean the payment failed. It may mean the payment succeeded but the response was lost. If the client retries with a new transaction request, the provider may charge twice.

The fix is idempotency. Generate an idempotency key per business operation, store it, send it to the payment provider, and return the existing result when the same operation is retried.

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
class PaymentService {
    private final PaymentRepository payments;
    private final PaymentProvider provider;

    @Transactional
    PaymentResult charge(String orderId, Money amount) {
        return payments.findByOrderId(orderId)
                .map(Payment::result)
                .orElseGet(() -> createCharge(orderId, amount));
    }

    private PaymentResult createCharge(String orderId, Money amount) {
        String idempotencyKey =  "order-payment-" + orderId;
        PaymentResult result = provider.charge(amount, idempotencyKey);
        payments.save( new Payment(orderId, idempotencyKey, result));
        return result;
    }
}
```

In an interview, say: "I never blindly retry payment creation. I retry with the same idempotency key, use a unique constraint on the business operation, and reconcile unknown outcomes with the provider."

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

A scheduler inside each application instance will run once per instance. If you scale from one pod to six pods, the same scheduled method may execute six times unless it is coordinated.

Common fixes are:

- Move the job to a dedicated worker deployment with one replica.
- Use Kubernetes CronJob for the scheduled trigger.
- Use a distributed lock such as ShedLock backed by the database.
- Make the job idempotent so repeated execution is harmless.

```java
import net.javacrumbs.shedlock.spring.annotation.SchedulerLock;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
class InvoiceJob {
    private final InvoiceService invoiceService;

    InvoiceJob(InvoiceService invoiceService) {
        this.invoiceService = invoiceService;
    }

    @Scheduled(cron = "0 0/5 * * * *")
    @SchedulerLock(name = "invoiceJob", lockAtMostFor = "PT4M", lockAtLeastFor = "PT30S")
    void generateInvoices() {
        invoiceService.generatePendingInvoices();
    }
}
```

The deeper answer is that scheduling is a distributed systems problem after scaling. A local in-memory scheduler is not a cluster-wide coordinator.

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

This happens when request threads wait too long on the slow service. Eventually thread pools fill up, queues grow, timeouts stack, and unrelated endpoints suffer because they share the same resources.

Use isolation:

- Short timeouts.
- Circuit breakers.
- Bulkheads.
- Separate thread pools or connection pools per dependency.
- Fallbacks for non-critical features.
- Backpressure instead of unbounded waiting.

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import io.github.resilience4j.timelimiter.annotation.TimeLimiter;
import java.util.concurrent.CompletableFuture;
import org.springframework.stereotype.Service;

@Service
class RecommendationService {
    private final RecommendationClient client;

    RecommendationService(RecommendationClient client) {
        this.client = client;
    }

    @CircuitBreaker(name = "recommendations", fallbackMethod = "fallback")
    @TimeLimiter(name = "recommendations")
    CompletableFuture<List<Product>> recommendations(String userId) {
        return CompletableFuture.supplyAsync(() -> client.fetch(userId));
    }

    CompletableFuture<List<Product>> fallback(String userId, Throwable ex) {
        return CompletableFuture.completedFuture(List.of());
    }
}
```

The interview signal is understanding blast radius. A slow optional dependency should not consume all request threads for critical checkout, login, or payment flows.

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

Random 500s often come from application behavior that appears only for certain inputs, timing, data states, or concurrency patterns. Healthy infrastructure does not prove healthy business logic.

Check:

- Recent deployments and feature flags.
- Exception types and stack traces.
- Null handling for rare data.
- Race conditions.
- Serialization and deserialization errors.
- Timeouts hidden as generic 500 responses.
- Dependency client exceptions not mapped to proper status codes.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
class ApiExceptionHandler {
    @ExceptionHandler (OrderNotFoundException. class )
    @ResponseStatus (HttpStatus.NOT_FOUND)
    ErrorResponse notFound(OrderNotFoundException ex) {
        return new ErrorResponse( "ORDER_NOT_FOUND" , ex.getMessage());
    }

    @ExceptionHandler (IllegalArgumentException. class )
    @ResponseStatus (HttpStatus.BAD_REQUEST)
    ErrorResponse badRequest(IllegalArgumentException ex) {
        return new ErrorResponse( "BAD_REQUEST" , ex.getMessage());
    }
}
```

Good production APIs distinguish client errors, dependency errors, timeouts, and true server bugs. That makes random 500s much easier to isolate.

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

Basic health checks often prove only that the process is alive. They may not prove that users can log in, read data, write data, reach dependencies, or complete the main business flow.

Separate checks:

- Liveness: should the container be restarted?
- Readiness: should this pod receive traffic?
- Startup: is the app still booting?
- Synthetic checks: can a real user journey work?

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
class DatabaseReadinessIndicator  implements HealthIndicator {
    private final UserRepository users;

    DatabaseReadinessIndicator(UserRepository users) {
        this .users = users;
    }

    @Override
    public Health health() {
        return users.canRead()
                ? Health.up().build()
                : Health.down().withDetail(  "database" , "read failed" ).build();
    }
}
```

The caution: readiness checks should be meaningful but not so heavy that they create extra outages. For user-facing reliability, pair health checks with synthetic monitoring and business

metrics.

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

Caching creates a second copy of data. Staleness happens when updates do not invalidate or refresh the cache correctly, when TTLs are too long, or when multiple services update the same data through different paths.

Common strategies:

- Cache-aside with explicit eviction after writes.
- Short TTL for frequently changing data.
- Versioned cache keys.
- Event-driven invalidation.
- Write-through cache for simple ownership models.
- Avoid caching strongly consistent data unless the business allows it.

```
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
class ProductService {
    private final ProductRepository products;

    @Cacheable(cacheNames = "products", key = "#sku")
    Product getProduct(String sku) {
        return products.findBySku(sku).orElseThrow();
    }

    @CacheEvict(cacheNames = "products", key = "#sku")
    Product updatePrice(String sku, Money newPrice) {
        Product product = products.findBySku(sku).orElseThrow();
        product.changePrice(newPrice);
        return products.save(product);
    }
}
```

A mature answer mentions consistency requirements. "Stale" is not always a bug. Product recommendations may tolerate it. Account balance and payment status usually cannot.

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

Logs without correlation are isolated fragments. In microservices, one user request may pass through gateway, auth, order, payment, inventory, and notification services. Without a trace ID or correlation ID, it is hard to connect those log lines.

Use structured logs and propagate correlation IDs across HTTP, Kafka, and async work.

```
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.UUID;
import org.slf4j.MDC;
import org.springframework.web.filter.OncePerRequestFilter;

class CorrelationIdFilter  extends  OncePerRequestFilter {
    static  final String HEADER =  "X-Correlation-Id" ;

    @Override
    protected  void doFilterInternal(HttpServletRequest request,
                                     HttpServletResponse response,
                                     FilterChain chain)
             throws  ServletException, IOException {
        String id = request.getHeader(HEADER);
        if (id == null || id.isBlank()) {
            id = UUID.randomUUID().toString();
        }
        MDC.put( "correlationId" , id);
        response.setHeader(HEADER, id);
        try {
            chain.doFilter(request, response);
        }  finally {
            MDC.clear();
        }
    }
}
```

The best answer adds distributed tracing. Logs tell what happened in one service. Traces show the path and timing across services.

12. JVM memory usage slowly increases after every deployment.

A slow memory increase may be normal warmup, but after every deployment it can also indicate a leak, cache growth, classloader retention, unbounded queues, high-cardinality metrics, thread-local leaks, or static collections retaining old data.

Check:

- Heap after full GC.
- Old generation trend.
- Object histograms.
- Heap dump dominators.
- Cache sizes.
- Queue sizes.
- Number of live threads.
- Metaspace growth after redeploys.

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

class BadRequestCache {
    private final Map<String, UserProfile> cache =   new ConcurrentHashMap<>();

    UserProfile get(String userId) {
        return cache.computeIfAbsent(userId,   this::loadProfile);
    }

    private UserProfile loadProfile(String userId) {
        return new UserProfile(userId);
    }
}
```

The code above never evicts. Under real production traffic, unique keys keep growing. Prefer bounded caches with size limits and TTLs.

```
LoadingCache<String, UserProfile> profiles = Caffeine.newBuilder()
        .maximumSize( 100_000 )
        .expireAfterWrite(Duration.ofMinutes(   30))
        .build( this::loadProfile);
```

In interviews, avoid saying only "increase heap." First prove whether it is a leak, then find the retaining path.

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

Production gateways add behavior that staging may not match: path rewriting, authentication, authorization, headers, request body limits, rate limits, CORS rules, timeouts, TLS, WAF rules, and different base paths.

Check:

- Actual URL path received by the app.
- Required headers after gateway forwarding.
- Token issuer and audience.
- Request and response size limits.
- Gateway timeout versus application timeout.
- Whether the app trusts forwarded protocol and host headers.

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
class DebugHeaderController {
    @GetMapping ("/internal/debug/headers"  )
    ResponseEntity<Map<String, String>> headers(
             @RequestHeader  Map<String, String> headers) {
        return ResponseEntity.ok(headers);
    }
}
```

This kind of endpoint should be protected or temporary. The interview point is that the failing system may be the contract between gateway and app, not the endpoint logic itself.

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

Stable CPU with exhausted threads usually means threads are waiting, not computing. They may be blocked on database connections, downstream HTTP calls, locks, file I/O, synchronized sections, or queue operations.

Check thread dumps. Look for many threads in ‘WAITING‘, ‘TIMED_WAITING‘, or blocked states with similar stack traces. Also check executor queue size, active count, rejected task count, and dependency latency.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class BlockingExecutorProblem {
    private final ExecutorService executor = Executors.newFixedThreadPool(    20);
    private final SlowClient slowClient;

    void submit(String id) {
        executor.submit(() -> slowClient.call(id));    // Threads wait here under slowness.
    }
}
```

Fixes include bounded queues, short timeouts, separate pools per dependency, backpressure, and virtual threads for suitable blocking I/O workloads.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> slowClient.call(   "customer-123" ));
}
```

Virtual threads help with many blocking calls, but they do not remove the need for dependency timeouts, rate limits, and database capacity planning.

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

Circuit breakers are useful, but they are not a complete resilience strategy. Cascading failures can still happen if timeouts are too long, thread pools are shared, fallback logic is slow, retries amplify traffic, or the breaker opens only after the system is already saturated.

Check:

- Timeout is shorter than caller timeout.
- Retry count and backoff.
- Circuit breaker failure threshold and sliding window.
- Bulkhead isolation.
- Fallback cost.
- Whether all dependency calls go through the breaker.
- Whether async calls and scheduled jobs bypass resilience policies.

```java
import io.github.resilience4j.bulkhead.annotation.Bulkhead;
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import io.github.resilience4j.retry.annotation.Retry;
import io.github.resilience4j.timelimiter.annotation.TimeLimiter;
import java.util.concurrent.CompletableFuture;

class PricingFacade {
    private final PricingClient pricingClient;

    @TimeLimiter(name = "pricing")
    @CircuitBreaker(name = "pricing", fallbackMethod = "fallbackPrice")
    @Bulkhead(name = "pricing")
    @Retry(name = "pricing")
    CompletableFuture<Price> price(String sku) {
        return CompletableFuture.supplyAsync(() -> pricingClient.getPrice(sku));
    }

    CompletableFuture<Price> fallbackPrice(String sku, Throwable ex) {
        return CompletableFuture.completedFuture(Price.unavailable(sku));
    }
}
```

The strongest answer is: "A circuit breaker protects future calls after failures are detected. To stop cascading failure, I also need timeouts, bulkheads, bounded queues, controlled retries, graceful degradation, and observability."

Quick Interview Checklist

When a production backend issue appears, answer in this order:

1. Define the symptom using metrics: latency, errors, saturation, traffic, lag, or correctness.
2. Locate the boundary: gateway, app, database, cache, queue, downstream, JVM, or scheduler.
3. Compare p50, p95, and p99 behavior.
4. Check recent changes: deploys, config, traffic, data volume, feature flags, and dependency changes.
5. Protect the system: timeouts, backpressure, circuit breakers, bulkheads, idempotency, and rate limits.
6. Fix root cause and add monitoring so the same issue is visible earlier next time.

The interviewer is usually testing production judgment: can you isolate the bottleneck, reduce blast radius, protect data correctness, and explain tradeoffs clearly?