* "What’s the team’s current tech stack for trade processing?"
* "How does Luxoft approach CI/CD for financial systems?"

**Q1: How would you design a thread-safe caching system for high-frequency trading data?**

**Key Points:**

* ConcurrentHashMap + AtomicReference for thread safety.
* Avoid synchronized blocks for better throughput.

*public class* TradingCache {  
 *private final* ConcurrentHashMap<String, AtomicReference<MarketData>> cache = *new* ConcurrentHashMap<>();  
  
 *public void* update(String symbol, MarketData data) {  
 cache.compute(symbol, (k, v) -> {  
 *if* (v == *null*) v = *new* AtomicReference<>();  
 v.set(data);  
 *return* v;  
 });  
 }  
  
 *public* MarketData get(String symbol) {  
 *return* cache.getOrDefault(symbol, *new* AtomicReference<>()).get();  
 }  
}

**Q2: Explain how you’d optimize a slow SQL query on a 10M+ row**TRADES**table.**

*-- 1. Add indexes on filtered/sorted columns  
CREATE* INDEX idx\_trade\_date *ON* TRADES(trade\_date);  
  
*-- 2. Partition by date range  
ALTER TABLE* TRADES PARTITION *BY* RANGE (trade\_date) (  
 PARTITION p\_2023 *VALUES* LESS THAN ('2024-01-01')  
);  
  
*-- 3. Rewrite query to use covering index*EXPLAIN ANALYZE *SELECT* trade\_id *FROM* TRADES   
*WHERE* trade\_date > '2024-01-01' *AND* currency = 'USD';

**Q3: How would you secure a Capital Markets microservice?**

**Key Features:**

* OAuth2/JWT for authentication.
* Role-based access control (RBAC).
* @Configuration  
  @EnableWebSecurity  
  *public class* SecurityConfig {  
   @Bean  
   *SecurityFilterChain* securityFilterChain(HttpSecurity http) *throws* Exception {  
   http  
   .authorizeHttpRequests(auth -> auth  
   .requestMatchers("/api/trades/\*\*").hasRole("TRADER")  
   .anyRequest().authenticated()  
   )  
   .oauth2ResourceServer(oauth2 -> oauth2.jwt(*Customizer*.*withDefaults*()));  
   *return* http.build();  
   }  
  }

**Q4: Design a REST API for FX trade execution with idempotency.**

@PostMapping("/fx/trades")  
*public* ResponseEntity<Trade> executeTrade(  
 @RequestBody TradeRequest request,  
 @RequestHeader("Idempotency-Key") String idempotencyKey) {  
  
 *// 1. Check Redis for existing idempotencyKey  
 // 2. If not exists, process trade and store result in Redis  
 // 3. Return 409 Conflict if key exists*}

what is the issue if I don't use idempotency ?

**1. Duplicate Side Effects**

Without idempotency:

* Users may retry requests due to timeouts or errors.
* Each retry may create **duplicate records**, **multiple transactions**, or **multiple messages sent** — leading to incorrect state/actions.
* Example: Resubmitting a payment twice charges the customer twice.

**Complexity in Clients**

Without idempotency:

* Clients must **track requests**, detect duplicates, and add complex retry logic.
* With idempotency:
  + Clients generate a unique key (e.g. UUID).
  + Server caches the result on first request.
  + Subsequent retries with the same key return the **same outcome**, transparently and safely.

you can configure your **Spring Boot application** to **enforce idempotency for all POST requests by default**

**Option 1: Use an Interceptor or Filter + Idempotency Header**

You can create a **Spring HandlerInterceptor** or OncePerRequestFilter that:

1. Checks incoming POST requests for an Idempotency-Key header.
2. Constructs a namespaced key like idempotency:fx:<idempotencyKey>.
3. Checks if this key exists in Redis (or any distributed cache).
   * If it **exists** → retrieve and return the stored response.
   * If it **doesn't exist** → process the request, store the response under that key, and return it.

You can register this interceptor to apply to **all POST endpoints** in your WebMvcConfigurer.

This is exactly what libraries like

* **TransferWise’s idempotence4j** or
* **spring-idempotency-filter**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Authorization | Authenticates the user & grants access (Bearer <JWT>, Basic Auth, etc.) | Ensuring only traders can execute FX trades. | | Idempotency-Key | Ensures the same request isn’t processed twice (e.g., duplicate trades). | Preventing duplicate FX trade executions. | |

**Q5: How would you deploy a JBoss app to Kubernetes with zero downtime?**

**Dockerize JBoss**:

FROM jboss/wildfly:26.1.0

COPY target/app.war /opt/jboss/wildfly/standalone/deployments/

CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0"]

**K8s Rollout**:

spec:

strategy:

rollingUpdate:

maxUnavailable: 0

maxSurge: 1

readinessProbe:

httpGet:

path: /app/health

port: 8080

**Q6: How do you monitor microservices in production?**

**Tools:**

* **Metrics**: Prometheus + Grafana (JVM, HTTP latency).
* **Logs**: ELK Stack (Elasticsearch + Kibana).
* **Tracing**: Jaeger/Zipkin for distributed tracing.

**Q8: What’s the difference between Value Date and Trade Date?**

**A:**

* **Trade Date**: When the trade is executed.
* **Value Date**: When cash flows are settled (e.g., T+2 for FX).

**Q10: How would you debug a memory leak in production?**

jps # find <PID>

jmap -dump:live,format=b,file=heap.hprof <PID>

jvisualvm --openfile heap.hprof

1. Capture heap dump:

bash

jmap -dump:live,format=b,file=heap.hprof <PID>

1. Analyze with Eclipse MAT.
2. Look for retained objects in old-gen memory.

Java Flight Recorder (JFR) is a tool for collecting diagnostic and profiling data about a running Java application

|  |
| --- |
| * **Event Recording:**   JFR records a wide range of events happening within the JVM and the Java application, including thread activity, garbage collection, lock contention, and more.   * **Low Overhead:**   JFR is designed to have minimal impact on the performance of the application, making it suitable for production use.   * **Data Collection:**   JFR collects a continuous stream of data about the application's behavior, which can be analyzed later to identify performance bottlenecks or other issues.   * **Analysis with JMC:**   The data collected by JFR is typically analyzed using Java Mission Control (JMC), a separate tool that provides visualizations and tools for exploring the recorded data. |

|  |
| --- |
| **Trace & Fix in Code**  Once suspicious classes or structures are identified:   1. Find where they're referenced (e.g., static collections, caches, thread-locals). 2. Fix code to remove stale entries, null references, or unbounded growth. 3. Re-run the app and capture a new heap dump to verify the leak is resolved. |

|  |
| --- |
| **Tips & Best Practices**   * Use **live** flag to filter unreachable objects. * Heap analysis can require lots of memory; ensure your client (MAT/VisualVM) has adequate heap. * Monitor **GC logs**: repeated full GCs without memory reclamation indicate leaks. |