Response

1. Interview-Style Opening

"Distributed tracing is non-negotiable in microservices. Without it, debugging a failed request across five different services is like looking for a needle in five different haystacks. My approach is to ensure that every single log line—from the frontend ingress to the deepest backend database call—is tagged with a unique **Trace ID** so we can reconstruct the entire request lifecycle instantly."

2. Problem Understanding and Clarification

The core issue is **observability**. When a user reports "My order failed," and that order involves the Order Service, Inventory Service, and Payment Service, we cannot check three separate log files hoping timestamps match. We need a "thread" that ties them all together.

Assumptions & Goals:

- **Unique Identifier:** Every incoming request gets a generated UUID (Trace ID).
- **Propagation:** This ID must be passed in HTTP headers (e.g., `X-Correlation-ID` or `traceparent`) to downstream services.
- **Context Awareness:** The ID must be available in the logging context (MDC) so that every `logger.info()` call automatically includes it without developers manually adding it to the string.

3. High-Level Approach (The MDC Pattern)

I use the **MDC (Mapped Diagnostic Context)** pattern in Java, specifically with **Micrometer Tracing** (formerly Spring Cloud Sleuth) in Spring Boot 3.

1. **Ingress:** A filter intercepts the request at the Gateway or Service entry.
2. **Creation:** If a `traceId` header exists, use it; otherwise, generate a new one.
3. **Storage:** Put this ID into the **MDC** (a thread-local map provided by SLF4J).
4. **Logging:** Configure `logback.xml` to print `%X{traceId}` in every log line.
5. **Propagation:** When calling another service (via `RestTemplate` or `Feign`), an interceptor reads the ID from MDC and injects it into the outgoing HTTP headers.

4. Visual Explanation

Diagram: The Life of a Trace ID

```
User Request
    |
    v
[ API Gateway ]  <-- 1. Generates Trace ID: "abc-123"
    |                (Put in MDC)
    |
    |-- Log: "[abc-123] Gateway received request"
    |
    v (Header: X-Trace-Id: abc-123)
    |
[ Order Service ] <-- 2. Reads Header, puts "abc-123" in MDC
    |
    |-- Log: "[abc-123] Order validation started"
    |
    v (Header: X-Trace-Id: abc-123)
    |
[ Payment Service ] <-- 3. Reads Header
    |
    |-- Log: "[abc-123] Payment Failed!"  <-- ERROR FOUND
```

Explanation:
By searching for "abc-123" in our centralized logging tool (Splunk/ELK), I see the story chronologically across all three services.

5. Java Code (Production-Quality)

In Spring Boot 3, **Micrometer Tracing** handles most of this automatically. However, understanding the manual implementation via a `Filter` demonstrates deeper knowledge.

1. The Request Filter (The "Interceptor"):

```java
import jakarta.servlet.*;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.slf4j.MDC;
import org.springframework.stereotype.Component;
import java.io.IOException;
import java.util.UUID;

@Component
public class CorrelationIdFilter implements Filter {

    private static final String CORRELATION_ID_HEADER = "X-Correlation-Id";
    private static final String MDC_KEY = "traceId";

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain ch
            throws IOException, ServletException {
```

```
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        // 1. Extract or Generate ID
        String correlationId = req.getHeader(CORRELATION_ID_HEADER);
        if (correlationId == null || correlationId.isEmpty()) {
            correlationId = UUID.randomUUID().toString();
        }

        // 2. Put in MDC (Thread Local Storage)
        MDC.put(MDC_KEY, correlationId);

        // 3. Return ID to client for their reference
        res.addHeader(CORRELATION_ID_HEADER, correlationId);

        try {
            chain.doFilter(request, response);
        } finally {
            // 4. CLEANUP is mandatory to prevent memory leaks in thread pools
            MDC.remove(MDC_KEY);
        }
    }
}
```

2. The Log Configuration (`logback-spring.xml`):

```
<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- %X{traceId} extracts the value from MDC -->
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] [%X{traceId}] %-5level %logger{36}
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
    </root>
</configuration>
```

3. Propagating to Downstream Services (RestTemplate):

```
@Bean
public RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();

    // Add interceptor to inject header in outgoing requests
    restTemplate.getInterceptors().add((request, body, execution) -> {
        String traceId = MDC.get("traceId");
        if (traceId != null) {
            request.getHeaders().add("X-Correlation-Id", traceId);
        }
        return execution.execute(request, body);
    });
    return restTemplate;
}
```

6. Code Walkthrough (Line-by-Line)

- `MDC.put(MDC_KEY, correlationId)`: This is the magic line. It stores the ID in a map bound to the *current thread*. Any logger running on this thread can now access "traceId".
- `try...finally { MDC.remove() }`: This is critical. Tomcat/Jetty use thread pools. If you don't clear the MDC, the next request reusing this thread might inherit the *old* trace ID, causing confusing logs.
- `%X{traceId}` **in Logback**: This tells the logging framework: "Look into the MDC map, find the key `traceId`, and print it here."
- `restTemplate.getInterceptors()`: This ensures the chain isn't broken. Without this, Service B would generate a *new* ID, and we'd lose the link to Service A.

7. How to Explain This to the Interviewer

"Yes, I have implemented distributed logging using the **ELK Stack** (Elasticsearch, Logstash, Kibana).

In my last project, we used **Micrometer Tracing** (or Spring Cloud Sleuth). The moment a request hit our Gateway, we assigned it a Trace ID.

The key challenge was ensuring this ID propagated correctly across asynchronous boundaries, like when we used **Kafka**. We had to write a custom serializer/deserializer to pull the Trace ID from the MDC and inject it into the Kafka Record Headers.

This allowed us to search Kibana for one ID and see the entire flow:

- HTTP Request (Service A)
- Kafka Message Produced (Service A)
- Kafka Message Consumed (Service B)
- Database Update (Service B)

Without this propagation, async flows are black holes."

8. Edge Cases and Follow-Up Questions

Q: What happens with async threads (`@Async`)?
A: `MDC` is thread-local, so child threads don't inherit it by default. You need to use a `TaskDecorator` in Spring to copy the MDC context from the parent thread to the child thread.

Q: How do you handle high log volume?
*A: We don't log everything. We use **Sampling**. For successful requests, we might only log INFO*

level for 1% of traffic. For ERRORs, we log 100%.

Q: Trace ID vs Span ID?
*A: A **Trace ID** tracks the whole journey (Gateway → Service A → Service B). A **Span ID** tracks a specific hop (just Service A). A Trace has multiple Spans.*

9. Optimization and Trade-offs

- **Performance Overhead:** MDC operations are fast, but massive log ingestion (ELK) is expensive. We optimized by switching logs to **JSON format** directly. Instead of parsing text strings (Logstash grok patterns), we wrote JSON logs so Elasticsearch could index them immediately with less CPU usage.
- **Structured Logging:** Instead of `log.info("User " + user + " failed")`, we use Structured Logging arguments `log.info("User login failed", kv("userId", user))`. This makes the `userId` a queryable field in Datadog/Splunk, not just part of a text string.