perplexity

How you can enable distributed tracing in your microservice architecture.

To enable distributed tracing in a microservice architecture, you need to implement a mechanism that tags each request with a unique ID and propagates it across all service boundaries.

The standard industry approach uses **OpenTelemetry** (the modern standard) or **Micrometer Tracing** (for Spring Boot 3+), visualizing data in tools like **Zipkin**, **Jaeger**, or **Datadog**.

Step-by-Step Implementation (Spring Boot 3 Example)

In Spring Boot 3, Spring Cloud Sleuth has been replaced by **Micrometer Tracing**.

1. Add Dependencies

Add the following to your `pom.xml` (for both Producer and Consumer services):

```
<!-- Micrometer Tracing Bridge -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<!-- Zipkin Reporter (to send traces to Zipkin server) -->
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
<!-- Actuator (enables metrics/tracing) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Configuration (`application.yml`)

Enable tracing in your configuration file. You need to tell Spring where to send the traces and how many requests to sample.

```
management:
  tracing:
    sampling:
      probability: 1.0  # 1.0 = Trace 100% of requests (use 0.1 for 10% in Prod)
  zipkin:
```

```
  tracing:
    endpoint: "http://localhost:9411/api/v2/spans" # URL of your Zipkin server
```

3. Run Infrastructure (Zipkin Server)

You need a backend to store and view the traces. The easiest way is via Docker:

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

How It Works (The Flow)

1. **Trace ID Creation:** When a user calls Service A, the tracing library intercepts the request and generates a unique `Trace ID` (e.g., `abc-123`) and a `Span ID`.
2. **Context Propagation:** Service A calls Service B. The library automatically injects the `Trace ID` into the HTTP Headers (e.g., `traceparent` or `b3` headers).
3. **Trace Continuation:** Service B receives the request, extracts the `Trace ID` from the header, and uses it for its own logs. It creates a new `Span ID` (child span) linked to the same Trace ID.
4. **Reporting:** Both services asynchronously send their span data (start time, end time, metadata) to the Zipkin server.
5. **Visualization:** You open the Zipkin UI (`http://localhost:9411`) and search for the Trace ID. You see a waterfall graph showing exactly how long Service A took, how long the network call took, and how long Service B took.

Interview Talking Points

- **"Why do we need it?"** To debug latency issues and visualize the path of a request across distributed services. Without it, logs are isolated and hard to correlate.
- **"What about Log Correlation?"** Distributed tracing also adds the `Trace ID` and `Span ID` to your application logs (via MDC). This allows you to search Splunk/ELK for a specific Trace ID and see all logs from all services for that single transaction.
- **"Standards:"** Mention **OpenTelemetry** (OTel) as the vendor-neutral standard for generating and collecting traces, metrics, and logs. Spring Boot 3 uses Micrometer Tracing which supports OTel bridging.

⁂

1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg
2. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg
3. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg

4. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg