Observability with OpenTelemetry

A Complete Guide for Java/Spring Developers

Observability & OpenTelemetry Basics

Summary - Monitoring

- Monitoring is the process of collecting and analyzing predefined metrics (like CPU usage, response time…etc) to detect issues and raise alerts when something goes wrong.
- **Example of predefined checks / alerts**
  - CPU Usage > 90%
  - Response Time > 2s
- Monitoring tells us "***something is wrong***"

Summary - Observability

- Ability to understand the internal state of a system by analyzing the ***outputs / signals*** it produces.
- Observability helps us understand "***why***" something went wrong.

Summary - Telemetry Signals

- Logs
- Metrics
- Traces

Summary - OpenTelemetry (Otel)

- OpenTelemetry is an open-source, vendor-neutral observability framework that provides standard APIs, SDKs, and tools to generate, collect, and export telemetry data (logs, metrics, and traces).

Summary - Instrumentation

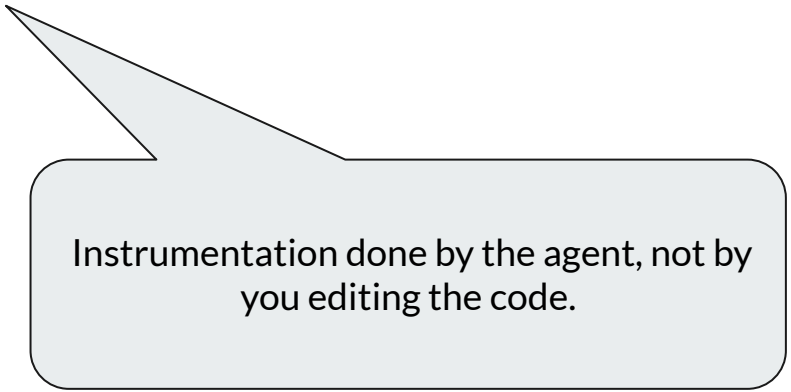
- Adding the necessary code / tooling to your application so that it emits telemetry data (logs, metrics, traces)
  - Automatic / Zero-code
  - Manual / Code-based

Distributed Tracing with OpenTelemetry

Automatic Instrumentation

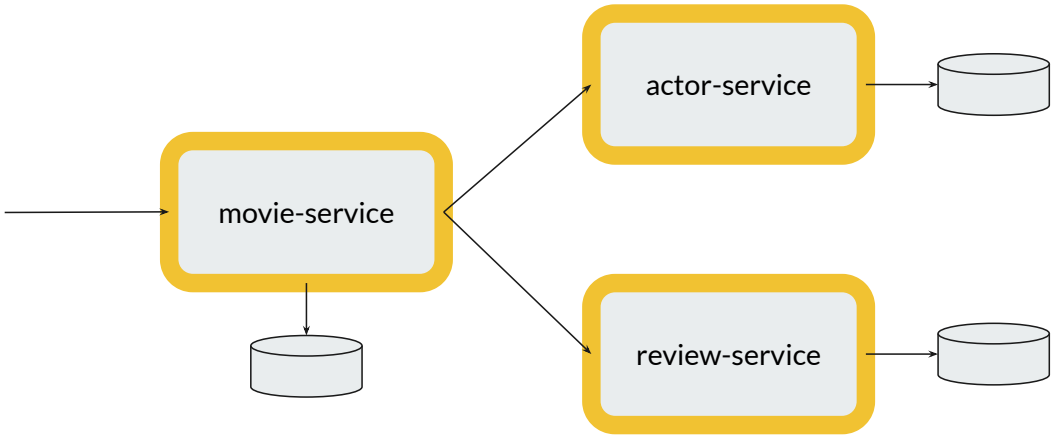
Summary - Automatic Instrumentation

- No code changes inside ***src/main/java***
- We attach an agent at run time.
  - This agent runs alongside our application and captures the request.
  - We configure the agent via environment variables / command line options - NOT by modifying source code.



Summary - OpenTelemetry Agent

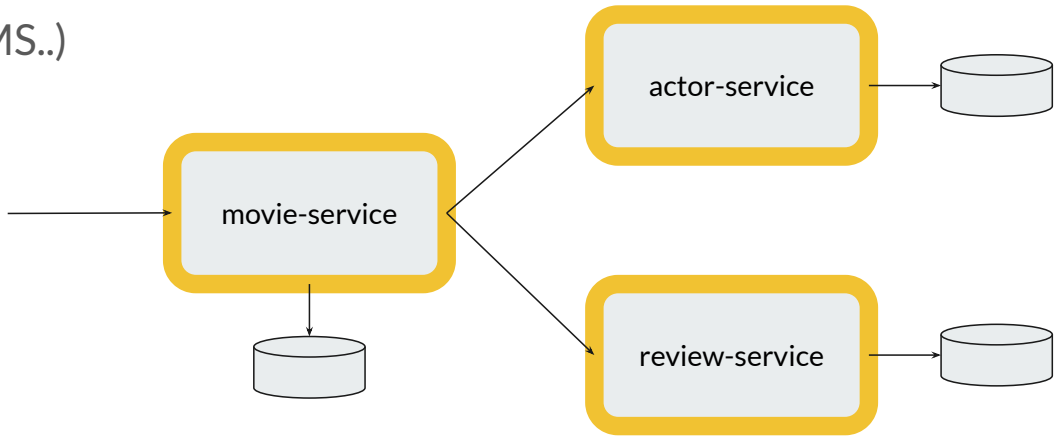
- standalone executable (jar file / a set of libs)
- OpenTelemetry provides this agent.



Summary - OpenTelemetry Agent

- ByteCode Manipulation
- Spring (Spring Boot, WebMVC, WebFlux)
- Servlet containers (Tomcat, Jetty, etc.)
- HTTP clients (RestClient, WebClient, HTTP Client…)
- Databases (JDBC, R2DBC …)
- Messaging (Kafka, RabbitMQ, JMS..)

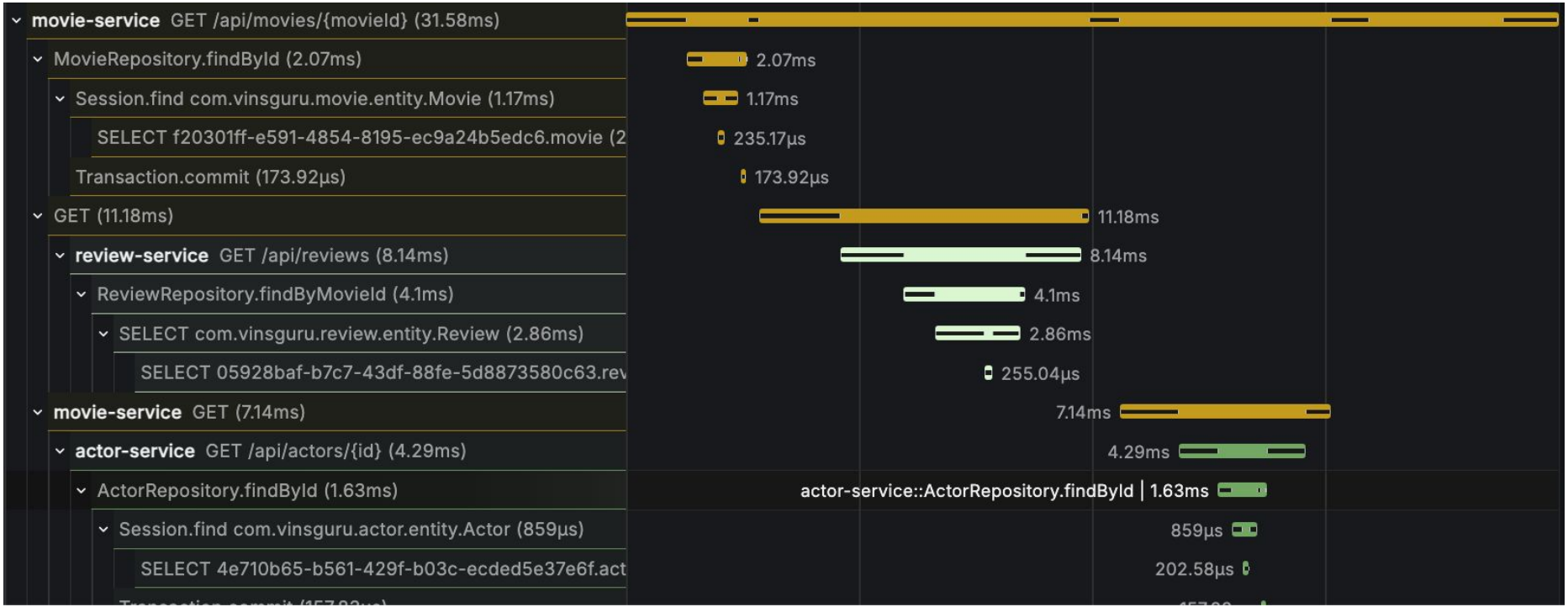
We can use Manual Instrumentation if your framework / lib is NOT supported!



Summary - Otel-Collector

- **Vendor-agnostic proxy** that receives, processes and exports telemetry data!
- A central hub for all the observability data
- It decouples our application from the destination of telemetry data.

Summary - Distributed Tracing



Summary: Parent Child Relationship

- **traceId**
- **Span**
  - **name** (GET /api/reviews, DB Query)
  - **spanId** (unique identifier for each span)
  - **traceId** (shared across all spans that belong to the same trace.)
  - **parentSpanId** (if any. For ex: DB query as part of incoming request)
  - **startTime**
  - **endTime**
  - **attributes**
  - …
  - …

Summary: Parent Child Relationship

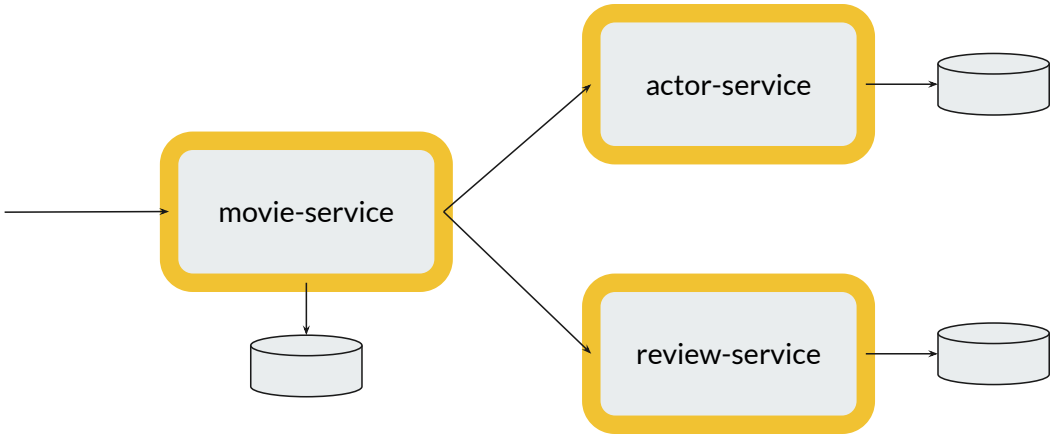
- traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01

version

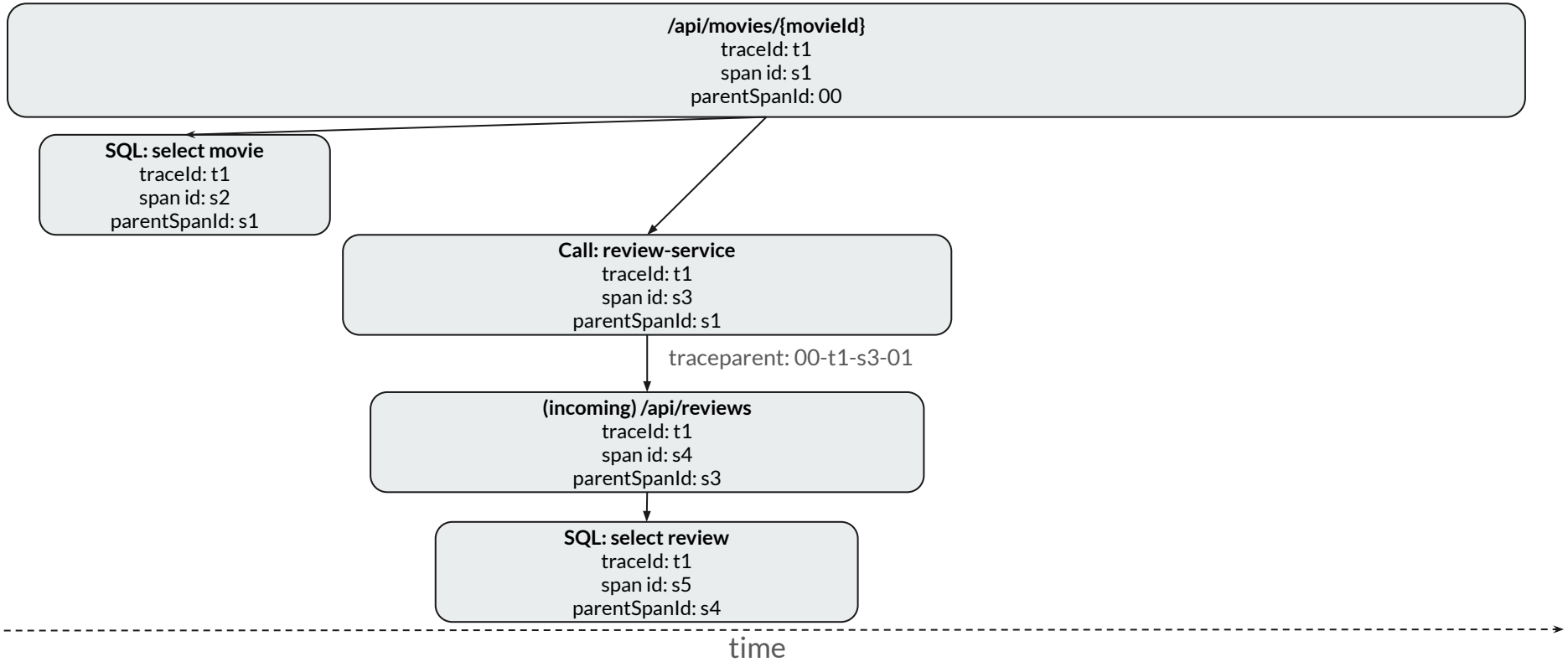
trace id

span id

trace flag

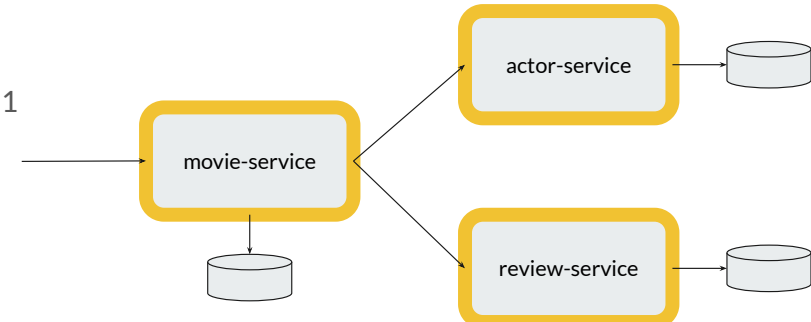


Summary: Parent Child Relationship



Summary - Attributes: Span vs Resource

- Resource Attributes
  - Belong to the **service/process** which emits telemetry data.
    - service.name: movie-service
    - container.id: 00394343434
    - aws.region: us-east-1
  - They remain constant for all the spans!
- Span Attributes
  - Belong to the specific span (operation)
    - http.method: GET
    - url.path: /api/movies/1
    - db.statement: SELECT * FROM reviews WHERE movieId=1



Sampling Strategies

Summary - Sampling

- Sampling is the process of deciding which traces (or spans) to record and export, and which ones to drop.
  - Control the volume of telemetry data
  - Reduce the storage and processing costs
  - Still capture enough traces to troubleshoot issues effectively.

Summary - Types Of Sampling

- Head Sampling
  - Always On
  - Always Off
  - Traceid Ratio
  - Parent Based
    - Always On
    - Always Off
    - Traceid Ratio
- Tail Sampling (Rule based sampling)
  - Status Code
  - Latency
  - Attribute

Metrics

Summary - Metrics

- Metrics are numerical measurements collected over time.
- They are lightweight & aggregated.
  - Cheap to store & query
- Big picture view

Traces → What happened with this request?

Metrics → How is the system doing overall?

Summary - Types Of Metrics

- Counter
  - Only goes up
    - number of requests processed
    - number of errors
- Gauge / UpDownCounter
  - Can go up and down
    - CPU usage
    - Available Inventory
- Histogram
  - Distribution of values
    - Request latency (requests completed in <10ms, <50ms, <100ms, etc.).
    - Size of uploaded files (<10MB, <5MB…)

Summary - Prometheus

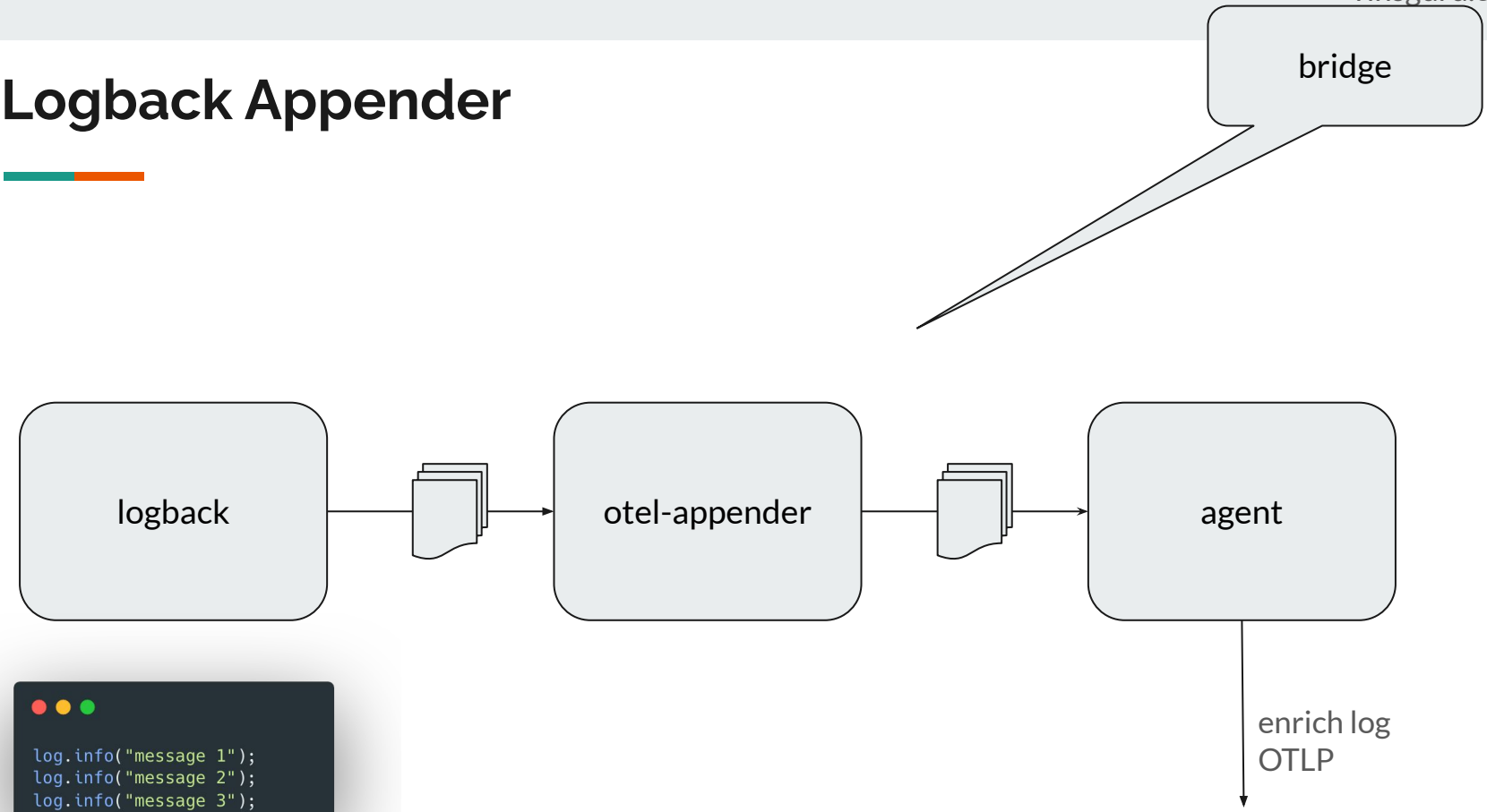
- Prometheus is a ***time-series*** database designed for storing and querying metrics.
  - Collects, Stores and Query these metrics
- PromQL - Prometheus Query Language
  - How many requests failed in the last 5 minutes?
  - What is the average response time of ***movie-service***?

Summary - Prometheus - Data Accuracy!!?

- Metrics are NOT mission-critical data. They are operational signals.
- Metrics are NOT meant for business use cases like billing, customer transactions, or audits.
- Metrics are aggregated
    - We do NOT store every single HTTP request details. We store counters, sums, and buckets.
- Data is scraped at regular intervals (e.g., every 15 seconds). Very short-lived events might be missed.
- Functions like ***rate()***, ***increase()*** are approximations. It is all about finding the **trends / patterns**.

Logs

Logback Appender

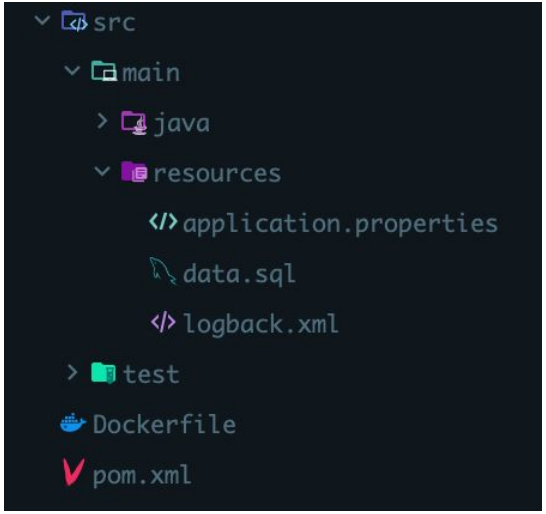



```
log.info("message 1");
log.info("message 2");
log.info("message 3");
```

Logback Appender

We need a logback ***appender*** to capture logging events.

1. ***pom.xml***: Add the otel dependency for appender.
2. **logback.xml**: Configure to use the appender.

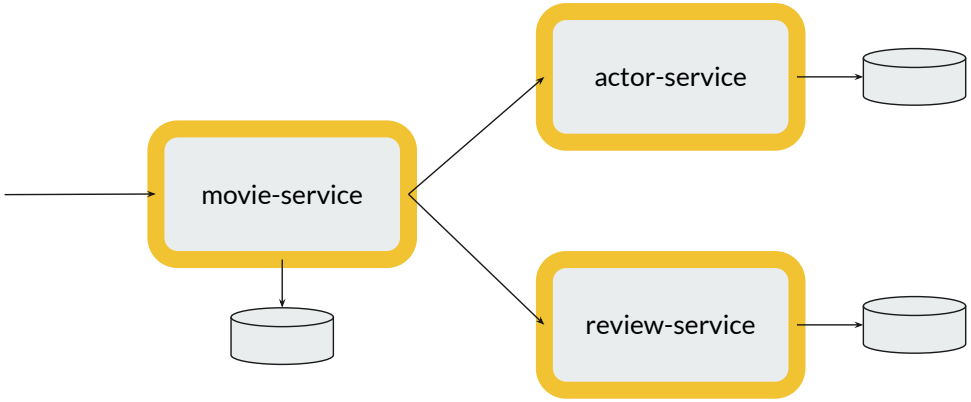



```xml
<configuration>

    <!-- Console Appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} %-5level [%15.15t] %cyan(%-30.30logger{30}) : %m%n</pattern>
        </encoder>
    </appender>

    <!-- OpenTelemetry-Collector Appender -->
    <appender name="OTLP" class="io.opentelemetry.instrumentation.logback.appender.v1_0.OpenTelemetryAppender"/>

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

</configuration>
```

Logs

- Agent can not send the logs to the collector without an appender.
- Agent can enrich the logs with trace context.
  - Agent generates trace id even when we use ***always_off*** sampler!
- We can filter logs for a specific request using the trace id.

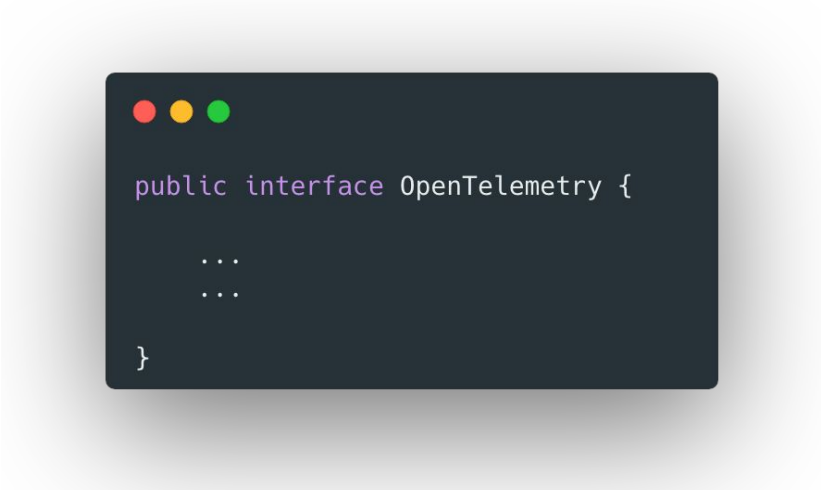
```
[Thread-1] INFO  Starting payment
[Thread-2] INFO  Fetching user profile
[Thread-1] INFO  Payment success
[Thread-2] INFO  User profile loaded
```



Manual Instrumentation

Summary - OpenTelemetry

- It is an interface / main entrypoint to the OpenTelemetry API
- It is the bridge between our application and OpenTelemetry SDK.



Summary - OpenTelemetry - Providers

- A central registry and factory that gives tracers and meters, and manages how traces, metrics, and logs are processed and exported.
  - ***TracerProvider*** → tracing
  - ***MeterProvider*** → metrics
  - ***LoggerProvider*** → logs

```
public interface OpenTelemetry {

    TracerProvider getTracerProvider();
    MeterProvider getMeterProvider();
    LoggerProvider getLogsBridge();
    ...
    ...

}
```

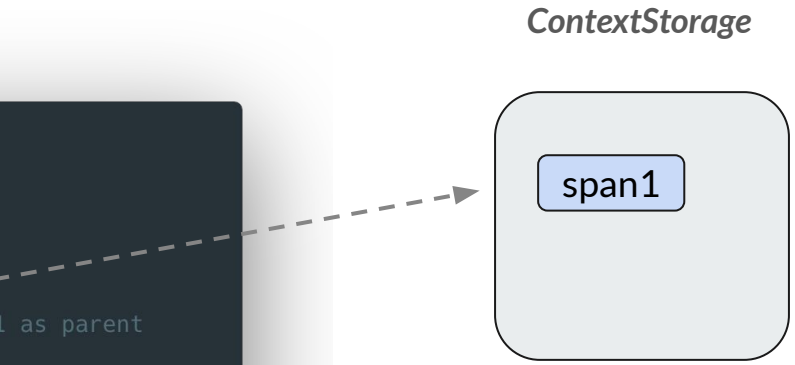
```
OpenTelemetrySdk.builder()
                .setMeterProvider(meterProvider())
                .setTracerProvider(tracerProvider())
                .setLoggerProvider(loggerProvider())
                .build();
```

Summary - Creating Span

```
// Tracer is a lightweight, thread-safe object.
// Create it once (similar to a Spring bean).
Tracer tracer = ...;

// -----------

// Whenever we process a request,
// We create a new Span using the tracer.
Span span = tracer.spanBuilder("process-order")
                  .startSpan();

doSomeWork();

span.end();
```

Summary - Span Parent Child Relationship

```java
// create a root span
var span1 = tracer.spanBuilder("span1").startSpan();

// makeCurrent - span1 is stored in thread-local context
// any spans created within this block will automatically use span1 as parent
try (var scope = span1.makeCurrent()) {
    method2();
}

// inside method2 - any new span will automatically have span1 as parent
void method2() {
    var span2 = tracer.spanBuilder("span2").startSpan();
    // do work...
    span2.end();
}
```



Summary - Context

- Context is an ***immutable*** object that carries request-scoped data (like trace and span information…etc). This context flows across threads and services so different parts of the application can associate their work with the same trace.

Summary - Context wrap

Context.current().wrap(runnable)

```
// pseudo-code showing context wrap
Runnable wrap(Runnable runnable){
    var ctx = ....; // capture current context
    return () -> {
        // set the captured context "ctx" for this new thread
        try(use ctx){
            runnable.run();  // execute code with the correct context
        }
    };
}
```

Summary - Span Kind

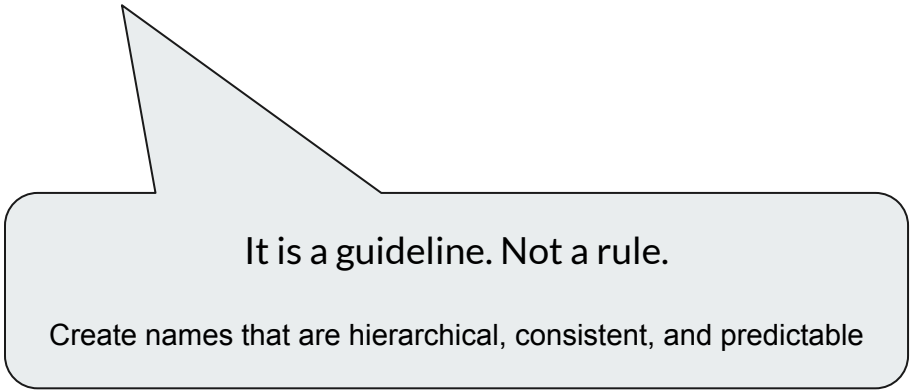
SERVER	Span representing handling an incoming request (e.g., HTTP request to your service).
CLIENT	Span representing making an outgoing request (e.g., HTTP call to another service).
PRODUCER	Span representing sending a message/event to a broker or messaging system.
CONSUMER	Span representing receiving or processing a message/event from a broker.
INTERNAL	Span representing internal operations within your service that are not network calls or messaging.

Summary - Types Of Metrics

- Counter
  - Only goes up
    - number of requests processed
    - number of errors
- Gauge / UpDownCounter
  - Can go up and down
    - CPU usage
    - Available Inventory
- Histogram
  - Distribution of values
    - Request latency (requests completed in <10ms, <50ms, <100ms, etc.).
    - Size of uploaded files (<10MB, <5MB…)

Summary - Metric Naming Convention

- **[namespace].[entity or domain].[operation or action].[type or unit]**
  - namespace → high-level system area (http, db, jvm, app)
  - entity/domain → what the metric is about (server, client, memory, product)
  - operation/action → what is happening (request, view, usage)
  - type/unit → nature of the value (duration, count, utilization)



Summary - Metric Unit

Type	Unit	Comment
Dimensionless Counts	1 {request} {error} {rbc}	plain count request count error count red blood cells count
Duration	s ms us ns	seconds milliseconds microseconds nanoseconds
Data	By	bytes
Ratio / Percentage	1 %	0-1 (dimensionless) 0-100

Summary - MDC

- Mapped Diagnostic Context
- Enrich logs with contextual information automatically included in every log message.

Summary - MDC

MDC uses ThreadLocal

```
// When the user logs in or a request starts
MDC.put("userId", "123");
```

```
public void processPayment(PaymentDetails paymentDetails){
    ...
    ...
    log.info("Payment processed. Amount: {}", paymentDetails.getAmount());
}
```

```
public void sendNotification(NotificationDetails notificationDetails){
    ...
    log.info("Notification email sent");
}
```

```
// When the user logs out or request ends
MDC.clear();
```

```
2025-10-01 10:12:45 INFO  [userId=123] Payment processed. Amount: 250.00
2025-10-01 10:12:45 INFO  [userId=123] Notification email sent
```

Summary - Targeted Debugging

- Sometimes an issue occurs only for a particular user, order, or request. You need detailed context for that single request / transaction.

TurboFilter can enable DEBUG logging dynamically based on certain conditions.