Observability

Closing two threads the API Gateway and Resilience4j lessons deliberately left open: actually propagating the correlation id into order-service's own logs (MDC) and its outgoing calls to inventory-service, structured (JSON) logging, metrics with Micrometer /Actuator, how distributed tracing builds on the same correlation id concept, and how Resilience4j already integrates with Micrometer automatically.

Intermediate 26 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/observability

---

Observability

Several earlier lessons in this category left a thread deliberately unfinished. The API Gateway lesson's `CorrelationIdGatewayFilter` assigned a correlation id but explicitly said "actually propagating it into outgoing calls between order-service and inventory-service... is covered by the upcoming Observability lesson." The Resilience4j lesson's `CircuitBreakerEventListener` logged state transitions, but called that "a real precursor to the metrics/dashboards the Observability lesson covers later." This lesson ties both threads together, and introduces the practice that makes a whole system of independently deployed services actually understandable from the outside.

What Is Observability?

Observability is the ability to understand what's happening INSIDE a running system by examining what it produces from the outside -- its logs, its metrics, and its traces -- without needing to attach a debugger or guess. In a single application, "what's going wrong" often means reading a stack trace in one console. In a system of many services, the same question means correlating information scattered across many DIFFERENT services' logs and dashboards -- which is exactly what observability tooling exists to make practical.

Why Does It Exist?

By the time this course reached the Distributed Transactions lesson, a single business operation (placing an order) could touch order-service's database, a Kafka topic, and inventory-service's database, entirely asynchronously. If something goes wrong -- an order silently never gets confirmed -- there's no single stack trace to look at anymore. Without a

shared correlation id, structured logs, and metrics, diagnosing that failure means manually guessing which service's logs to check and when, across however many services are involved. Observability turns that guesswork into a query.

History

"Observability" as a term borrowed from control theory (a system is observable if its internal state can be inferred from its external outputs) was popularized in software specifically as microservices adoption grew through the 2010s -- monolith-era tooling (one application, one log file, one process to attach a debugger to) simply didn't scale to systems of many independently deployed services. Micrometer, the metrics facade this lesson uses, was created in 2018 specifically to give Spring Boot a vendor-neutral metrics API (the same relationship SLF4J has to logging implementations) -- Spring Boot Actuator has used it as its metrics foundation ever since.

The Three Pillars: Logs, Metrics, and Traces

Observability is commonly described as resting on three complementary kinds of data. LOGS are discrete, timestamped events, usually free text or structured fields -- good for understanding exactly what happened at one point in time. METRICS are numeric measurements aggregated over time (a counter, a gauge, a timing histogram) -- good for spotting trends and setting alerts, but they don't tell you about any ONE specific request. TRACES follow a SINGLE request as it moves across multiple services -- good for answering "where did THIS particular request spend its time, and which service did it fail in." This lesson builds all three, connected by one shared correlation id.

Propagating the Correlation Id: Finishing What the Gateway Started

`CorrelationIdGatewayFilter` (see the API Gateway lesson) already assigns an `X-Correlation-Id` header before a request reaches order-service -- but that header being SET doesn't automatically make order-service's own code, logs, or outgoing calls aware of it. Two pieces close that gap.

```
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import org.slf4j.MDC;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.UUID;

// Runs inside order-service (and inventory-service, the same class copied into
// both -- see "Propagating the Correlation Id: Finishing What the Gateway
// Started"). api-gateway's CorrelationIdGatewayFilter (see the API Gateway
// lesson) already assigns an X-Correlation-Id header before a request ever
// reaches order-service -- this filter is what makes THAT id actually usable
// inside order-service's own code: it reads the header and puts it into SLF4J's
// MDC (Mapped Diagnostic Context), a thread-local map every log statement can
// automatically include (see "Structured Logging: Making Logs Machine-
// Readable").
@Component
@WebFilter("/*")
class CorrelationIdMdcFilter implements jakarta.servlet.Filter {

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

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        // A request that DIDN'T come through api-gateway (a direct call during
        // local development, for instance) won't have this header at all --
        // generating one here means order-service's own logs are still
        // traceable even without the gateway in front of it.
        String correlationId = httpRequest.getHeader(CORRELATION_ID_HEADER);
        if (correlationId == null || correlationId.isBlank()) {
            correlationId = UUID.randomUUID().toString();
        }

        MDC.put(MDC_KEY, correlationId);
        try {
            chain.doFilter(request, response);
        } finally {
            // MDC is thread-local, and this thread will be REUSED for a
            // different request later (a servlet container's thread pool) --
            // forgetting this cleanup would leak one request's correlation id
            // into another request's logs (see "Common Mistakes").
            MDC.remove(MDC_KEY);
        }
    }
}
```

```
import org.slf4j.MDC;
import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;

import java.io.IOException;

// CorrelationIdMdcFilter puts the correlation id into MDC for order-service's
// OWN logs -- but that alone does NOT make inventory-service see the same id
// when order-service calls it through ResilientStockClient (see the
// Resilience4j lesson). Without this interceptor, inventory-service would
// generate a BRAND NEW correlation id for that request (CorrelationIdMdcFilter
// running there too, finding no incoming header) -- breaking the trace right
// at the service boundary.
//
// Registered on the SAME @LoadBalanced RestClient.Builder bean from the
// Service Discovery & Eureka lesson's LoadBalancedRestClientConfig -- adding
// this interceptor doesn't change anything else about how ResilientStockClient
// calls inventory-service, only what headers go along with the call.
class RestClientCorrelationIdInterceptor implements ClientHttpRequestInterceptor {

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

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution)
            throws IOException {
        String correlationId = MDC.get("correlationId");
        if (correlationId != null) {
            request.getHeaders().add(CORRELATION_ID_HEADER, correlationId);
        }
        return execution.execute(request, body);
    }
}
```

> `CorrelationIdMdcFilter` generates its OWN id when no header is present, rather than requiring api-gateway to be in front of every request -- this keeps order-service's own logs traceable even when it's called directly during local development.

Structured Logging: Making Logs Machine-Readable

A plain text log line is easy for a human to read in ONE service's own console, but hard to search reliably across MANY services' logs once they're collected in one place. Structured (JSON) logging turns every field -- including the correlation id now sitting in MDC -- into a queryable field instead of free text.

```
<!--
    Placed at src/main/resources/logback-spring.xml -- Spring Boot picks this up
    automatically, no extra configuration needed. A plain log line ("2026-01-01
    12:00:00 INFO OrderController - order placed") is easy for a HUMAN to read
    in one service's own console, but hard for a LOG AGGREGATOR (something that
    collects logs from every service into one searchable place) to parse
    reliably across many services with slightly different formats. A structured
    (JSON) log line turns every field -- including the correlation id
    CorrelationIdMdcFilter put into MDC -- into a queryable field instead of
    free text (see "Structured Logging: Making Logs Machine-Readable").
-->
<configuration>
    <appender name="JSON_CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <!-- Every key in MDC (correlationId, set by CorrelationIdMdcFilter)
                 is automatically included as its OWN field in the JSON output --
                 no per-field configuration needed here. -->
            <includeMdcKeyName>correlationId</includeMdcKeyName>
        </encoder>
    </appender>

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

Metrics with Micrometer and Actuator

Micrometer is the metrics facade Spring Boot Actuator is built on -- the same relationship SLF4J has to a logging implementation. A `MeterRegistry` bean is autoconfigured automatically once `spring-boot-starter-actuator` is on the classpath.

```
# Added to order-service's application.yml -- widens the "management.endpoints"
# block from the Spring Boot Microservice Basics lesson's "Health Checks: Is
# the Service Up?" section, which only exposed "health". Nothing already
# working (the health endpoint itself) changes.

management:
  endpoints:
    web:
      exposure:
        include: health, metrics, prometheus   # "health" was already here --
                                                # "metrics" exposes Micrometer's
                                                # raw metric data, "prometheus"
                                                # exposes the SAME data in the
                                                # text format a Prometheus
                                                # server scrapes on its own
                                                # schedule (see "Metrics with
                                                # Micrometer and Actuator")
  metrics:
    tags:
      application: ${spring.application.name}  # every metric this service
                                                 # produces gets tagged with its
                                                 # OWN service name -- essential
                                                 # once metrics from many
                                                 # services land in the same
                                                 # place
```

```
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Component;

// Micrometer is the metrics FACADE Spring Boot Actuator is built on -- the same
// relationship SLF4J has to a logging implementation (see the Spring Boot
// Microservice Basics lesson's "Logging and Correlation" section). A
// MeterRegistry bean is autoconfigured automatically once
// spring-boot-starter-actuator is on the classpath, with no extra setup needed
// beyond ActuatorMetricsConfig.yml's exposure settings.
//
// OrderService.create(...) (see the Spring Boot Microservice Basics lesson)
// would call recordOrderPlaced() right alongside publishing OrderPlacedEvent
// (see the Event-Driven Architecture & Kafka lesson) -- a metric and an event
// about the SAME fact, serving different purposes: the event drives the saga,
// the metric drives dashboards and alerts.
@Component
class OrderMetrics {

    private final Counter ordersPlacedCounter;

    OrderMetrics(MeterRegistry meterRegistry) {
        this.ordersPlacedCounter = Counter.builder("orders.placed")
                .description("Number of orders successfully placed")
                .register(meterRegistry);
    }

    void recordOrderPlaced() {
        ordersPlacedCounter.increment();
    }
}
```

Distributed Tracing: Following One Request Across Services

A trace connects every span (one service's contribution to handling a request) that resulted from a single originating request, letting a dashboard show exactly how long order-service spent before calling inventory-service, and how long inventory-service took to respond. Micrometer Tracing (paired with a tracing backend like Zipkin, which actually stores and visualizes traces) builds this on the SAME correlation id concept this lesson already established -- a trace id plays the same connecting role `X-Correlation-Id` does, propagated the same way across the same service boundaries.

> Standing up a real tracing backend (Zipkin, or a hosted equivalent) is genuinely useful in production but out of scope for this lesson to build from scratch -- the correlation id infrastructure above already gives this course's examples a working, searchable trail across logs, which is often enough to start with before adding a dedicated tracing backend.

Exposing What Resilience4j Was Already Tracking

`CircuitBreakerEventListener` (see the Resilience4j lesson) logged state transitions by hand -- but Resilience4j already integrates with Micrometer automatically once both are on the classpath, exposing circuit breaker state, call counts, and failure rates as metrics with NO extra code at all. The manual listener is still useful for immediate, human-readable log lines during development; the Micrometer integration is what a real dashboard or alert would actually watch in production.

Best Practices

- **Propagate the correlation id across EVERY service boundary**, not just the ones this lesson happened to build -- a broken link anywhere in the chain makes the whole trace useless past that point.
- **Tag every metric with the service that produced it** (see `ActuatorMetricsConfig.yml` 's `management.metrics.tags.application`) -- essential once metrics from many services land in one dashboard.
- **Prefer a library's built-in metrics integration (like Resilience4j's) over hand-rolled logging** where one exists -- see "Exposing What Resilience4j Was Already Tracking".
- **Treat logs, metrics, and traces as complementary, not redundant** -- reach for the one that answers the specific question being asked (see "The Three Pillars").

Common Mistakes

- **Forgetting to clear MDC after a request finishes.** A servlet container reuses threads across requests -- without the `finally` block in `CorrelationIdMdcFilter`, one request's correlation id leaks into a completely unrelated later request's logs.
- **Assigning a correlation id at the gateway but never propagating it past the first service.** Without `RestClientCorrelationIdInterceptor`, inventory-service would silently generate its OWN id, breaking the trace exactly at the point it's most needed.
- **Treating metrics as a replacement for logs, or vice versa.** A metric tells you a failure rate went up; a log tells you what actually failed and why -- neither answers the other's question.
- **Adding observability as an afterthought, service by service, instead of a consistent pattern from the start.** A correlation id that's propagated in three services and missing in a fourth breaks every trace that touches that fourth service.

Summary, Cheat Sheet, and Glossary

Observability means understanding a running system from its logs, metrics, and traces, without attaching a debugger. `CorrelationIdMdcFilter` and `RestClientCorrelationIdInterceptor` finish propagating the correlation id api-gateway started assigning, making it usable in structured (JSON) logs and, eventually, distributed traces. Micrometer, autoconfigured through Actuator, provides metrics -- both custom ones (`OrderMetrics`) and ones libraries already produce for free (Resilience4j's circuit breaker metrics). The three pillars answer different questions and are meant to complement each other, not replace one another.

Quick reference:

```
// Propagating correlation id
MDC.put("correlationId", correlationId);           // makes it available to
                                                    // every log line on this thread
request.getHeaders().add("X-Correlation-Id", MDC.get("correlationId"));  // forwards
                                                    // it to the next service

// A custom metric
Counter.builder("orders.placed")
    .register(meterRegistry)
    .increment();
```

Glossary

Observability — The ability to understand a running system's internal state from its external outputs (logs, metrics, traces).

MDC (Mapped Diagnostic Context) — SLF4J's thread-local map that lets contextual data (like a correlation id) be automatically included in every log line on that thread.

Structured Logging — Logging in a machine-parseable format (typically JSON) instead of free text, so individual fields are queryable.

Micrometer — The vendor-neutral metrics facade Spring Boot Actuator is built on.

Trace — The connected set of spans across multiple services that resulted from one originating request.