Observability

API Gateway ve Resilience4j derslerinin kasıtlı olarak açık bıraktığı iki ipliği kapatmak: correlation id'yi order-service'in kendi loglarına (MDC) ve inventory-service'e giden çağrılara GERÇEKTEN yaymak, structured (JSON) loglama, Micrometer/Actuator ile metrikler, distributed tracing'in aynı correlation id kavramı üzerine inşa edilmesi, ve Resilience4j'nin Micrometer'la zaten otomatik entegre olması.

Orta 26 dk

Son güncelleme: 23 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/observability

---

Observability

Bu kategorideki birkaç önceki ders bilinçli olarak bitirilmemiş bir iplik bıraktı. API Gateway dersinin `CorrelationIdGatewayFilter`'ı bir correlation id atadı ama açıkça "onu order-service ile inventory-service arasındaki giden çağrılara GERÇEKTEN aktarmak... yakında gelecek Observability dersinin konusu" dedi. Resilience4j dersinin `CircuitBreakerEventListener`'ı durum geçişlerini logladı, ama bunu "Observability dersinin ilerde kapsayacağı metrik /dashboard'ların gerçek bir öncüsü" olarak adlandırdı. Bu ders her iki ipliği de birbirine bağlıyor, ve bağımsız dağıtılmış bir servisler bütününü dışarıdan gerçekten anlaşılır kılan pratiği tanıtıyor.

Observability Nedir?

Observability, çalışan bir sistemin İÇİNDE ne olduğunu, dışarıdan ürettiklerini -- log'larını, metriklerini ve trace'lerini -- inceleyerek anlama yeteneğidir, bir debugger bağlamaya ya da tahmin etmeye gerek kalmadan. Tek bir uygulamada, "ne yanlış gidiyor" genellikle tek bir konsoldaki bir stack trace'i okumak anlamına gelir. Birçok servisten oluşan bir sistemde, aynı soru, birçok FARKLI servisin log'ları ve dashboard'ları arasına dağılmış bilgiyi ilişkilendirmek anlamına gelir -- observability araçlarının var olma amacı tam olarak bunu pratik hale getirmek.

Neden Var?

Bu kurs "Distributed Transactions" dersine geldiğinde, tek bir iş operasyonu (bir sipariş vermek) order-service'in veritabanına, bir Kafka topic'ine, ve inventory-service'in veritabanına, tamamen asenkron olarak dokunabiliyordu. Bir şeyler ters giderse -- bir sipariş

sessizce hiç onaylanmazsa -- artık bakılacak tek bir stack trace yok. Paylaşılan bir correlation id, structured log'lar ve metrikler olmadan, o başarısızlığı teşhis etmek, kaç servis işin içindeyse, hangi servisin log'larına ne zaman bakılacağını elle tahmin etmek anlamına gelir. Observability bu tahmin oyununu bir sorguya dönüştürür.

Tarihçe

Kontrol teorisinden ödünç alınan "observability" terimi (bir sistemin iç durumu dış çıktılarından çıkarılabiliyorsa o sistem observable'dır), yazılımda özellikle mikroservis benimsemesinin 2010'lar boyunca büyümesiyle popülerleşti -- monolit-dönemi araçları (tek uygulama, tek log dosyası, debugger bağlanacak tek bir süreç) birçok bağımsız dağıtılmış servisten oluşan sistemlere basitçe ölçeklenmiyordu. Bu dersin kullandığı metrik facade'i Micrometer, Spring Boot'a vendor-nötr bir metrik API'si vermek için özel olarak 2018'de yaratıldı (SLF4J'in loglama implementasyonlarıyla olan ilişkisiyle aynı) -- Spring Boot Actuator o zamandan beri metrik temeli olarak onu kullanıyor.

Üç Sütun: Log'lar, Metrikler ve Trace'ler

Observability yaygın olarak üç tamamlayıcı veri türüne dayandığı şeklinde tanımlanır. LOG'lar, ayrık, zaman damgalı olaylardır, genellikle düz metin ya da yapılandırılmış alanlar -- tek bir zaman noktasında tam olarak ne olduğunu anlamak için iyi. METRİKLER, zaman içinde toplanan sayısal ölçümlerdir (bir sayaç, bir gauge, bir zamanlama histogramı) -- eğilimleri fark etmek ve alarm kurmak için iyi, ama herhangi TEK bir istek hakkında bilgi vermezler. TRACE'ler, TEK bir isteği birden fazla servis boyunca izler -- "BU belirli istek zamanını nerede geçirdi, ve hangi serviste başarısız oldu" sorusuna cevap vermek için iyi. Bu ders üçünü de, tek bir paylaşılan correlation id ile bağlanmış olarak inşa ediyor.

Correlation Id'yi Yaymak: Gateway'in Başlattığını Bitirmek

`CorrelationIdGatewayFilter` (API Gateway dersine bakınız), order-service'e bir istek ulaşmadan önce bir `X-Correlation-Id` header'ı zaten atıyor -- ama bu header'ın AYARLANMIŞ olması, order-service'in kendi kodunu, log'larını ya da giden çağrılarını otomatik olarak bunun farkında yapmıyor. İki parça bu boşluğu kapatıyor.

```
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`, api-gateway'in her isteğin önünde olmasını gerektirmek yerine, header hiç yoksa KENDİ id'sini üretir -- bu, order-service yerel geliştirme sırasında doğrudan çağrıldığında bile kendi log'larını izlenebilir tutar.

Structured Logging: Log'ları Makine Tarafından Okunabilir Yapmak

Düz metin bir log satırı, TEK bir servisin kendi konsolunda bir insan için okunması kolay, ama bir kez birçok servisin log'u tek bir yerde toplanınca güvenilir bir şekilde aramak zor. Structured (JSON) loglama, correlation id de dahil her alanı -- şu anda MDC'de oturan -- düz metin yerine sorgulanabilir bir alana dönüştürür.

```
<!--
    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>
```

Micrometer ve Actuator ile Metrikler

Micrometer, Spring Boot Actuator'ın üzerine inşa edildiği metrik facade'i -- SLF4J'in bir loglama implementasyonuyla olan ilişkisiyle aynı. `spring-boot-starter-actuator` classpath'te olduğu anda bir `MeterRegistry` bean'i otomatik olarak yapılandırılır.

```
# 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: Tek Bir İsteği Servisler Boyunca İzlemek

Bir trace, tek bir kaynak istekten kaynaklanan her span'ı (bir servisin bir isteği ele almadaki katkısı) birbirine bağlar, bir dashboard'un order-service'in inventory-service'i çağırmadan önce ne kadar zaman harcadığını, ve inventory-service'in cevap vermesinin ne kadar sürdüğünü tam olarak göstermesini sağlar. Micrometer Tracing (Zipkin gibi, trace'leri gerçekten saklayan ve görselleştiren bir tracing backend'iyle eşleştirilmiş), bunu bu dersin zaten kurduğu AYNI correlation id kavramı üzerine inşa eder -- bir trace id, `X-Correlation-Id` 'nin oynadığı AYNI bağlayıcı rolü oynar, aynı servis sınırları boyunca aynı şekilde yayılır.

> Gerçek bir tracing backend'i (Zipkin, ya da barındırılan bir eşdeğeri) kurmak production'da gerçekten faydalı, ama bunu sıfırdan inşa etmek bu dersin kapsamı dışında -- yukarıdaki correlation id altyapısı bu kursun örneklerine zaten çalışan, aranabilir bir iz veriyor, özel bir tracing backend'i eklemeden önce başlamak için çoğu zaman yeterli.

Resilience4j'nin Zaten İzlediğini Ortaya Çıkarmak

`CircuitBreakerEventListener` (Resilience4j dersine bakınız), durum geçişlerini elle logladı -- ama Resilience4j, ikisi de classpath'teyken Micrometer'la zaten otomatik olarak entegre olur, circuit breaker durumunu, çağrı sayılarını ve başarısızlık oranlarını HİÇBİR ek kod olmadan metrik olarak ortaya çıkarır. Elle yazılan listener, geliştirme sırasında anlık, insan tarafından okunabilir log satırları için hâlâ faydalı; Micrometer entegrasyonu, gerçek bir dashboard ya da alarm'ın production'da gerçekten izleyeceği şey.

Best Practices

- **Correlation id'yi HER servis sınırında yay**, yalnızca bu dersin inşa ettiği sınırlarda değil -- zincirdeki herhangi bir yerde kırık bir bağlantı, o noktadan sonra tüm trace'i işe yaramaz kılar.
- **Her metriği onu üreten servisle etiketle** (`ActuatorMetricsConfig.yml`'nin `management.metrics.tags.application`'ına bakınız) -- birçok servisten gelen metrikler tek bir dashboard'a inince zorunlu.
- **Var olduğunda bir kütüphanenin yerleşik metrik entegrasyonunu (Resilience4j'ninki gibi) elle yazılmış loglamaya tercih et** -- "Resilience4j'nin Zaten İzlediğini Ortaya Çıkarmak" bölümüne bakınız.
- **Log'ları, metrikleri ve trace'leri birbirini TAMAMLAYAN, birbirinin YERİNE geçmeyen şeyler olarak ele al** -- sorulan spesifik soruya cevap vereni seç ("Üç Sütun" a bakınız).

Yaygın Hatalar

- **İstek bittikten sonra MDC'yi temizlemeyi unutmak.** Bir servlet container thread'leri istekler arasında yeniden kullanır -- `CorrelationIdMdcFilter`'daki `finally` bloğu olmadan, bir isteğin correlation id'si tamamen ilgisiz, sonraki bir isteğin log'larına sızar.
- **Correlation id'yi gateway'de atayıp ilk servisin ötesine hiç yaymamak.** `RestClientCorrelationIdInterceptor` olmadan, inventory-service sessizce KENDİ id'sini üretir, trace'i tam da en çok ihtiyaç duyulan noktada kırar.
- **Metrikleri log'ların, ya da tam tersini, yerine geçen bir şey gibi ele almak.** Bir metrik bir başarısızlık oranının arttığını söyler; bir log neyin gerçekten başarısız olduğunu ve nedenini söyler -- hiçbiri diğerinin sorusunu cevaplamaz.
- **Observability'yi baştan tutarlı bir desen yerine, servis servis, sonradan akla gelen bir şey olarak eklemek.** Üç serviste yayılan ama dördüncüde eksik olan bir correlation id, o dördüncü servise dokunan her trace'i kırar.

Özet, Cheat Sheet ve Terimler Sözlüğü

Observability, çalışan bir sistemi, bir debugger bağlamadan, log'larından, metriklerinden ve trace'lerinden anlamak demektir. `CorrelationIdMdcFilter` ve `RestClientCorrelationIdInterceptor`, api-gateway'in atamaya başladığı correlation id'yi yaymayı bitirir, onu structured (JSON) log'larda, ve er ya da geç distributed trace'lerde kullanılabilir yapar. Actuator üzerinden otomatik yapılandırılan Micrometer, hem özel (`OrderMetrics`) hem de kütüphanelerin zaten bedavaya ürettiği (Resilience4j'nin circuit breaker metrikleri) metrikleri sağlar. Üç sütun farklı sorulara cevap verir ve birbirinin yerini almak yerine birbirini tamamlamak için tasarlanmıştır.

Hızlı referans:

```
// Correlation id'yi yaymak
MDC.put("correlationId", correlationId);           // bu thread'deki her log
                                                     // satırında kullanılabilir yapar
request.getHeaders().add("X-Correlation-Id", MDC.get("correlationId"));  // bir
                                                     // sonraki servise iletir

// Özel bir metrik
Counter.builder("orders.placed")
    .register(meterRegistry)
    .increment();
```

Terimler Sözlüğü

Observability — Çalışan bir sistemin iç durumunu dış çıktılarından (log'lar, metrikler, trace'ler) anlama yeteneği.

MDC (Mapped Diagnostic Context) — SLF4J'in, bağlamsal verinin (bir correlation id gibi) o thread'deki her log satırına otomatik olarak dahil edilmesini sağlayan thread-local map'i.

Structured Logging — Düz metin yerine makine tarafından ayrıştırılabilir bir formatta (tipik olarak JSON) loglama, böylece tek tek alanlar sorgulanabilir olur.

Micrometer — Spring Boot Actuator'ın üzerine inşa edildiği, vendor-nötr metrik facade'i.

Trace — Tek bir kaynak istekten kaynaklanan, birden fazla servis boyunca bağlı span kümesi.