# Spring Boot Actuator

# In diesem Modul

- Actuator-Grundlagen: wichtige Endpunkte, Exponierung & Absicherung
- Management-Konfiguration: Base-Path, Ports, Security
- Metriken mit Micrometer: Counter, Gauge, Timer, DistributionSummary
- Health & Probes: Custom HealthIndicator, Liveness/Readiness
- Custom Endpoints und Prometheus/Grafana Integration
- Observability: Micrometer Tracing (OTel Bridge), HTTP/Messaging Propagation
- @Observed Annotation
- Exemplars (Metrics ↔️ Traces Korrelation)

# Was ist Actuator?

- Bietet "production-ready" Features für Monitoring und Management.
- Erlaubt das Überwachen, Sammeln von Metriken, Verstehen des Application-Zustands.
- Exponiert Daten über HTTP-Endpoints oder JMX.

**Dependency:** `spring-boot-starter-actuator`

## Wichtige Endpunkte

- `/actuator/health`: Zeigt den Gesundheitszustand der Anwendung und integrierter Komponenten (DB, Disk, etc.).
- `/actuator/info`: Allgemeine Informationen (Build-Version, Git-Commit).
- `/actuator/metrics`: Listet verfügbare Metriken.
- `/actuator/shutdown`: Beendet die Anwendung (standardmäßig deaktiviert).

# Exponieren und Absichern von Endpoints

Standardmäßig sind nur `/health` und `/info` exponiert.

**`application.yml` Konfiguration:**

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics # Oder '*' für alle
        exclude: shutdown # Sicherheitsrisiko!
      base-path: /manage # Ändert den Basis-Pfad (z.B. /manage/health)
    health:
      show-details: always # Details immer anzeigen
  endpoint:
    shutdown:
      enabled: true # Muss explizit aktiviert werden
```

# Security für Actuator

```
@Configuration @EnableWebSecurity
public class ActuatorSecurity {

    // Import: org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointRequest
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http.authorizeHttpRequests(auth -> auth
                .requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ADMIN") // Nur Admins
                .anyRequest().permitAll()
            )
            .build();
    }
}
```

# Runtime Log-Level Anpassung

Man kann den Log-Level einzelner Pakete **zur Laufzeit** ändern, ohne Neustart.

**Request:**

`POST /actuator/loggers/com.example.service`

```
{
  "configuredLevel": "DEBUG"
}
```

# Custom Metrics mit Micrometer

# Micrometer

Micrometer ist die Metrik-Fassade von Spring Boot, die verschiedene Monitoring-Systeme unterstützt (Prometheus, Datadog, etc.).

## Meter-Typen

- `Counter`: Zählt Inkremente (z.B. Fehler, abgeschlossene Vorgänge).
- `Gauge`: Zeigt den aktuellen Wert an (z.B. Anzahl der Elemente in einer Queue, CPU-Auslastung).
- `Timer`: Misst die Dauer von Operationen.
- `DistributionSummary`: Misst die Verteilung von Werten (z.B. Dateigrößen).

# Beispiel: Counter

```java
@Service
public class OrderService {

    private final Counter orderProcessedCounter;
    private final Counter orderFailedCounter;

    public OrderService(MeterRegistry meterRegistry) {
        this.orderProcessedCounter = meterRegistry.counter("orders.processed", "status", "success");
        this.orderFailedCounter = meterRegistry.counter("orders.processed", "status", "failed");
    }

    public Order processOrder(Order order) {
        try {
            // ...
            orderProcessedCounter.increment();
            return order;
        } catch (Exception e) {
            orderFailedCounter.increment();
            throw e;
        }
    }
}
```

# Beispiel: Timer programmatisch oder deklarativ

```java
@Service
public class OrderService {

    private final Timer orderProcessingTimer;

    public OrderService(MeterRegistry meterRegistry) {
        this.orderProcessingTimer = meterRegistry.timer("orders.processing.duration");
    }

    public Order processOrder(Order order) {
        return orderProcessingTimer.record(() -> {
            // ...
        });
    }

    @Timed(value = "order.creation.time", description = "Zeit für das Erstellen einer Bestellung")
    public Order createOrder(Order order) {
        // ... Logik
        return order;
    }
}
```

# Health Indicator

Standardmäßig prüft Spring Boot die Datenbank, Disk Space etc. Man kann aber eigene Gesundheitschecks hinzufügen.

# Kubernetes Probes (Liveness & Readiness)

In Kubernetes reicht ein einfaches "Health: UP" oft nicht.

- **Liveness Probe:** "Lebt der Container noch?". Wenn nein -> Container Restart.
  - Pfad: `/actuator/health/liveness`
- **Readiness Probe:** "Kann der Container Traffic annehmen?". Wenn nein -> Kein Traffic vom Service.
  - Pfad: `/actuator/health/readiness`

**Aktivierung:**

```
management.endpoint.health.probes.enabled=true
```

# Custom HealthIndicator

```
@Component
public class CustomServiceHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        if (isServiceUp()) {
            return Health.up().withDetail("service.version", "1.0.0").build();
        } else {
            return Health.down()
                        .withDetail("error.message", "Verbindung zu externem Service fehlgeschlagen")
                        .withDetail("last.attempt", LocalDateTime.now())
                        .build();
        }
    }

    private boolean isServiceUp() { /** Implementierung **/}
}
```

Dieser Health Check erscheint dann unter `/actuator/health` als `customService: { "status": "UP", ... }`.

# Custom Actuator Endpoints

Für spezielle Management-Operationen, die nicht von den Standard-Endpoints abgedeckt werden.

```java
@Component
@Endpoint(id = "featureToggle") // Der Pfad wird /actuator/featureToggle
public class FeatureToggleEndpoint {

    private boolean featureEnabled = false;

    @ReadOperation // GET /actuator/featureToggle
    public Map<String, Object> getFeatureStatus() {
        return Map.of("featureEnabled", featureEnabled);
    }

    @WriteOperation // POST /actuator/featureToggle (mit Body)
    public Map<String, Object> setFeatureStatus(@Selector String featureName, boolean enabled) {
        if ("myAdvancedFeature".equals(featureName)) {
            this.featureEnabled = enabled;
            return Map.of("status", "updated", "feature", featureName, "enabled", enabled);
        }
        return Map.of("status", "error", "message", "Unknown feature: " + featureName);
    }
}
```

# Custom Endpoint-Operationen

- `@Endpoint(id = "...")`: Definiert den Basis-Pfad des Endpoints.
- `@ReadOperation`: GET-Operation.
- `@WriteOperation`: POST-Operation.
- `@DeleteOperation`: DELETE-Operation.
- `@Selector`: Ermöglicht Pfad-Variablen (z.B. `/actuator/featureToggle/{featureName}`).

# Prometheus & Grafana Integration

## Prometheus

Ein Open-Source-Monitoring-System, das Metriken "abfragt" (scraped).

1. **Dependency:** `io.micrometer:micrometer-registry-prometheus`
2. **Endpoint:** Spring Boot exponiert einen `/actuator/prometheus` Endpoint im Prometheus-Format.
   Prometheus wird so konfiguriert, dass es diesen Endpoint regelmäßig abfragt.

## Grafana

Eine Open-Source-Plattform für Analysen und interaktive Dashboards.

- Verbindet sich mit Prometheus als Datenquelle.
- Visualisiert die Metriken in Dashboards (z.B. JVM-Metriken, Custom Metrics).

**Ablauf:**

1. Spring Boot App generiert Metriken über Micrometer.
2. `/actuator/prometheus` liefert diese im Prometheus-Format.
3. Prometheus scraped (holt) die Daten regelmäßig von diesem Endpoint.
4. Grafana fragt Prometheus ab und visualisiert die Daten.

# Distributed Tracing & Observability

# Von Sleuth zu Micrometer Tracing

In Spring Boot 3 wurde **Spring Cloud Sleuth** durch **Micrometer Tracing** abgelöst.

- **Ziel:** Einen Request über mehrere Microservices hinweg verfolgen.
- **Trace ID:** Eindeutige ID für den gesamten Request-Flow.
- **Span ID:** ID für einen einzelnen Arbeitsschritt (z.B. Service A ruft Service B).

**Log Korrelation:**
Die IDs werden automatisch in die Logs geschrieben (MDC), sodass man in Kibana/Splunk nach einer TraceID filtern kann.

# Tracing Setup

Benötigte Dependencies (für OpenTelemetry Standard):

```
<!-- Die Fassade -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<!-- Der Exporter (z.B. zu Zipkin oder Jaeger) -->
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

```
Spring Boot konfiguriert den Tracer automatisch, sobald die Bridge im Classpath liegt.
```

# Context Propagation

Damit der Zusammenhang zwischen Requests über Services hinweg erkannt werden kann, müssen Trace-IDs übertragen werden.

- **W3C TraceContext:** Der neue Standard (Default in Boot 3 / OTel).
  - Header: `traceparent`
- **B3 Headers:** Der alte Zipkin/Sleuth Standard.
  - Header: `X-B3-TraceId`, `X-B3-SpanId`

**Wichtig:** Wenn alte Services (Sleuth) mit neuen (Boot 3) zusammenarbeiten, muss man oft das Format anpassen:
`management.tracing.propagation.type=b3`

# HTTP Propagation

Spring Boot instrumentiert automatisch:

- `RestTemplate` / `TestRestTemplate`
- `WebClient`
- `RestClient` (neu)

**Voraussetzung:** Man darf `new RestTemplate()` nicht selbst aufrufen, sondern muss den Builder nutzen!

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    // Der Builder fügt Interceptors hinzu, die den 'traceparent' Header setzen
    return builder.build();
}
```

# Propagation über Messaging (JMS / Kafka)

Der Trace-Kontext kann ähnlich wie bei HTTP auch mit den Headern einer Nachricht im Messaging propagiert werden. Das folgende Beispiel zeigt, wie das bei JMS funktioniert.

## Producer (JMS)

```
@Autowired JmsTemplate jmsTemplate; // Automatisch instrumentiert

public void send() {
    // Schreibt 'traceparent' in die JMS Properties
    jmsTemplate.convertAndSend("queue.orders", new OrderCmd());
}
```

## Consumer

```
@JmsListener(destination = "queue.orders")
public void onMessage(OrderCmd cmd) {
    // Liest 'traceparent', setzt den Context fort
    // und erzeugt einen neuen Child-Span
    log.info("Processing order"); // TraceId ist im Log!
}
```

# Programmatisches Tracing (Observation API)

Die **Observation API** (seit Boot 3) vereinheitlicht Metrics und Tracing.

```
@Autowired ObservationRegistry registry;

public void doWork() {
    Observation.createNotStarted("my.custom.operation", registry)
        .lowCardinalityKeyValue("type", "batch")
        .observe(() -> {
            // Code hier wird gemessen (Timer)
            // UND getraced (Span)
            heavyCalculation();
        });
}
```

# @Observed Annotation

Die deklarative Alternative zur programmatischen Observation API.

```
@Service
public class OrderService {

    @Observed(
        name = "order.processing",
        contextualName = "process-order",
        lowCardinalityKeyValues = {"orderType", "standard"}
    )
    public Order processOrder(Order order) {
        // Automatisch: Timer-Metrik + Trace-Span
        return doProcessing(order);
    }
}
```

**Voraussetzung:** `@EnableAspectJAutoProxy` und `ObservedAspect` Bean.

# ObservedAspect konfigurieren

```
@Configuration
public class ObservabilityConfig {

    @Bean
    public ObservedAspect observedAspect(ObservationRegistry registry) {
        return new ObservedAspect(registry);
    }
}
```

Alternativ: `spring-boot-starter-aop` + Auto-Configuration in Boot 3.2+.

# @Observed vs. Programmatisch

| Aspekt | @Observed | Observation API |
|---|---|---|
| Boilerplate | Minimal | Mehr Code |
| Flexibilität | Standard-Werte | Volle Kontrolle |
| Dynamische Tags | Nein | Ja |
| Error Handling | Automatisch | Manuell möglich |

**Empfehlung:** `@Observed` für Standard-Fälle, API für komplexe Szenarien.

# Exemplars

# Was sind Exemplars?

Exemplars verbinden **Metriken mit Traces**.

- **Problem:** Hohe Latenz in Metrik sichtbar, aber welcher Request war es?
- **Lösung:** Exemplar speichert `traceId` als Referenz zur Metrik.

```
http_request_duration_seconds{...} 0.5 # {traceId="abc123"}
```

# Exemplars aktivieren

```yaml
management:
  metrics:
    distribution:
      percentiles-histogram:
        http.server.requests: true
  prometheus:
    metrics:
      export:
        enabled: true
  tracing:
    sampling:
      probability: 1.0  # 100% Sampling für Demo
```

**Dependency:** `io.micrometer:micrometer-tracing-bridge-otel`

# Exemplars in Prometheus/Grafana

In **Grafana** können Exemplars als Punkte auf Histogrammen angezeigt werden.

1. Prometheus scrapet Metriken mit Exemplars
2. Grafana zeigt Histogramm + Exemplar-Punkte
3. Klick auf Exemplar → Link zu Trace in Jaeger/Zipkin/Tempo

**Ablauf:**

```
Metrik (hohe Latenz) → Exemplar (traceId) → Trace → Root Cause
```

# Exemplar-Konfiguration (Detail)

```
@Configuration
public class ExemplarConfig {

    @Bean
    public DefaultExemplarSampler exemplarSampler(SpanContextSupplier supplier) {
        // Nur bei aktiven Traces Exemplars erzeugen
        return new DefaultExemplarSampler(supplier);
    }
}
```

Spring Boot 3.2+ konfiguriert dies automatisch, wenn Tracing aktiv ist.