Spring Boot

Monitoring

Gesamtinhaltsverzeichnis

1

Einleitung

1 Einleitung

1 Einleitung

Herzlich willkommen zum Seminar über das Monitoring von Spring Boot-Anwendungen! In diesem Seminar werden wir uns mit einer entscheidenden Komponente des Entwicklungsprozesses befassen - dem Monitoring von Spring Boot-Anwendungen.

Spring Boot hat sich zu einem der beliebtesten Frameworks für die Entwicklung von Java-Anwendungen entwickelt. Es bietet eine Vielzahl von Funktionen, die Entwicklern helfen, robuste und skalierbare Anwendungen zu erstellen. Doch um sicherzustellen, dass unsere Anwendungen reibungslos laufen und den Anforderungen gerecht werden, ist ein effektives Monitoring unerlässlich.

In diesem Seminar werden wir uns eingehend mit verschiedenen Aspekten des Monitorings von Spring Boot-Anwendungen befassen. Wir werden untersuchen, warum Monitoring wichtig ist, welche Schlüsselmetriken und Protokolle überwacht werden sollten und wie wir diese Informationen effektiv nutzen können, um die Leistung und Verfügbarkeit unserer Anwendungen zu verbessern.

Wir werden uns auch mit verschiedenen Tools und Techniken zur Überwachung von Spring Boot-Anwendungen befassen. Dabei werden wir sowohl Open-Source-Lösungen als auch kommerzielle Produkte betrachten, die uns helfen, die gewünschten Einblicke in das Verhalten unserer Anwendungen zu gewinnen.

Darüber hinaus werden wir bewährte Vorgehensweisen und Empfehlungen diskutieren, wie wir das Monitoring in unsere Entwicklungs- und Bereitstellungsprozesse integrieren können. Wir werden uns auch mit der Skalierung und Überwachung von verteilten Systemen befassen, da Spring Boot Anwendungen häufig in solchen Umgebungen eingesetzt werden.

Am Ende dieses Seminars werden Sie mit einem fundierten Verständnis für das Monitoring von Spring Boot-Anwendungen ausgestattet sein. Sie werden in der Lage sein, effektive Überwachungsstrategien zu entwickeln und geeignete Tools einzusetzen, um die Leistung und Zuverlässigkeit Ihrer Anwendungen zu optimieren.

Wir freuen uns darauf, Ihnen die Welt des Monitorings von Spring Boot-Anwendungen näherzubringen und Ihnen wertvolle Erkenntnisse und Fähigkeiten zu vermitteln. Lassen Sie uns gemeinsam die Möglichkeiten erkunden, die das Monitoring bietet, um das Beste aus Ihren Spring Boot-Anwendungen herauszuholen!

2

Keypoints

2 Keypoints

1. Bedeutung des Monitorings:
   - Warum ist Monitoring wichtig für Spring Boot-Anwendungen?
   - Auswirkungen eines fehlenden oder unzureichenden Monitorings.

2. Überwachungskonzepte und Metriken:
   - Welche Metriken sind für das Monitoring von Spring Boot-Anwendungen relevant?
   - Überwachung der Ressourcennutzung (CPU, Speicher, Festplatte usw.).
   - Verfolgung von Anfrage-Latenzzeiten und Durchsatz.
   - Protokollierung und Fehlerüberwachung.

3. Monitoring-Tools und -Technologien:
   - Open-Source-Lösungen wie Prometheus, Grafana.
   - Kommerzielle Monitoring-Tools und Cloud-Dienste.
   - Einsatz von APM-Tools (Application Performance Monitoring).

4. Konfiguration des Monitorings in Spring Boot:
   - Integration von Monitoring-Frameworks und Bibliotheken (z. B. Micrometer).
   - Konfiguration von Metrik-Aggregatoren und Protokollierungsformaten.
   - Verwendung von Health Checks und Actuator-Endpunkten.

5. Integration von Monitoring in den Entwicklungsprozess:
   - Inkludierung von Monitoring-Anforderungen in den Anwendungsdesignprozess.
   - Automatisierung von Monitoringerstellung und -bereitstellung.
   - Überwachung von Testumgebungen.

6. Skalierung und Überwachung verteilter Systeme:
   - Herausforderungen beim Monitoring von verteilten Spring Boot-Anwendungen.
   - Anwendung von Distributed Tracing und Service Meshes.
   - Überwachung von Microservices-Architekturen.

7. Best Practices für das Monitoring von Spring Boot-Anwendungen:
   - Festlegung von Schwellenwerten und Alarmierungen.
   - Nutzung von Dashboards und visuellen Darstellungen.
   - Kontinuierliche Verbesserung und Anpassung der Überwachungsstrategie.

3

Codecentric Admin

3 Codecentric Admin

3.1 Beispiel mit drei Servern:

Monitoring von Spring Boot-Anwendungen mit Codecentric Admin

Hinweis: „codecentric“ ist mit der aktuellen Version 3.1.0 noch nicht verfügbar. Wir verwenden deswegen Spring Boot 3.0.7

Das Beispiel mit den drei Spring Boot-Servern besteht aus zwei zu überwachenden Anwendungen, genannt FirstService und SecondService, sowie einer Monitoranwendung, die das Codecentric Admin-Dashboard verwendet.

Die FirstService- und SecondService-Anwendungen repräsentieren typische Spring Boot-Anwendungen, die verschiedene Funktionen und Endpunkte enthalten. Diese Anwendungen können beispielsweise RESTful-APIs bereitstellen, Datenbankzugriffe durchführen oder andere Geschäftslogik implementieren.

Die Monitoranwendung (Codecentric Admin) dient dazu, die beiden Server FirstService und SecondService zu überwachen und deren Status, Metriken und Protokolle anzuzeigen. Die Monitoranwendung bietet ein Admin-Dashboard, in dem die wichtigsten Informationen und Metriken der überwachten Server angezeigt werden.

Hier sind einige der Funktionen, die der Spring Boot Actuator von Codecentric bietet:

1. Gesundheitsüberwachung (Health Monitoring): Der Actuator stellt einen Endpunkt bereit, der den Gesundheitszustand der Anwendung angibt. Dies ermöglicht es Betreibern, den Zustand der Anwendung zu überwachen und zu überprüfen, ob sie ordnungsgemäß funktioniert.

2. Metriken (Metrics): Der Actuator sammelt verschiedene Metriken zur Laufzeit der Anwendung, wie z.B. CPU-Auslastung, Speicherverbrauch und Anzahl der HTTP-Anfragen. Diese Metriken können verwendet werden, um die Leistung der Anwendung zu überwachen und Engpässe zu identifizieren.

3. Überwachung der Ressourcen (Resource Monitoring): Der Actuator bietet Endpunkte, um Informationen über die Ressourcennutzung der Anwendung abzurufen. Dies umfasst Informationen über Prozessorzeit, Speichernutzung und Threadauslastung.

4. Konfigurationsdetails (Configuration Details): Mit dem Actuator können Sie die aktuellen Konfigurationsdetails der Anwendung abrufen. Dies kann hilfreich sein, um zu überprüfen, ob die Anwendung die erwarteten Konfigurationseinstellungen verwendet.

5. Protokollierung und Tracing (Logging and Tracing): Der Actuator ermöglicht die Konfiguration der Protokollierung und das Abrufen von Protokolldateien zur Fehlerbehebung und Überwachung von Anwendungen. Außerdem können Sie Tracing-Informationen erhalten, um die Ausführungspfade von Anfragen nachzuvollziehen.

6. Ausführungsstatus (Runtime Status): Der Actuator stellt Informationen über den Status der Anwendung zur Verfügung, z.B. die Version von Spring Boot, die Java-Version und andere Details zur Laufzeitumgebung.

Um dieses Beispiel umzusetzen, können Sie die folgenden Schritte durchführen:

1. Erstellen Sie die FirstService-Anwendung:

- Konfigurieren Sie die erforderlichen Abhängigkeiten und Funktionen für die FirstService-Anwendung.

- Aktivieren Sie den Spring Boot Actuator und konfigurieren Sie die gewünschten Actuator-Endpunkte, um Metriken, Gesundheitszustand und Protokolle bereitzustellen.

- Aktivieren Sie die Security-Konfiguration für die Actuator-Endpunkte, um den Zugriff auf diese Endpunkte zu kontrollieren.

2. Erstellen Sie die SecondService-Anwendung:

- Führen Sie die gleichen Schritte wie für die FirstService-Anwendung aus, um die erforderlichen Abhängigkeiten, Funktionen, Actuator-Endpunkte und die Security-Konfiguration einzurichten.

3. Erstellen Sie die Monitoranwendung (Codecentric Admin):

- Konfigurieren Sie die erforderlichen Abhängigkeiten für die Monitoranwendung.

- Stellen Sie sicher, dass die Monitoranwendung das Admin-Dashboard bereitstellt, um die überwachten Server anzuzeigen.

- Konfigurieren Sie die Sicherheitseinstellungen für das Admin-Dashboard, um den Zugriff auf die Überwachungsdaten zu kontrollieren.

4. Starten Sie die Anwendungen:

- Starten Sie die FirstService- und SecondService-Anwendungen sowie die Monitoranwendung auf verschiedenen Ports oder Hosts.

Nachdem die Anwendungen gestartet wurden, können Sie das Codecentric Admin-Dashboard aufrufen und die überwachten Server FirstService und SecondService darin anzeigen. Das Dashboard zeigt Informationen wie Metriken (z. B. CPU-Auslastung, Speichernutzung), Gesundheitszustand und Protokolle der überwachten Server an. Sie können auch benutzerdefinierte Dashboards erstellen, um spezifische Metriken und Informationen zu visualisieren und Warnmeldungen einzurichten, um auf bestimmte Ereignisse oder Schwellenwerte zu reagieren.

3.2 Erforderliche Einstellungen:

3.2.1 Dependencies:

FirstService und SecondService:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

und für den AdminServer:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

3.2.2 Die Application-Property für FirstService und SecondService:

```
server.port: 8090 #bzw 8100
management.server.port: 8091 bzw 8101
management.server.address: 127.0.0.1

spring.security.user.name=admin
spring.security.user.password=geheim

spring.boot.admin.client.username=admin
spring.boot.admin.client.password=geheim

#codecentric
spring.boot.admin.client.url=http://localhost:8080
spring.boot.admin.client.instance.name=FirstService
management.endpoints.web.exposure.include=*


management.endpoints.enabled-by-default=true
management.endpoint.info.enabled=true
management.info.env.enabled=true

management.endpoint.health.roles=always

info.app.name=FirstService
info.app.java.version=17
info.app.type=Spring Boot
```

3.2.3 Security:

```
package de.limago.wepapp17.config;


import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public InMemoryUserDetailsManager userDetailsService(PasswordEncoder passwordEncoder) {
      UserDetails user = User.withUsername("user")
            .password(passwordEncoder.encode("password"))
            .roles("USER")
            .build();

      UserDetails admin = User.withUsername("admin")
            .password(passwordEncoder.encode("geheim"))
            .roles("USER", "ADMIN")
            .build();

      return new InMemoryUserDetailsManager(user, admin);
   }

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

      http.authorizeRequests(requests->requests
            .requestMatchers("/actuator/**").permitAll()
```

```
                .anyRequest()
                .permitAll()
        );
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        PasswordEncoder encoder = PasswordEncoderFactories.createDelegatingPasswordEncoder();
        return encoder;
    }
}
```

4

Micrometer

4 Micrometer

4.1 Begriffe zum Thema Überwachung

- Metrik: Eine Metrik ist eine quantitative Messung oder eine numerische Darstellung eines Aspekts des Systems oder der Anwendung. Metriken werden verwendet, um verschiedene Arten von Leistung, Zustand oder Verhalten zu quantifizieren und zu überwachen. Beispiele für Metriken sind die Anzahl der Anfragen pro Sekunde, die CPU-Auslastung oder der Speicherverbrauch.

- Trace: Ein Trace ist eine Aufzeichnung des Pfades, den eine Anfrage in einem verteilten System nimmt. Es enthält Informationen über die verschiedenen Komponenten oder Dienste, die die Anfrage durchläuft, und die Zeiten, die für die Verarbeitung in jedem Schritt benötigt werden. Traces werden verwendet, um die Leistung und das Verhalten von verteilten Systemen zu analysieren und Engpässe oder Fehler zu identifizieren.

- Log: Ein Log ist eine Aufzeichnung von Ereignissen, die während der Ausführung einer Anwendung oder eines Systems auftreten. Logs enthalten normalerweise Zeitstempel, Nachrichten und zusätzliche Informationen wie Warnungen, Fehler oder Debugging-Informationen. Logs dienen zur Fehlerbehebung, Überwachung und Auditing von Anwendungen und Systemen.

- Span: Ein Span ist ein einzelnes Segment eines Traces. Es stellt die Zeitspanne dar, die für die Verarbeitung einer Anfrage in einer Komponente oder einem Dienst benötigt wird. Ein Span enthält Informationen wie Start- und Endzeitpunkte, Dauer, Tags und Protokolleinträge. Spannweiten werden verwendet, um die Leistung einzelner Komponenten in einem verteilten System zu analysieren und Engpässe oder Verzögerungen zu identifizieren.

- Tag: Ein Tag ist ein Schlüssel-Wert-Paar, das einer Metrik, einem Trace oder einem Logeintrag zugeordnet ist. Tags dienen dazu, zusätzliche Informationen oder Metadaten zu einer Messung hinzuzufügen und die Daten nach bestimmten Kriterien zu filtern oder zu gruppieren. Tags können verwendet werden, um Metriken oder Traces nach bestimmten Dimensionen wie dem Namen einer Anwendung, dem Hostnamen oder dem Status zu unterscheiden.

Diese Begriffe sind grundlegend für das Monitoring und die Analyse von Anwendungen und Systemen. Sie ermöglichen es Entwicklern und Betreibern, Daten zu sammeln, zu analysieren und Einblicke in das Verhalten, die Leistung und den Zustand einer Anwendung oder eines Systems zu gewinnen.

4.2 Übersicht

Micrometer ist eine Metriken-Bibliothek, die mit Spring Boot integriert werden kann. Sie ermöglicht die Erfassung und Veröffentlichung von Metriken aus Spring Boot-Anwendungen. Micrometer bietet eine einheitliche Schnittstelle für verschiedene Monitoring-Systeme wie Prometheus, Graphite, InfluxDB und viele andere.

Durch die Integration von Micrometer in Spring Boot können Entwickler Metriken über verschiedene Aspekte ihrer Anwendung erfassen, darunter:

1. Systemmetriken: Micrometer kann Systemmetriken wie CPU-Auslastung, Speicherverbrauch, Threadauslastung und weitere Betriebssystemmetriken erfassen. Diese Metriken helfen dabei, die Auslastung der Anwendungsumgebung zu überwachen.

2. Anwendungsmetriken: Entwickler können spezifische Metriken für ihre Anwendung definieren, um Informationen über bestimmte Funktionen, Abläufe oder Performance-Kennzahlen zu erfassen. Zum Beispiel können Metriken für die Anzahl der Anfragen, die Verarbeitungszeit von Anfragen oder die Anzahl der Datenbankzugriffe definiert werden.

3. Datenbankmetriken: Micrometer bietet Unterstützung für die Erfassung von Metriken aus verschiedenen Datenbanken wie MySQL, PostgreSQL, MongoDB usw. Entwickler können Metriken wie die Anzahl der ausgeführten Datenbankabfragen, die Verbindungs- und Ausführungszeiten und andere relevante Metriken erfassen.

4. HTTP-Anfragemetriken: Micrometer ermöglicht die Erfassung von Metriken für eingehende HTTP-Anfragen, einschließlich Anzahl der Anfragen, Antwortzeiten, Fehlerquoten und Verteilung der Anfragen über verschiedene Endpunkte.

5. Log-Metriken: Micrometer kann auch Metriken aus Log-Nachrichten extrahieren, um beispielsweise die Häufigkeit bestimmter Log-Ereignisse zu überwachen.

4.3 Konzepte von Micrometer

1. Metrik-Registry: Die Metrik-Registry ist die zentrale Komponente von Micrometer. Sie dient zur Registrierung, Speicherung und Verwaltung von Metriken. Eine Metrik-Registry ermöglicht es Ihnen, Metriken zu erstellen, sie zu aktualisieren und statistische Informationen über sie zu generieren.

2. Metriken: Metriken sind quantitative Messwerte, die Informationen über den Zustand, die Leistung oder das Verhalten der Anwendung liefern. Micrometer unterstützt verschiedene Arten von Metriken, darunter Zähler (Counter), Zähler mit Inkrement (LongTaskTimer), Zähler mit Durchsatz (Timer), Histogramme, Verteilungen und vieles mehr.

3. Tags: Tags sind optionale Schlüssel-Wert-Paare, die Metriken zusätzliche Kontextinformationen hinzufügen. Sie ermöglichen eine detaillierte Segmentierung und Filterung von Metriken. Tags können beispielsweise verwendet werden, um Metriken nach bestimmten Dimensionen wie Anwendungskomponenten, Umgebungen oder Versionen zu unterscheiden.

4. Meter-Typen: Micrometer bietet verschiedene Meter-Typen zum Erfassen spezifischer Metrikarten. Zum Beispiel der "Counter" zum Zählen von Ereignissen, der "Timer" zur Messung von Ausführungszeiten, der "Gauge" zur Erfassung von momentanen Werten und der "DistributionSummary" zum Erfassen von Summen und Durchschnittswerten.

5. Metrik-Namen: Jede Metrik wird durch einen eindeutigen Namen identifiziert. Der Name sollte spezifisch und aussagekräftig sein, um die Metrik zu beschreiben, z. B. "anfragen.prozessiert" oder "speicher.belegung". Ein guter Name hilft bei der Identifizierung und Analyse der Metrikdaten.

6. Exporter: Micrometer unterstützt verschiedene Exporter, die die Metrikdaten an verschiedene Backends oder Monitoring-Systeme weiterleiten können. Dazu gehören Prometheus, Graphite, InfluxDB, Datadog und viele andere. Exporter ermöglichen die Visualisierung und Analyse von Metriken in diesen Systemen.

Details zu den Konzepten finden Sie hier: https://micrometer.io/docs/concepts

4.4 Metriken

1. Antwortzeiten: Micrometer kann die Dauer von Operationen oder Aufrufen messen, z.B. die Zeit, die eine bestimmte Methode zum Ausführen benötigt.

2. Fehlerquoten: Es können Metriken für Fehler oder Ausnahmen erfasst werden, um die Häufigkeit und Verteilung von Fehlern in der Anwendung zu überwachen.

3. Durchsatz: Micrometer kann messen, wie viele Operationen pro Zeiteinheit durchgeführt werden, um den Durchsatz der Anwendung zu überwachen.

4. Ressourcennutzung: Es können Metriken zur Überwachung der CPU-Auslastung, des Speicherbedarfs oder anderer Ressourcen in der Anwendung erfasst werden.

5. Datenbankabfragen: Micrometer ermöglicht die Erfassung von Metriken für Datenbankabfragen, wie z.B. die Anzahl der ausgeführten Abfragen, die Ausführungszeit oder den Durchsatz von Datenbankoperationen.

6. HTTP-Anfragen: Es können Metriken für HTTP-Anfragen erfasst werden, wie z.B. die Anzahl der Anfragen, die Antwortzeiten oder der Durchsatz von HTTP-Aufrufen.

4.5 Meter-Register

Es gibt verschiedene Meter-Register (Metrik-Register), die zur Erfassung und Verwaltung von Metriken verwendet werden können. Hier sind einige Meter-Register:

1. Counter: Ein Counter (Zähler) misst die Anzahl von Ereignissen oder Vorkommnissen. Er erhöht sich jedes Mal, wenn ein Ereignis auftritt, und kann zur Erfassung von z.B. Anfragen, Fehler oder Benutzeraktionen verwendet werden.

2. Gauge: Ein Gauge (Messwert) ist ein einzelner Wert, der zu einem bestimmten Zeitpunkt gemessen wird. Er kann verwendet werden, um z.B. aktuelle Werte wie die Größe eines Caches oder den aktuellen Speicherbedarf zu erfassen.

3. Timer: Ein Timer (Zeitmesser) misst die Dauer von Operationen oder Ereignissen. Er erfasst die Zeit zwischen dem Start und dem Ende einer Operation und kann zur Messung von Antwortzeiten, Ausführungszeiten von Methoden usw. verwendet werden.

4. Distribution Summary: Eine Distribution Summary (Verteilungszusammenfassung) misst Verteilungsstatistiken wie den Durchschnitt, das Maximum, das Minimum und die Anzahl der Werte. Sie kann verwendet werden, um z.B. Latenzzeiten oder Größen von Dateien zu erfassen.

5. Long Task Timer: Ein Long Task Timer (Zeitmesser für lange Aufgaben) ist ähnlich wie ein Timer, aber er ist für lang andauernde Aufgaben gedacht. Er kann verwendet werden, um z.B. die Ausführungszeit von Hintergrundprozessen oder Batch-Jobs zu messen.

Diese Meter-Register sind nur einige Beispiele, die in Micrometer verfügbar sind. Es gibt auch weitere Register und Metrik-Typen, die spezifischere Anwendungsfälle abdecken.

5

Codebeispiele für MeterRegister

5 Codebeispiele für MeterRegister

5.1 Configklasse

```
package de.limago.simpleappwithprometheus.config;

import io.micrometer.common.annotation.ValueExpressionResolver;
import io.micrometer.common.annotation.ValueResolver;
import io.micrometer.core.aop.MeterTagAnnotationHandler;
import io.micrometer.core.aop.TimedAspect;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MicrometerConfig {

    @Bean
    public TimedAspect timedAspect(MeterRegistry registry) {
        var timeAspect =  new TimedAspect(registry);
//        ValueResolver valueResolver = parameter -> "Value from myCustomTagValueResolver [" + parameter + "]";
//
//// Example of a ValueExpressionResolver that uses Spring Expression Language
//        ValueExpressionResolver valueExpressionResolver = new SpelValueExpressionResolver();
//
//
//// Setting the handler on the aspect
//        timeAspect.setMeterTagAnnotationHandler(
//                new MeterTagAnnotationHandler(aClass -> valueResolver, aClass -> valueExpressionResolver));
        return timeAspect;
    }
}
```

Hinweis: spring starter aop muss als Dependency in die POM eingetragen werden.

5.2 Controllerklasse

```
package de.limago.simpleappwithprometheus.controller;

import io.micrometer.core.annotation.Counted;
import io.micrometer.core.annotation.Timed;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Tags;
import lombok.extern.log4j.Log4j2;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.LinkedList;
import java.util.List;
import java.util.NoSuchElementException;

@RestController
@RequestMapping("demo")
@Timed("demo")
@Log4j2
public class DemoController {

    private LinkedList<Long> list = new LinkedList<>();
    private final MeterRegistry registry;
    private final MyComponent myComponent;

    // Update the constructor to create the gauge
    DemoController(MeterRegistry registry, MyComponent myComponent ) {
        this.registry = registry;
        this.myComponent = myComponent;
        registry.gaugeCollectionSize("example.list.size", Tags.empty(), list);
    }
    @GetMapping(path = "path-1", produces = MediaType.TEXT_PLAIN_VALUE)
    @Counted("getone.counted")
    public String getOne() {
        log.info("get_one");
        myComponent.foo();
        myComponent.foo();
        myComponent.foo();
        myComponent.bar();
        myComponent.bar();
        return "path-1";
    }
    @GetMapping(path = "path-2", produces = MediaType.TEXT_PLAIN_VALUE)
    // @Timed(value = "example.metric.name", Perzentile = { 0,95, 0,75 })
    // https://medium.com/clarityai-engineering/effectively-measuring-execution-times-with-micrometer-datadog-
5ad15fb8abee
    @Timed(value = "demo.getsecond", description = "eine doofe Beschreibung")
    public String getSecond() throws Exception{
        log.info("get_one");
        Thread.sleep(20);
        return "path-2";
    }



    @GetMapping(path = "gauge/{number}")
    // https://www.baeldung.com/java-netflix-spectator
    public Long checkListSize(@PathVariable("number") long number) {
        log.info("checkListSize");
        if (number == 2 || number % 2 == 0) {
            // add even numbers to the list
            list.add(number);
```

```
        } else {
            // remove items from the list for odd numbers
            try {
                number = list.removeFirst();
            } catch (NoSuchElementException nse) {
                number = 0;
            }
        }
        return number;
    }
}
```

5.3 Serviceklasse

```
package de.limago.simpleappwithprometheus.controller;

import io.micrometer.core.annotation.Timed;
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Metrics;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

@Component

public class MyComponent {

    private final Counter counter = Metrics.counter("bar.counter");


    @Timed("foo")
    public void foo() {
        // do nothing
    }

    public void bar() {
        counter.increment();
    }


}
```

5.4 Metrics.counter

Die Zeile "private final Counter counter = Metrics.counter("bar.counter");" verwendet die Metriken-Bibliothek Micrometer in Java.

Die Methode "Metrics.counter("bar.counter")" ruft die statische Methode "counter()" der Klasse "Metrics" auf, um einen Zähler (Counter) für das Ereignis "bar.counter" zu erstellen.

Ein Counter ist eine Metrik, die verwendet wird, um die Anzahl bestimmter Ereignisse zu zählen. In diesem Fall wird der Counter verwendet, um die Anzahl der Ereignisse des Typs "bar.counter" zu verfolgen.

Ein Gauge hingegen ist eine andere Art von Metrik. Ein Gauge misst den aktuellen Zustand oder Wert einer bestimmten Eigenschaft. Im Gegensatz zum Counter, der inkrementiert oder dekrementiert wird, spiegelt ein Gauge einen festen Wert wider.

5.5 Metrics.gauge

Wenn Sie also einen Gauge anstelle eines Counters erstellen möchten, könnten Sie den Code wie folgt ändern:

private final Gauge gauge = Metrics.gauge("bar.gauge", Tags.empty(), new AtomicInteger());

Hier wird die Methode "Metrics.gauge("bar.gauge", Tags.empty(), new AtomicInteger())" verwendet, um einen Gauge für das Ereignis "bar.gauge" zu erstellen. Das "AtomicInteger"-Objekt dient als Wert des Gauges und kann entsprechend aktualisiert werden, um den aktuellen Zustand widerzuspiegeln.

Die Methode "registry.gaugeCollectionSize("example.list.size", Tags.empty(), list);" in Micrometer erstellt einen sogenannten "Gauge" (eine Metrik) mit dem Namen "example.list.size", der die Größe einer bestimmten Sammlung (z. B. einer Liste) misst und überwacht.

In diesem Fall wird die Methode "gaugeCollectionSize" der "registry"-Instanz aufgerufen, wobei "example.list.size" als Name des Gauges und "Tags.empty()" als leere Menge von Tags übergeben wird. Die Methode erwartet auch eine Sammlung (Collection), in diesem Fall eine Liste, die überwacht werden soll.

Der Gauge, der durch diese Methode erstellt wird, misst und verfolgt automatisch die Größe der übergebenen Liste. Jedes Mal, wenn die Größe der Liste geändert wird (z. B. durch das Hinzufügen oder

Entfernen von Elementen), wird der Gauge aktualisiert und die neue Größe der Liste wird erfasst.

Der Gauge kann dann in einer Metrik-Registry (wie z. B. Prometheus) registriert und überwacht werden. Auf diese Weise können Sie die Größe der Liste im Laufe der Zeit verfolgen und analysieren, um beispielsweise Trends, Spitzenwerte oder andere Muster zu identifizieren.

Hier ist ein Beispiel für die Verwendung der Methode "gaugeCollectionSize":

```
private LinkedList<Long> list = new LinkedList<>();
private final MeterRegistry registry;

// Update the constructor to create the gauge
DemoController(MeterRegistry registry, MyComponent myComponent ) {
   this.registry = registry;
   this.myComponent = myComponent;
   registry.gaugeCollectionSize("example.list.size", Tags.empty(), list);
}
```

In diesem Beispiel wird eine leere Liste "list" erstellt. Der "registry" (der ein CounterService-Objekt ist, das eine Metrik-Registry repräsentiert) wird verwendet, um den Gauge "example.list.size" zu erstellen und mit der Liste zu verknüpfen. Dadurch wird die Größe der Liste überwacht und als Metrik erfasst.

5.6 @Timed

Die Annotation "@Timed("foo")" in Micrometer ist eine Funktion zur Instrumentierung von Code in Java-Anwendungen, um die Ausführungszeit bestimmter Methoden oder Funktionen zu messen.

Indem Sie die Annotation "@Timed" auf eine Methode anwenden und einen Namen als Parameter angeben, können Sie angeben, dass die Ausführungszeit dieser Methode erfasst werden soll. In diesem Fall ist der Name "foo" der Bezeichner für diese Timed-Metrik.

Während der Laufzeit der Anwendung erfasst Micrometer automatisch die Zeit, die für die Ausführung der mit "@Timed" annotierten Methode benötigt wird. Diese Informationen werden dann in den Metrik-Registry-Systemen erfasst und können später analysiert oder überwacht werden.

Das Timed-Feature ist besonders nützlich, um Engpässe oder langsam laufende Methoden in einer Anwendung zu identifizieren. Durch die Messung der Ausführungszeit können Entwickler Performance-Probleme lokalisieren und optimieren.

In diesem Beispiel wird die Methode "public void foo()" mit der "@Timed" Annotation versehen und der Name "foo" als Parameter übergeben. Während der Laufzeit wird die Zeit gemessen, die für die Ausführung dieser Methode benötigt wird, und in den Metriken registriert.

Die erfassten Metriken können dann in verschiedenen Metrik-Registry-Systemen wie Prometheus oder Graphite gesammelt und angezeigt werden, um einen Überblick über die Leistung und Auslastung der Anwendung zu erhalten.

5.7 @Counted

Die Annotationen "@Counted" und "@Timed" sollten nicht gleichzeitig auf dieselbe Methode oder Funktion angewendet werden. @Timed zählt ebenso die Aufrufe. @Counted erfasst jedoch nicht die Laufzeit.

Die Zeile "@Counted("getone.counted")" ist eine Annotation in Micrometer, die verwendet wird, um einen Zähler (Counter) für eine bestimmte Methode zu erstellen und zu inkrementieren.

Die Annotation "@Counted" wird auf eine Methode angewendet, um zu kennzeichnen, dass der Aufruf dieser Methode gezählt werden soll. "getone.counted" ist dabei der Bezeichner oder der Name des Zählers, der diesen spezifischen Aufruf verfolgt.

Wenn die annotierte Methode aufgerufen wird, inkrementiert Micrometer automatisch den entsprechenden Zähler. Dadurch können Sie die Anzahl der Aufrufe dieser Methode erfassen und überwachen.

Hier ist ein Beispiel für die Verwendung der "@Counted" Annotation:

```
@GetMapping(path = "path-1", produces = MediaType.TEXT_PLAIN_VALUE)
@Counted("getone.counted")
public String getOne() {
   log.info("get_one");
   myComponent.foo();
   myComponent.foo();
   myComponent.foo();
   myComponent.bar();
   myComponent.bar();
   return "path-1";
}
```

In diesem Beispiel wird die Methode "getOne()" mit der "@Counted" Annotation versehen und der Name "getone.counted" als Parameter übergeben. Jedes Mal, wenn die Methode aufgerufen wird, erhöht sich der Zähler "getone.counted" um eins.

Die erfassten Zähler können dann in einer Metrik-Registry (wie z. B. Prometheus) registriert und überwacht werden. Dies ermöglicht die Analyse der Anzahl der Aufrufe der annotierten Methode und kann beispielsweise zur Identifizierung von stark beanspruchten oder selten aufgerufenen Funktionen in einer Anwendung verwendet werden.

5.8 Histogramme

Ein Histogramm ist eine statistische Metrik, die in Micrometer verwendet wird, um die Verteilung von Werten zu erfassen. Es ermöglicht die Analyse und Überwachung von Wertebereichen und deren Häufigkeit in einem bestimmten Zeitraum.

Im Zusammenhang mit Micrometer wird ein Histogramm verwendet, um die Verteilung von Werten einer bestimmten Metrik zu quantifizieren. Es wird normalerweise verwendet, um Metriken wie die Antwortzeit von Anfragen, die Größe von Datenpaketen oder andere kontinuierliche numerische Werte zu erfassen.

Micrometer bietet eine spezielle Klasse namens `Histogram` zur Erfassung und Verwaltung von Histogramm-Metriken. Sie können ein Histogramm erstellen, indem Sie die entsprechende Metrik-Registry verwenden und die erforderlichen Konfigurationen festlegen.

Hier ist ein Beispiel für die Verwendung eines Histogramms in Micrometer:

Histogram histogram = registry.histogram("response.time");

In diesem Beispiel wird ein Histogramm mit dem Namen "response.time" in der Metrik-Registry erstellt. Das Histogramm kann dann verwendet werden, um Werte zu erfassen und statistische Informationen über die Verteilung dieser Werte zu generieren.

Um Werte zum Histogramm hinzuzufügen, können Sie die Methode `record(value)` verwenden:

histogram.record(50); // Füge einen Wert von 50 zum Histogramm hinzu

Das Histogramm aggregiert und analysiert automatisch die hinzugefügten Werte und generiert statistische Informationen wie den Durchschnitt, die Standardabweichung, das Maximum, das Minimum und verschiedene Quantile (z. B. das 99. Perzentil). Diese Informationen können verwendet werden, um die Verteilung der Metrikwerte zu verstehen und Engpässe oder Ausreißer zu identifizieren.

Die Konfiguration des Histogramms in Micrometer umfasst verschiedene Parameter wie den Bucketsbereich (definiert die Wertebereiche für die Erfassung von Häufigkeiten) und die Anzahl der Abtastungen. Die genaue Konfiguration kann je nach den Anforderungen Ihrer Anwendung variieren.

Die Annotation `@Timed("bla", histogramm=true)` wird in Micrometer verwendet, um die Zeitmessung einer Methode zu aktivieren und gleichzeitig ein Histogramm für die erfassten Zeiten zu erstellen.

Wenn Sie diese Annotation auf eine Methode anwenden, wird die Ausführungszeit der Methode erfasst und als Metrik gesammelt. Das `@Timed`-Attribut ermöglicht es Ihnen, den Namen der Metrik anzugeben, unter der die erfasste Zeit gespeichert wird. In diesem Fall wird der Name "bla" verwendet.

Durch das Hinzufügen von `histogramm=true` wird Micrometer auch ein Histogramm für die erfassten Zeiten erstellen. Ein Histogramm erfasst nicht nur den Durchschnitt oder das Maximum der Zeitwerte, sondern erfasst auch Informationen über die Verteilung der Zeiten, einschließlich Quantilen und Buckets.

Hier ist ein Beispiel für die Verwendung der `@Timed`-Annotation mit Histogramm in Micrometer:

```
@Timed(value = "bla", histogram = true)
public void myMethod() {
    // Methodenimplementierung
}
```

Mit dieser Konfiguration wird die Ausführungszeit der Methode "myMethod()" gemessen und in der Metrik mit dem Namen "bla" gespeichert. Gleichzeitig wird ein Histogramm der erfassten Zeiten erstellt, das zusätzliche statistische Informationen über die Verteilung der Zeiten bietet.

6

Actuatorendpunkte

6 Actuatorendpunkte

Der Actuator-Endpunkt (Actuator endpoint) ist ein spezieller Endpunkt in einer Anwendung, der von der Spring Boot Actuator-Bibliothek bereitgestellt wird. Diese Bibliothek ermöglicht das Überwachen und Verwalten einer laufenden Spring Boot-Anwendung.

Der Actuator-Endpunkt bietet verschiedene nützliche Funktionen und Informationen über den Zustand und die Verwaltung der Anwendung. Diese Endpunkte sind standardmäßig in einer Spring Boot-Anwendung verfügbar und können für Diagnosezwecke oder zur Überwachung des Systems verwendet werden.

Einige der häufig verwendeten Actuator-Endpunkte sind:

1. `/actuator/health`: Gibt den Zustand der Anwendung zurück (z. B. "UP" für gesund oder "DOWN" für nicht verfügbar). Dieser Endpunkt wird oft verwendet, um die Gesundheit des Systems zu überprüfen.

2. `/actuator/info`: Liefert allgemeine Informationen über die Anwendung, wie z. B. Versionsnummern, Beschreibungen oder andere konfigurierbare Metadaten.

3. `/actuator/metrics`: Bietet Zugriff auf verschiedene Metriken der Anwendung, wie z. B. CPU-Auslastung, Speicherbedarf oder Anzahl der Anfragen. Dieser Endpunkt ermöglicht die Überwachung der Leistung und Ressourcennutzung der Anwendung.

4. `/actuator/auditevents`: Stellt eine Liste von Auditing-Ereignissen bereit, die in der Anwendung aufgetreten sind. Dieser Endpunkt kann verwendet werden, um Informationen über Aktionen oder Ereignisse zu erhalten, die in der Anwendung protokolliert wurden.

Es gibt noch weitere Endpunkte, die von der Spring Boot Actuator-Bibliothek bereitgestellt werden. Sie können auch eigene benutzerdefinierte Endpunkte erstellen, um spezifische Informationen oder Aktionen in der Anwendung zu unterstützen.

Die Actuator-Endpunkte können über HTTP-Anfragen aufgerufen werden, und die zurückgegebenen Informationen können in einem

geeigneten Format wie JSON oder XML abgerufen werden. Die genaue Konfiguration der Endpunkte und deren Zugriffsrechte können in der Konfigurationsdatei der Spring Boot-Anwendung festgelegt werden.

Hinweis: Die genauen verfügbaren Endpunkte und ihre Pfade können je nach verwendeter Version der Spring Boot Actuator-Bibliothek variieren. Es wird empfohlen, die offizielle Dokumentation der verwendeten Version zu konsultieren, um detaillierte Informationen zu erhalten.

Hier finden Sie eine genaue Beschreibung: https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/

6.1 Security

Actuator-Endpunkte in einer Anwendung bieten wertvolle Einblicke in den Zustand, die Konfiguration und die Metriken der Anwendung. Da diese Endpunkte sensible Informationen preisgeben können, ist es wichtig, sie angemessen abzusichern und mit Sicherheitsmechanismen zu versehen. Hier sind einige Gründe, warum Actuator-Endpunkte mit Sicherheit versehen werden sollten:

1. Schutz sensibler Informationen: Actuator-Endpunkte können Details über den Zustand und die Konfiguration der Anwendung preisgeben, einschließlich Metriken, Environment-Daten, Thread-Stack-Traces usw. Diese Informationen können wertvoll sein, aber sie sollten nicht für jeden zugänglich sein. Durch die Verwendung von Sicherheitsmechanismen wie Authentifizierung und Autorisierung können Sie sicherstellen, dass nur autorisierte Benutzer oder Systeme auf diese sensiblen Informationen zugreifen können.

2. Verhinderung von unerwünschten Aktionen: Einige Actuator-Endpunkte ermöglichen auch das Ausführen von Aktionen, die Auswirkungen auf die Anwendung haben können, wie das Neustarten der Anwendung oder das Aktualisieren der Konfiguration. Wenn diese Endpunkte nicht richtig gesichert sind, können unautorisierte Benutzer oder böswillige Angreifer möglicherweise diese Aktionen ausführen und die Anwendung beeinträchtigen. Durch die Implementierung von Sicherheitsmaßnahmen können Sie unerwünschte Aktionen verhindern und die Integrität Ihrer Anwendung schützen.

3. Erfüllung von Compliance-Anforderungen: In einigen Branchen oder Anwendungsfällen gibt es spezifische Compliance-Anforderungen hinsichtlich der Sicherheit und des Schutzes von sensiblen Daten. Indem Sie Actuator-Endpunkte mit Sicherheitsmaßnahmen versehen, können

Sie dazu beitragen, diese Compliance-Anforderungen zu erfüllen und die Vertraulichkeit und Integrität der Anwendung zu gewährleisten.

4. Reduzierung von Angriffsflächen: Eine nicht ausreichend gesicherte Actuator-Endpunkt-Schnittstelle kann eine potenzielle Angriffsfläche für Angreifer darstellen. Durch die Implementierung von Sicherheitsmechanismen können Sie das Risiko von unbefugtem Zugriff, Denial-of-Service-Angriffen oder anderen bösartigen Aktivitäten verringern und die Angriffsfläche Ihrer Anwendung reduzieren.

Die genaue Art der Sicherheitsmechanismen, die für Actuator-Endpunkte implementiert werden sollten, hängt von den spezifischen Anforderungen und der verwendeten Technologie ab. Es können verschiedene Ansätze verwendet werden, wie beispielsweise die Integration mit vorhandenen Authentifizierungs- und Autorisierungssystemen, die Verwendung von API-Schlüsseln oder die Konfiguration von Zugriffsbeschränkungen auf Netzwerkebene. Es wird empfohlen, die Sicherheitsrichtlinien und bewährten Methoden Ihrer Plattform oder Frameworks zu konsultieren, um sicherzustellen, dass die Actuator-Endpunkte angemessen abgesichert sind.

7

Monitoring Systems

7 Monitoring Systems

7.1 Übersicht

Micrometer unterstützt eine Vielzahl von Monitoring-Systemen und Backends, mit denen Sie Ihre Metriken visualisieren, analysieren und alarmieren können. Hier sind einige der unterstützten Monitoring-Systeme für Micrometer:

1. Prometheus: Prometheus ist ein leistungsstarkes Open-Source-Monitoring-System. Micrometer bietet native Integration mit Prometheus, sodass Sie Metriken direkt an Prometheus weiterleiten können. Prometheus bietet umfangreiche Funktionen zur Visualisierung, Abfrage und Alarmierung von Metriken.

2. Graphite: Graphite ist ein bekanntes Monitoring-System, das sich auf die Visualisierung von Metriken spezialisiert hat. Micrometer ermöglicht es Ihnen, Metriken an Graphite zu senden, um Grafiken und Dashboards zu erstellen und Metrikdaten zu analysieren.

3. InfluxDB: InfluxDB ist eine leistungsstarke Zeitreihendatenbank. Micrometer bietet Unterstützung für die Integration mit InfluxDB, sodass Sie Metriken in InfluxDB speichern und die Daten für Analysen und Visualisierungen nutzen können.

4. Datadog: Datadog ist ein umfassendes Monitoring- und APM-Tool. Micrometer unterstützt die Integration mit Datadog, sodass Sie Metriken an Datadog senden und dort visualisieren und analysieren können. Darüber hinaus bietet Datadog fortschrittliche Funktionen wie Alarmierung und Anomalieerkennung.

5. Wavefront: Wavefront ist ein Cloud-basiertes Monitoring- und Analysewerkzeug. Micrometer ermöglicht die Integration mit Wavefront, sodass Sie Metriken an Wavefront senden und die leistungsstarken Analyse- und Visualisierungsfunktionen nutzen können.

6. New Relic: New Relic ist ein beliebtes APM- und Monitoring-Tool. Micrometer bietet Unterstützung für die Integration mit New Relic, sodass Sie Metriken an New Relic senden und die umfangreichen Funktionen zur Überwachung und Leistungsoptimierung nutzen können.

Dies sind nur einige der unterstützten Monitoring-Systeme für Micrometer. Es gibt auch Integrationen mit anderen Tools und Systemen wie AppOptics, Humio, Azure Monitor, Google Cloud Monitoring und mehr. Die breite Unterstützung für verschiedene Monitoring-Systeme ermöglicht es Ihnen, Micrometer in Ihrer bevorzugten Umgebung zu nutzen und Metrikdaten effektiv zu visualisieren, analysieren und alarmieren.

7.2 Push basierte Systeme

Micrometer unterstützt auch Push-basierte Monitoring-Systeme, bei denen die Anwendung aktiv die Metrikdaten an das Monitoring-System sendet.

Liste einiger Push-Basierten Systeme:

1. AppOptics: AppOptics ist ein Cloud-basiertes Monitoring- und Observability-Tool von SolarWinds. Es ermöglicht die Überwachung von Anwendungen, Infrastruktur und Metriken in Echtzeit. AppOptics bietet Funktionen wie Metrik-Visualisierung, Alarmierung, Tracing und Log-Monitoring.

2. Atlas: Atlas ist ein Cloud-basiertes Monitoring-System von MongoDB. Es wurde entwickelt, um speziell MongoDB-Datenbanken zu überwachen und bietet umfangreiche Metrik- und Leistungsanalysen für MongoDB-Instanzen.

2.b Atlas Netflix: Atlas ist ein internes Cloud-native Monitoring-System, das von Netflix entwickelt wurde. Es wurde speziell für die Überwachung der Netflix-Infrastruktur und -Anwendungen in der Cloud entwickelt. Atlas basiert auf dem Open-Source-Projekt Graphite und wurde stark erweitert, um den Anforderungen von Netflix gerecht zu werden.

Atlas ermöglicht die Erfassung und Speicherung von Metriken in Echtzeit. Es unterstützt die Überwachung von verschiedenen Komponenten, einschließlich Anwendungen, Infrastruktur, Netzwerk und mehr. Mit Atlas

können Metriken visualisiert, analysiert und alarmiert werden, um die Leistung und den Zustand des Netflix-Ökosystems zu überwachen.

Eine besondere Eigenschaft von Atlas ist seine Skalierbarkeit und Fähigkeit, mit der dynamischen und verteilten Natur der Netflix-Cloud umzugehen. Es kann große Mengen von Metrikdaten verarbeiten und bietet Mechanismen zur Aggregation und Zusammenfassung von Metriken auf verschiedene Ebenen, um eine effiziente Speicherung und Abfrage zu ermöglichen.

Atlas bietet auch umfangreiche Tools und APIs zur Interaktion mit den Metrikdaten. Entwickler können auf die Metriken über benutzerdefinierte Dashboards, Abfragesprachen und API-Zugriff zugreifen, um spezifische Analysen und Diagnosen durchzuführen.

Als internes Monitoring-System von Netflix wurde Atlas speziell auf die Anforderungen und Skalierung der Netflix-Infrastruktur zugeschnitten. Es bietet eine leistungsstarke und flexible Lösung für die Überwachung und Analyse von Metriken in der Cloud-Umgebung von Netflix.

3. Azure Monitor: Azure Monitor ist der Monitoring- und Diagnose-Dienst von Microsoft Azure. Es ermöglicht die Überwachung von Anwendungen, Infrastruktur und Metriken in der Azure-Cloud. Azure Monitor bietet Funktionen wie Metrik- und Protokollüberwachung, Alarmierung und Diagnosewerkzeuge.

4. Datadog: Datadog ist ein leistungsstarkes Observability- und Monitoring-Tool für Cloud- und On-Premises-Umgebungen. Es unterstützt die Überwachung von Metriken, Protokollen, Tracing und APM (Application Performance Monitoring). Datadog bietet umfangreiche Funktionen zur Visualisierung, Alarmierung und Analyse von Metriken.

5. Dynatrace: Dynatrace ist ein umfassendes APM- und Observability-Tool, das speziell für die Überwachung von verteilten Anwendungen und Microservices entwickelt wurde. Es bietet automatische Leistungsanpassung, End-to-End-Tracing, Metriküberwachung und KI-basierte Anomalieerkennung.

6. Elastic: Elastic ist das Unternehmen hinter Elasticsearch, Kibana und der Elastic Stack. Die Elastic-Plattform bietet Funktionen für Log-Analyse, Metriküberwachung, Protokollüberwachung, APM und SIEM (Security Information and Event Management).

7. Graphite: Graphite ist ein Open-Source-Monitoring- und Metriksystem, das sich auf die Speicherung und Visualisierung von Zeitreihen-Metriken spezialisiert hat. Es bietet Funktionen zur Aggregation, Grafikdarstellung und Abfrage von Metriken.

8. Ganglia: Ganglia ist ein Open-Source-Monitoring-System für Cluster- und Grid-Computing-Umgebungen. Es wurde entwickelt, um die Leistung von verteilten Systemen zu überwachen und bietet Funktionen zur Erfassung und Visualisierung von Metriken.

9. Humio: Humio ist ein modernes Log-Management- und Observability-Tool. Es ermöglicht die Erfassung, Speicherung, Analyse und Visualisierung von Protokolldaten. Humio bietet Echtzeit-Suche, Dashboards und Alarmierungsfunktionen.

10. InfluxDB: InfluxDB ist eine Open-Source-Zeitreihendatenbank, die für die Speicherung und Abfrage von Metriken optimiert ist. Sie ermöglicht die Erfassung und Analyse von Zeitreihen-Metriken in Echtzeit.

11. JMX (Java Management Extensions): JMX ist ein Java-Framework zur Verwaltung, Überwachung und Steuerung von Java-Anwendungen

7.3 Pull basierte Systeme

Micrometer unterstützt auch Pull-basierte Monitoring-Systeme, bei denen das Monitoring-System die Metrikdaten (via Rest-Api) von der Anwendung abruft. Hier sind einige der unterstützten Pull-Monitoring-Systeme für Micrometer:

1. Prometheus: Prometheus ist ein bekanntes Pull-basiertes Monitoring-System, das in der Cloud-Native-Umgebung weit verbreitet ist. Micrometer bietet eine native Integration mit Prometheus, sodass Sie

Metriken von Ihrer Anwendung in einem Prometheus-freundlichen Format bereitstellen können. Prometheus kann dann die Metriken von der Anwendung abrufen und für Visualisierung, Abfrage und Alarmierung verwenden.

2. JMX (Java Management Extensions): Micrometer bietet Unterstützung für die Integration mit JMX, einem Java-Framework zur Verwaltung und Überwachung von Anwendungen. Mit der JMX-Integration können Sie Metriken Ihrer Anwendung über JMX exportieren und von einem JMX-basierten Monitoring-System abrufen.

3. OpenTelemetry: OpenTelemetry ist ein Framework zur Instrumentierung von Anwendungen für die Tracing- und Metrik-Erfassung. Micrometer unterstützt die Integration mit OpenTelemetry, sodass Sie Metriken über das OpenTelemetry-Protokoll bereitstellen können. Monitoring-Systeme, die OpenTelemetry unterstützen, können dann die Metriken von der Anwendung abrufen.

4. Azure Monitor: Azure Monitor ist ein Monitoring- und Diagnose-Dienst von Microsoft Azure. Micrometer bietet eine spezielle Integration mit Azure Monitor, sodass Sie Metriken von Ihrer Anwendung an Azure Monitor senden können. Azure Monitor ruft die Metriken dann über die Azure Monitor-Pull-Schnittstelle ab.

Die Konfiguration und der Betrieb von Pull-basierten Monitoring-Systemen werden normalerweise vom Monitoring-System selbst verwaltet werden. Die Anwendung muss lediglich die Metriken zur Verfügung stellen, damit sie abgerufen werden können. Die Wahl des geeigneten Monitoring-Systems hängt von den Anforderungen Ihrer Anwendung und Ihrer Präferenz ab.

Durch die Unterstützung von Pull-Monitoring-Systemen ermöglicht es Micrometer Ihnen, Metriken in verschiedenen Umgebungen und mit verschiedenen Monitoring-Tools zu verwenden. Sie können Metriken effizient bereitstellen und Monitoring-Systeme können die Metriken von Ihrer Anwendung abrufen, um eine umfassende Überwachung und Analyse durchzuführen.

7.4 Dimensionalität

7.4.1 Beschreibung

In Micrometer bezieht sich die Dimensionalität auf die Möglichkeit, Metriken mit zusätzlichen Dimensionen zu versehen. Dies ermöglicht eine detailliertere Analyse und Segmentierung der Metrikdaten. Micrometer unterstützt die Verwendung von Tags, um Metriken mit benutzerdefinierten Dimensionen zu versehen.

Durch die Hinzufügung von Dimensionen können Metriken in verschiedene Kategorien unterteilt werden, basierend auf Attributen wie zum Beispiel Anwendung, Umgebung, Instanz, Region oder beliebigen anderen benutzerdefinierten Dimensionen. Dies ermöglicht es, Metrikdaten granularer zu analysieren und zu verstehen, wie verschiedene Faktoren die Leistung und das Verhalten der Anwendung beeinflussen.

Ein einfaches Beispiel könnte die Metrik "Anfrage-Dauer" sein. Durch Hinzufügen von Dimensionen wie "Anwendung" und "Endpunkt" können Sie die Anfrage-Dauer für verschiedene Anwendungen und Endpunkte getrennt betrachten und vergleichen. Dies ermöglicht es Ihnen, Engpässe zu identifizieren, bestimmte Bereiche zu optimieren oder Abweichungen zwischen verschiedenen Anwendungen zu erkennen.

Die Dimensionalität in Micrometer erweitert also den Funktionsumfang von Metriken, indem sie ihnen zusätzliche Informationen und Kontext verleiht. Dies erleichtert die Analyse, Visualisierung und das Monitoring von Metrikdaten und unterstützt bei der Fehlerbehebung und Optimierung von Anwendungen.

7.4.2 Unterstützung

Die Unterstützung für Dimensionalität in Micrometer hängt jedoch von den zugrunde liegenden Monitoring- und Metriksystemen ab, mit denen Micrometer integriert ist. Hier ist eine Aufschlüsselung, welche Systeme die Dimensionalität unterstützen und welche nicht:

Unterstützen Dimensionalität (Beispiele):

- Prometheus: Prometheus unterstützt die Verwendung von Labels, die ähnlich wie Tags in Micrometer fungieren. Labels ermöglichen die Segmentierung und Filterung von Metriken nach verschiedenen Dimensionen.

- InfluxDB: InfluxDB unterstützt die Verwendung von Tags, um Metriken nach verschiedenen Dimensionen zu gruppieren und abzufragen.

- OpenTelemetry: OpenTelemetry unterstützt die Verwendung von Attributen, die den Metriken hinzugefügt werden können, um zusätzliche Dimensionen darzustellen.

Unterstützen keine Dimensionalität:

- JMX (Java Management Extensions): JMX selbst unterstützt keine Dimensionalität. Es ermöglicht die Überwachung von Attributen einzelner MBeans, jedoch nicht die Segmentierung von Metriken nach Dimensionen.

- Graphite: Graphite unterstützt standardmäßig keine Tags oder Dimensionen. Es basiert auf einem flachen Namensraum für Metriken.

- Ganglia: Ganglia unterstützt ebenfalls keine nativen Tags oder Dimensionen. Es basiert auf einem flachen Namensraum für Metriken.

Diese Liste basiert auf dem aktuellen Stand und ie Funktionen der verschiedenen Monitoring- und Metriksysteme können sich im Laufe der Zeit ändern. Daher ist es ratsam, die Dokumentation der einzelnen Systeme zu überprüfen, um herauszufinden, ob und wie sie die Dimensionalität unterstützen.

8

Prometheus

8 Prometheus

8.1 Merkmale

Prometheus ist ein leistungsstarkes Open-Source-Monitoring- und Alarmierungssystem, das ursprünglich von SoundCloud entwickelt wurde und jetzt Teil des Cloud Native Computing Foundation (CNCF) Projekts ist. Es wurde entwickelt, um Anwendungen und Systeme in einer dynamischen, skalierbaren und verteilten Umgebung zu überwachen.

Prometheus zeichnet sich durch folgende Merkmale aus:

1. Metrikerfassung: Prometheus ermöglicht die Erfassung und Speicherung von Metriken in Echtzeit. Es kann Metriken von verschiedenen Komponenten wie Anwendungen, Systemen, Services, Containern und mehr erfassen. Prometheus unterstützt eine Vielzahl von Metrikformaten und Protokollen, darunter auch das Prometheus-eigene Datenformat.

2. Datenmodell: Prometheus verwendet ein multidimensionales Datenmodell, bei dem Metriken mit Labels (Schlüssel-Wert-Paaren) versehen werden können. Dadurch wird die Möglichkeit geschaffen, Metriken nach verschiedenen Dimensionen zu segmentieren und zu analysieren. Labels ermöglichen es, Metriken nach Anwendung, Umgebung, Instanz, Region und anderen benutzerdefinierten Dimensionen zu gruppieren.

3. Abfragesprache: Prometheus verfügt über eine leistungsstarke Abfragesprache namens PromQL (Prometheus Query Language). Mit PromQL können Benutzer komplexe Abfragen und Aggregationen auf die gesammelten Metriken durchführen, um Erkenntnisse und Trends zu gewinnen. PromQL bietet auch Funktionen zur Visualisierung und Grafikdarstellung der Metriken.

4. Alarmierung: Prometheus bietet eine integrierte Alarmierungsfunktion, die es Benutzern ermöglicht, Alarmregeln zu definieren und auf Basis von Metriken Alarme auszulösen. Alarme können an verschiedene Benachrichtigungskanäle gesendet werden, wie z. B. E-Mails, PagerDuty, Slack usw.

5. Skalierbarkeit: Prometheus wurde für eine hohe Skalierbarkeit konzipiert und kann große Mengen von Metrikdaten verarbeiten. Es verwendet eine Pull-basierte Architektur, bei der Prometheus-Agenten die Metriken von den überwachten Komponenten abrufen. Prometheus kann auch horizontal skaliert werden, um den Anforderungen von großen Umgebungen gerecht zu werden.

Prometheus hat eine starke Integration mit dem Kubernetes-Ökosystem und ist eine beliebte Wahl für das Monitoring von containerisierten Umgebungen. Es bietet auch eine breite Palette von Integrationen mit anderen Tools und Plattformen, um Daten von verschiedenen Quellen zu erfassen und zu analysieren.

Durch seine Flexibilität, Skalierbarkeit und die Unterstützung für dimensionale Metriken hat sich Prometheus zu einer beliebten Lösung für das Monitoring und die Observability von Cloud-native-Anwendungen entwickelt.

8.2 Dependency

```
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-registry-prometheus</artifactId>
   <scope>runtime</scope>
</dependency>
```

Weitere Einstellung sind meist nicht notwendig. Ggf. muss in der „application.properties“ der Endpoint aktiviert werden:

```
management.endpoints.web.exposure.include=*
```

schließt Prometheus mit ein.

Der Endpoint lautet /actuator/prometheus und muss in der Security freigeschaltet sein.

Eine Curlabfrage auf den Endpoint liefert folgendes Ergebnis:

```
C:\Users\JoachimWagner>curl "http://localhost:8130/actuator/prometheus"
# HELP jvm_info JVM version info
# TYPE jvm_info gauge
jvm_info{runtime="OpenJDK Runtime Environment",vendor="Eclipse Adoptium",version="17.0.4.1+1",} 1.0
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
```

```
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space",} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen",} 8.518631424E9
jvm_memory_max_bytes{area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeCache",} 5.0331648E7
jvm_memory_max_bytes{area="heap",id="G1 Eden Space",} -1.0
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space",} 1.073741824E9
# HELP logback_events_total Number of log events that were enabled by the effective log level
# TYPE logback_events_total counter
logback_events_total{level="warn",} 0.0
logback_events_total{level="debug",} 0.0
logback_events_total{level="error",} 0.0
logback_events_total{level="trace",} 0.0
logback_events_total{level="info",} 0.0
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
tomcat_sessions_alive_max_seconds 0.0
# HELP bar_counter_total
# TYPE bar_counter_total counter
bar_counter_total 2.0
# HELP jvm_threads_live_threads The current number of live threads including both daemon and non-daemon threads
# TYPE jvm_threads_live_threads gauge
jvm_threads_live_threads 27.0
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.686903833562E9
# HELP jvm_memory_usage_after_gc_percent The percentage of long-lived heap pool used after the last GC event, in the range [0..1]
# TYPE jvm_memory_usage_after_gc_percent gauge
jvm_memory_usage_after_gc_percent{area="heap",pool="long-lived",} 0.0
# HELP executor_queue_remaining_tasks The number of additional elements that this queue can ideally accept without blocking
# TYPE executor_queue_remaining_tasks gauge
executor_queue_remaining_tasks{name="applicationTaskExecutor",} 2.147483647E9
# HELP tomcat_sessions_expired_sessions_total
# TYPE tomcat_sessions_expired_sessions_total counter
tomcat_sessions_expired_sessions_total 0.0
# HELP demo_seconds_max
# TYPE demo_seconds_max gauge
demo_seconds_max{class="de.limago.simpleappwithprometheus.controller.DemoController",exception="none",method="getOne",} 0.0032031
# HELP demo_seconds
# TYPE demo_seconds summary
demo_seconds_count{class="de.limago.simpleappwithprometheus.controller.DemoController",exception="none",method="getOne",} 1.0
demo_seconds_sum{class="de.limago.simpleappwithprometheus.controller.DemoController",exception="none",method="getOne",} 0.0032031
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="G1 Survivor Space",} 4302448.0
jvm_memory_used_bytes{area="heap",id="G1 Old Gen",} 1.9518464E7
jvm_memory_used_bytes{area="nonheap",id="Metaspace",} 4.0783016E7
jvm_memory_used_bytes{area="nonheap",id="CodeCache",} 1.0671488E7
jvm_memory_used_bytes{area="heap",id="G1 Eden Space",} 5.4525952E7
jvm_memory_used_bytes{area="nonheap",id="Compressed Class Space",} 5827976.0
# HELP executor_active_threads The approximate number of threads that are actively executing tasks
# TYPE executor_active_threads gauge
```

executor_active_threads{name="applicationTaskExecutor",} 0.0

HELP jvm_buffer_count_buffers An estimate of the number of buffers in the pool

TYPE jvm_buffer_count_buffers gauge

jvm_buffer_count_buffers{id="mapped - 'non-volatile memory'",} 0.0

jvm_buffer_count_buffers{id="mapped",} 0.0

jvm_buffer_count_buffers{id="direct",} 7.0

HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process

TYPE process_cpu_usage gauge

process_cpu_usage 0.043079585156691136

HELP tomcat_sessions_created_sessions_total

TYPE tomcat_sessions_created_sessions_total counter

tomcat_sessions_created_sessions_total 0.0

HELP http_server_requests_active_seconds_max

TYPE http_server_requests_active_seconds_max gauge

http_server_requests_active_seconds_max{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN",} 0.5408161

HELP http_server_requests_active_seconds

TYPE http_server_requests_active_seconds summary

http_server_requests_active_seconds_active_count{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN",} 1.0

http_server_requests_active_seconds_duration_sum{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN",} 0.5407943

HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use

TYPE jvm_memory_committed_bytes gauge

jvm_memory_committed_bytes{area="heap",id="G1 Survivor Space",} 8388608.0

jvm_memory_committed_bytes{area="heap",id="G1 Old Gen",} 4.194304E7

jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 4.1418752E7

jvm_memory_committed_bytes{area="nonheap",id="CodeCache",} 1.6187392E7

jvm_memory_committed_bytes{area="heap",id="G1 Eden Space",} 6.291456E7

jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",} 6094848.0

HELP http_server_requests_seconds

TYPE http_server_requests_seconds summary

http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/demo/path-1",} 1.0

http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/demo/path-1",} 0.034228

http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/health",} 1.0

http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/health",} 0.0268214

http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="CLIENT_ERROR",status="404",uri="/**",} 1.0

http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="CLIENT_ERROR",status="404",uri="/**",} 0.0062748

HELP http_server_requests_seconds_max

TYPE http_server_requests_seconds_max gauge

http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/demo/path-1",} 0.034228

http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/health",} 0.0268214

http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="CLIENT_ERROR",status="404",uri="/**",} 0.0062748

HELP jvm_threads_daemon_threads The current number of live daemon threads

TYPE jvm_threads_daemon_threads gauge

jvm_threads_daemon_threads 23.0

HELP application_ready_time_seconds Time taken (ms) for the application to be ready to service requests

TYPE application_ready_time_seconds gauge

application_ready_time_seconds{main_application_class="de.limago.simpleappwithprometheus.SimpleAppWithPrometheusApplication",} 3.141

```
# HELP executor_completed_tasks_total The approximate total number of tasks that have completed execution
# TYPE executor_completed_tasks_total counter
executor_completed_tasks_total{name="applicationTaskExecutor",} 0.0
# HELP jvm_classes_loaded_classes The number of classes that are currently loaded in the Java virtual machine
# TYPE jvm_classes_loaded_classes gauge
jvm_classes_loaded_classes 9798.0
# HELP jvm_classes_unloaded_classes_total The total number of classes unloaded since the Java virtual machine has started execution
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total 2.0
# HELP executor_queued_tasks The approximate number of tasks that are queued for execution
# TYPE executor_queued_tasks gauge
executor_queued_tasks{name="applicationTaskExecutor",} 0.0
# HELP tomcat_sessions_active_max_sessions
# TYPE tomcat_sessions_active_max_sessions gauge
tomcat_sessions_active_max_sessions 0.0
# HELP executor_pool_core_threads The core number of threads for the pool
# TYPE executor_pool_core_threads gauge
executor_pool_core_threads{name="applicationTaskExecutor",} 8.0
# HELP tomcat_sessions_rejected_sessions_total
# TYPE tomcat_sessions_rejected_sessions_total counter
tomcat_sessions_rejected_sessions_total 0.0
# HELP foo_seconds
# TYPE foo_seconds summary
foo_seconds_count{class="de.limago.simpleappwithprometheus.controller.MyComponent",exception="none",method="foo",} 3.0
foo_seconds_sum{class="de.limago.simpleappwithprometheus.controller.MyComponent",exception="none",method="foo",} 0.0021819
# HELP foo_seconds_max
# TYPE foo_seconds_max gauge
foo_seconds_max{class="de.limago.simpleappwithprometheus.controller.MyComponent",exception="none",method="foo",} 0.0021201
# HELP jvm_compilation_time_ms_total The approximate accumulated elapsed time spent in compilation
# TYPE jvm_compilation_time_ms_total counter
jvm_compilation_time_ms_total{compiler="HotSpot 64-Bit Tiered Compilers",} 3436.0
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes 0.0
# HELP jvm_gc_memory_promoted_bytes_total Count of positive increases in the size of the old generation memory pool before GC to after GC
# TYPE jvm_gc_memory_promoted_bytes_total counter
jvm_gc_memory_promoted_bytes_total 0.0
# HELP executor_pool_size_threads The current number of threads in the pool
# TYPE executor_pool_size_threads gauge
executor_pool_size_threads{name="applicationTaskExecutor",} 0.0
# HELP disk_total_bytes Total space for path
# TYPE disk_total_bytes gauge
disk_total_bytes{path="C:\\Users\\JoachimWagner\\git\\Limago\\spring-operartions\\SimpleAppWithPrometheus\\.",} 1.022389907456E12
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count 16.0
# HELP jvm_buffer_total_capacity_bytes An estimate of the total capacity of the buffers in this pool
# TYPE jvm_buffer_total_capacity_bytes gauge
jvm_buffer_total_capacity_bytes{id="mapped - 'non-volatile memory'",} 0.0
jvm_buffer_total_capacity_bytes{id="mapped",} 0.0
jvm_buffer_total_capacity_bytes{id="direct",} 57344.0
# HELP application_started_time_seconds Time taken (ms) to start the application
```

```
# TYPE application_started_time_seconds gauge
application_started_time_seconds{main_application_class="de.limago.simpleappwithprometheus.SimpleAppWithPrometheusApplication",} 3.136
# HELP executor_pool_max_threads The maximum allowed number of threads in the pool
# TYPE executor_pool_max_threads gauge
executor_pool_max_threads{name="applicationTaskExecutor",} 2.147483647E9
# HELP jvm_gc_max_data_size_bytes Max size of long-lived heap memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes 8.518631424E9
# HELP process_uptime_seconds The uptime of the Java virtual machine
# TYPE process_uptime_seconds gauge
process_uptime_seconds 112.685
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that the Java virtual machine is using for this buffer pool
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{id="mapped - 'non-volatile memory'",} 0.0
jvm_buffer_memory_used_bytes{id="mapped",} 0.0
jvm_buffer_memory_used_bytes{id="direct",} 57344.0
# HELP jvm_threads_states_threads The current number of threads
# TYPE jvm_threads_states_threads gauge
jvm_threads_states_threads{state="runnable",} 11.0
jvm_threads_states_threads{state="blocked",} 0.0
jvm_threads_states_threads{state="waiting",} 11.0
jvm_threads_states_threads{state="timed-waiting",} 5.0
jvm_threads_states_threads{state="new",} 0.0
jvm_threads_states_threads{state="terminated",} 0.0
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young) heap memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total 0.0
# HELP jvm_threads_peak_threads The peak live thread count since the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak_threads gauge
jvm_threads_peak_threads 31.0
# HELP jvm_gc_overhead_percent An approximation of the percent of CPU time used by GC activities over the last lookback period or since monitoring began, whichever is shorter, in the range [0..1]
# TYPE jvm_gc_overhead_percent gauge
jvm_gc_overhead_percent 0.0
# HELP system_cpu_usage The "recent cpu usage" of the system the application is running in
# TYPE system_cpu_usage gauge
system_cpu_usage 0.15297880414159482
# HELP jvm_threads_started_threads_total The total number of application threads started in the JVM
# TYPE jvm_threads_started_threads_total counter
jvm_threads_started_threads_total 50.0
# HELP tomcat_sessions_active_current_sessions
# TYPE tomcat_sessions_active_current_sessions gauge
tomcat_sessions_active_current_sessions 0.0
# HELP example_list_size
# TYPE example_list_size gauge
example_list_size 0.0
# HELP disk_free_bytes Usable space for path
# TYPE disk_free_bytes gauge
disk_free_bytes{path="C:\\Users\\JoachimWagner\\git\\Limago\\spring-operartions\\SimpleAppWithPrometheus\\.",} 3.46373083136E11
```

8.3 Installation Prometheus Monitor Tool

8.3.1 Downloadseite

Hier gibt es die Downloads für die diversen Betriebssysteme

https://prometheus.io/download/

und die zugehörige Installationsanweisung

8.3.2 Docker

Die Installalation mit docker-compose ist sehr einfach. Hier das Script:

```
version: '2.1'

services:

  prometheus:
    image: prom/prometheus:v2.38.0
    #network_mode: host
    container_name: prometheus-container
    volumes:
      - ./prometheus/:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yaml'
    ports:
      - "9090:9090"
    restart: always
```

Im selben Ordner sollte ein Unterordner Namens „prometheus“ angelegt werden. In diesen Ordner packt eine Datei „prometheus.yaml“ mit folgendem Inhalt:

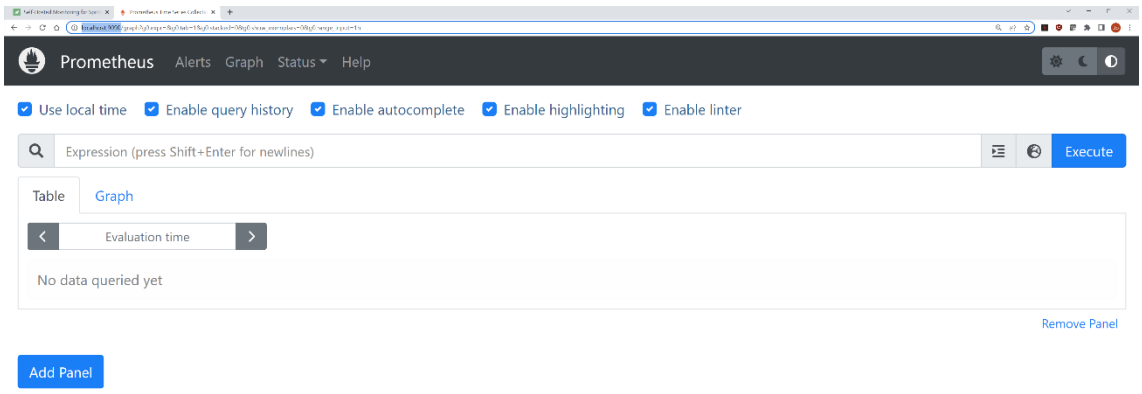
```
scrape_configs:
  - job_name: 'Spring Boot Application input'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 3s
    static_configs:
      - targets: ['host.docker.internal:8130']
        labels:
          application: 'SimpleApp'
```

8.4 Auswertung

Nach dem Start des Servers ist er unter

http://localhost:9090/

erreichbar.



In Prometheus werden Zeitreihen automatisch erstellt, wenn Metriken erfasst werden. Das Erstellen von Zeitreihen in Prometheus erfolgt in der Regel in zwei Schritten:

1. Metrikerfassung: Prometheus erfasst Metriken von den überwachten Komponenten, entweder durch direkte Integration über das Prometheus-Client-Bibliothek oder durch das Abrufen von Metriken von Exportern oder anderen Diensten, die das Prometheus-Expositionsformat unterstützen. Die Metriken werden mit Labels (Schlüssel-Wert-Paaren) versehen, die zusätzliche Dimensionen darstellen.

2. Speicherung der Zeitreihen: Prometheus speichert die erfassten Metriken als Zeitreihen in seinem internen Speicher. Jede Zeitreihe besteht aus einem eindeutigen Metriknamen und einem Satz von Label-Werten. Die Zeitreihen werden basierend auf ihren Labels und den zugehörigen Zeitstempeln in den Speicher geschrieben.

Prometheus verwendet einen speziellen Datenbanktyp, der als Zeitreihendatenbank bezeichnet wird, um die Zeitreihen effizient zu speichern und abzufragen. Die Zeitreihendatenbank organisiert die Daten so, dass sie schnell abgefragt und aggregiert werden können.

Sobald die Zeitreihen erstellt und gespeichert sind, stehen sie für Abfragen und Analysen zur Verfügung. Benutzer können PromQL verwenden, um Abfragen auf die Zeitreihen auszuführen und Metriken basierend auf verschiedenen Kriterien zu filtern, zu aggregieren und zu visualisieren. Die Ergebnisse können in Grafiken, Diagrammen oder anderen Formaten dargestellt werden, um Einblicke in das Verhalten und die Leistung der überwachten Komponenten zu gewinnen.

Prometheus speichert standardmäßig Metriken für einen bestimmten Zeitraum und verwirft ältere Daten. Die Aufbewahrungsdauer kann konfiguriert werden, um den Speicherplatzbedarf und die Anforderungen an die Datenaufbewahrung anzupassen.

8.5 PromQL

Beispiele für PromQL-Abfragen in Prometheus mit Erklärungen:

1. Abfrage nach einem bestimmten Metrikwert:

```
requests_total
```

Diese einfache Abfrage gibt den aktuellen Wert der Metrik "requests_total" zurück. Es wird keine Aggregation durchgeführt, es werden einfach alle verfügbaren Zeitreihen für diese Metrik zurückgegeben.

2. Aggregierte Abfrage mit Zeitspanne:

```
sum(rate(requests_total[5m]))
```

Diese Abfrage aggregiert den Wert der Metrik "requests_total" über einen Zeitraum von 5 Minuten. Die Funktion `rate()` berechnet die

durchschnittliche Rate des Metrikwerts pro Sekunde und die Funktion `sum()` addiert die Werte aller Zeitreihen zusammen.

3. Abfrage mit Filterung nach Label:

```
requests_total{method="GET"}
```

Diese Abfrage gibt den aktuellen Wert der Metrik "requests_total" nur für die Zeitreihen zurück, die das Label "method" mit dem Wert "GET" haben. Dadurch werden nur die Anfragen mit der HTTP-Methode "GET" berücksichtigt.

4. Aggregation über Label-Werte:

```
sum by (method) (requests_total)
```

Diese Abfrage aggregiert den Wert der Metrik "requests_total" für jedes eindeutige Label "method". Die Funktion `sum by (method)` führt eine separate Aggregation für jede eindeutige Methode durch und gibt den Gesamtwert für jede Methode zurück.

5. Zeitliche Verschiebung von Metriken:

```
rate(requests_total[5m]) offset 1h
```

Diese Abfrage berechnet die durchschnittliche Rate der Metrik "requests_total" über einen Zeitraum von 5 Minuten. Mit `offset 1h` wird die Abfrage um eine Stunde in die Vergangenheit verschoben. Dadurch wird der Metrikwert vor einer Stunde zurückgegeben.

Dies sind nur einige Beispiele für PromQL-Abfragen in Prometheus. PromQL bietet eine Vielzahl von Funktionen und Operatoren, mit denen komplexe Abfragen, Aggregationen, Filterungen und Transformationen auf Metrikdaten durchgeführt werden können. Durch das Kombinieren dieser Funktionen können Sie detaillierte Einblicke in das Verhalten und die Leistung Ihrer Anwendung oder Ihres Systems gewinnen. Es ist ratsam, die PromQL-Dokumentation zu konsultieren, um das volle Potenzial dieser Abfragesprache zu erkunden.

8.6 Erweiterungen für Grafana

8.6.1 Installation

Für die Anbindung an Grafana wird ein Node Explorer benötigt. Download hier: https://github.com/prometheus/node_exporter

Der Prometheus Node Exporter kann auf verschiedenen Betriebssystemen installiert werden. Hier sind einige gängige Installationsmethoden für verschiedene Plattformen:

1. Linux (Systemd):

- Laden Sie das neueste Release des Node Exporters von der offiziellen Prometheus GitHub-Seite herunter: https://github.com/prometheus/node_exporter/releases

- Extrahieren Sie das Archiv und wechseln Sie in das extrahierte Verzeichnis.

- Führen Sie den Befehl `sudo ./node_exporter` aus, um den Node Exporter zu starten. Dadurch wird der Exporter im Hintergrund als Systemd-Dienst gestartet.

2. Linux (Docker):

- Führen Sie den folgenden Befehl aus, um den Node Exporter als Docker-Container zu starten:

```
docker run -d -p 9100:9100 --name=node_exporter quay.io/prometheus/node-exporter
```

3. Windows:

- Laden Sie das neueste Release des Node Exporters von der offiziellen Prometheus GitHub-Seite herunter: https://github.com/prometheus/node_exporter/releases

- Extrahieren Sie das Archiv und wechseln Sie in das extrahierte Verzeichnis.

- Starten Sie den Node Exporter, indem Sie die `node_exporter.exe`-Datei ausführen.

Nach der Installation des Node Exporters sollte er standardmäßig auf dem Port 9100 laufen und Metriken über den `/metrics`-Endpunkt

bereitstellen. Sie können dies überprüfen, indem Sie `http://localhost:9100/metrics` in einem Webbrowser öffnen. Wenn die Metriken angezeigt werden, ist der Node Exporter erfolgreich installiert und läuft.

8.6.2 Konfiguration

Um den Node Exporter in Prometheus zu integrieren, müssen Sie die Konfigurationsdatei von Prometheus anpassen und den Node Exporter als Ziel hinzufügen. In der Konfigurationsdatei (`prometheus.yml`) können Sie eine neue `scrape_config`-Sektion hinzufügen, um den Node Exporter als `target` zu definieren. Beispiel:

```
scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['localhost:9100']
```

Speichern Sie die Konfigurationsdatei und starten Sie Prometheus neu. Danach sollte Prometheus beginnen, Metriken vom Node Exporter zu erfassen.

9

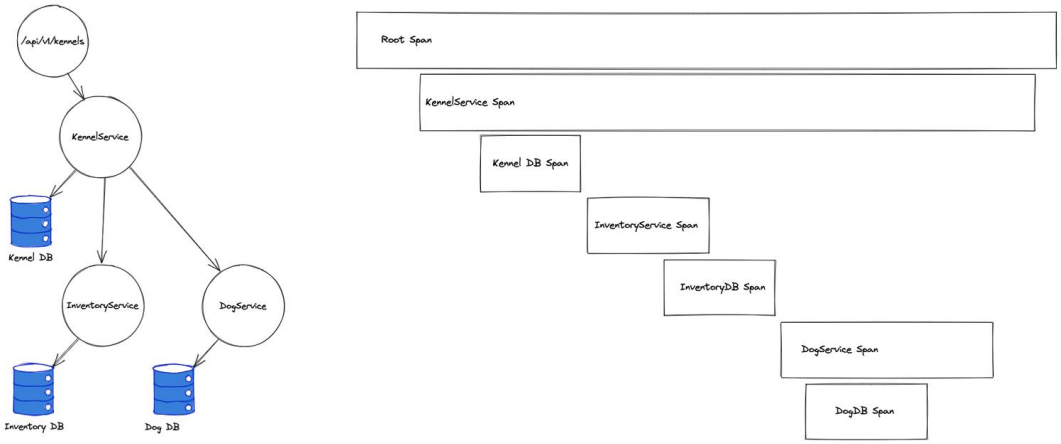
OpenTelemetry und Jaeger

9 OpenTelemetry und Jaeger

9.1 OpenTelemetry

9.2 Übersicht

OpenTelemetry ist ein Open-Source-Projekt, das entwickelt wurde, um Entwicklern eine standardisierte Möglichkeit zur Instrumentierung, Überwachung und Tracing von Anwendungen zu bieten. Es vereinfacht die Erfassung, das Sammeln und das Exportieren von Daten über verschiedene Komponenten einer Anwendung hinweg.



OpenTelemetry bietet Client-Bibliotheken und Agenten für die meisten gängigen Programmiersprachen. Es gibt zwei Arten von Instrumenten:

Automatische Instrumentierung

OpenTelmetry kann Daten für viele gängige Frameworks und Bibliotheken automatisch sammeln. Sie müssen keine Codeänderungen vornehmen.

Manuelle Instrumentierung

Wenn Sie anwendungsspezifischere Daten benötigen, bietet Ihnen das OpenTelemetry SDK die Möglichkeit, diese Daten mithilfe von OpenTelemetry-APIs und -SDKs zu erfassen.

Für Spring Boot-Anwendungen können wir den OpenTelemetry Java Jar-Agenten (Wir müssen lediglich die neueste Version des Java-Jar-Agenten herunterladen und die Anwendung damit ausführen) oder Maven- Dependencies verwenden.

Das OpenTelemetry-Projekt vereint die beiden bereits bestehenden Projekte OpenTracing und OpenCensus. Es stellt eine einheitliche und erweiterte Plattform zur Verfügung, die die Tracing- und Metrikerfassung in einer einzigen Lösung kombiniert. Dadurch wird die Implementierung und Integration von Tracing- und Metrikdaten in Anwendungen erleichtert.

OpenTelemetry bietet auch Integrationen mit verschiedenen Cloud-Anbietern, Observability-Plattformen und Tools zur weiteren Verarbeitung und Analyse von Überwachungsdaten. Es ermöglicht die nahtlose Integration von Instrumentierungs- und Überwachungsdaten in bestehende Monitoring- und Analysewerkzeuge.

9.2.1 Einrichtung

Application.properties

```
#openTelemetry
#tracing.url=http://localhost:4318/v1/traces

management.tracing.sampling.probability= 1.0

logging.pattern.level= '%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]'
```

OpenTelemetry pusht die Daten dann an http://localhost:4318/v1/traces

9.2.2 Code zur Demonstration

```
@GetMapping(path = "path-2", produces = MediaType.TEXT_PLAIN_VALUE)
// @Timed(value = "example.metric.name", Perzentile = { 0,95, 0,75 })
// https://medium.com/clarityai-engineering/effectively-measuring-execution-times-with-micrometer-datadog-5ad15fb8abee
```

```
@Timed(value = "demo.getsecond", description = "eine doofe Beschreibung")
public String getSecond() throws Exception{
    log.info("get_one");
    Thread.sleep(20);
    return "path-2";
}
@GetMapping(path = "path-3", produces = MediaType.TEXT_PLAIN_VALUE)
// @Timed(value = "example.metric.name", Perzentile = { 0,95, 0,75 })
// https://medium.com/clarityai-engineering/effectively-measuring-execution-times-with-micrometer-datadog-5ad15fb8abee
@Timed(value = "demo.getthird", description = "eine doofe Beschreibung")
public String getThird() throws Exception{
    log.info("get_third");
    String result = restTemplate.getForObject("http://localhost:8130/demo/path-2", String.class);
    return "path-3 enriched by "  + result;
}
```

Die untere Methode verwendet die obere Methode mit einer Wartezeit von ca. 20 Millisekunden. Die Aufrufkette soll verfolgt werden.

Dazu muss zunächst eine UI installiert werden. In diesem Fall der Service Jaeger UI.

9.3 Jaeger UI

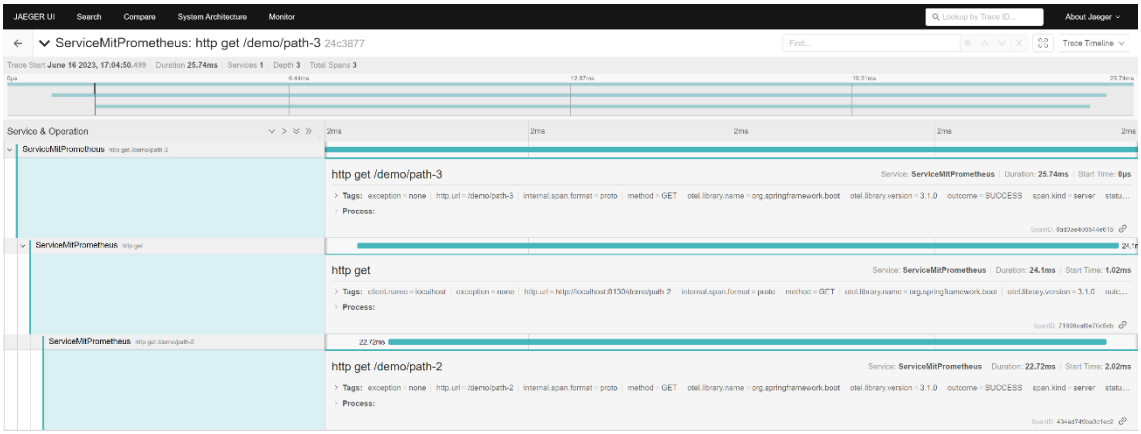
9.3.1 Installation

Auch hier kann ein Download für diverse BS erfolgen. In diesem Beispiel wird wieder Docker verwendet:

```
version: '3.9'
services:
  jaeger:
    image: jaegertracing/all-in-one:latest
    ports:
      - 4318:4318
      - 16686:16686
    environment:
      - COLLECTOR_OTLP_ENABLED=true
```

9.3.2 Anzeigen der Verfolgung

Mit dem Curl-Befehl „curl "http://localhost:8130/demo/path-3"“wird die Ausführung des zu messenden Requests angestossen.



Der Trace wird wie erwartet angezeigt,

10

Tempo

10 Tempo

10.1 Übersicht

Tempo ist ein verteiltes Tracing-System, das auf der OpenTelemetry-Technologie basiert. Es ermöglicht das Sammeln, Speichern und Abfragen von Traces in großem Maßstab. Tempo ist für die Skalierung konzipiert und unterstützt hohe Lasten von Traces, um eine umfassende Analyse und Überwachung der verteilten Anwendungen zu ermöglichen. Es bietet eine effiziente Speicherung und Abfrage von Traces und kann nahtlos in die Prometheus-Grafana-Oberfläche integriert werden. Tempo unterstützt die gängigen OpenTelemetry-Standards und ermöglicht die Erfassung von Traces aus verschiedenen Quellen.

10.2 Unterschied zu Jaeger

Jaeger und Tempo sind beides Open-Source-Projekte für die Tracing-basierte Observability, jedoch gibt es einige Unterschiede zwischen den beiden:

1. Architektur: Jaeger verwendet eine zentralisierte Architektur, bei der ein zentraler Jaeger-Server alle Tracing-Daten empfängt, speichert und verarbeitet. Es können mehrere Jaeger-Agenten eingesetzt werden, um die Daten von den Anwendungen zu sammeln und an den Jaeger-Server zu senden. Tempo hingegen verwendet eine verteilte Architektur, bei der Tracing-Daten dezentral erfasst und an verschiedene Tempo-Knoten (Receivers) gesendet werden. Diese Knoten speichern und indizieren die Daten und ermöglichen eine effiziente Suche und Analyse.

2. Skalierbarkeit: Aufgrund seiner verteilten Architektur bietet Tempo eine höhere Skalierbarkeit und bessere Leistung bei großen Datenmengen. Es kann leicht horizontal skaliert werden, indem zusätzliche Tempo-Knoten hinzugefügt werden. Jaeger hat eine zentralisierte Architektur, die bei hoher Last und großen Datenmengen an ihre Grenzen stoßen kann.

3. Speicherung: Jaeger verwendet eine eigene Speicherlösung, normalerweise basierend auf Cassandra oder Elasticsearch, um Tracing-Daten zu speichern. Tempo hingegen unterstützt verschiedene Backends für die Speicherung, darunter auch Prometheus und Loki. Dies ermöglicht eine nahtlose Integration mit anderen Observability-Tools und erweitert die Möglichkeiten der Datenanalyse und -visualisierung.

4. Einfachheit und Wartung: Tempo wurde entwickelt, um eine einfachere und wartungsfreundlichere Lösung für das Tracing-Management bereitzustellen. Es konzentriert sich auf die Kernfunktionalität des Tracings und eliminiert einige der Komplexitäten, die mit einer vollständigen Jaeger-Implementierung verbunden sind. Tempo bietet eine schlankere und effizientere Option für die Skalierung und Verwaltung von Tracing-Daten.

Insgesamt kann man sagen, dass Tempo eine Weiterentwicklung des Jaeger-Projekts ist und eine verteilte, hoch skalierbare und wartungsfreundliche Lösung für das Tracing bietet. Es ist speziell für den Einsatz in Umgebungen mit hohem Datenvolumen konzipiert. Jaeger hingegen ist eine etablierte Lösung für das Tracing und bietet zusätzliche Funktionen und Erweiterbarkeit, kann jedoch bei großen Datenmengen und Skalierungsanforderungen an ihre Grenzen stoßen. Die Wahl zwischen Jaeger und Tempo hängt von den spezifischen Anforderungen und der Größe der Tracing-Daten ab.

10.3 Tempo und Graphana

Tempo selbst verfügt über keine eigene GUI (grafische Benutzeroberfläche). Stattdessen kann Tempo nahtlos in Grafana integriert werden, einem leistungsstarken Observability-Dashboard-Tool, welches später beschrieben wird.

Grafana bietet eine umfangreiche Benutzeroberfläche zur Visualisierung und Analyse von Daten aus verschiedenen Quellen, einschließlich Traces von Tempo. Durch die Integration von Tempo in Grafana können Benutzer Traces anzeigen, filtern, aggregieren und analysieren, um Einblicke in die Leistung und das Verhalten ihrer Anwendungen zu gewinnen.

Insgesamt ermöglicht die Kombination von OpenTelemetry und Tempo eine umfassende Observability von Anwendungen. OpenTelemetry sorgt für die Erfassung von Traces und anderen Observability-Daten in Anwendungen, während Tempo die Infrastruktur zur Speicherung und Abfrage der Traces bereitstellt. Grafana dient als Oberfläche zur Visualisierung und Analyse der Traces und anderer Observability-Daten. Zusammen bieten sie eine leistungsstarke Lösung zur Überwachung und Fehlerbehebung von Anwendungen.

10.4 TraceQL

TraceQL ist eine Abfragesprache, die speziell für die Abfrage und Analyse von Traces in Tempo entwickelt wurde. Mit TraceQL können Entwickler und Systemadministratoren komplexe Abfragen auf Tracedaten ausführen, um Einblicke in die Leistung und das Verhalten ihrer Anwendungen zu gewinnen.

TraceQL ermöglicht die Filterung und Aggregation von Tracedaten anhand verschiedener Kriterien wie Tracernamen, Operationen, Tags und Zeitspannen. Hier sind einige Beispiele, wie TraceQL verwendet werden kann:

1. Filterung nach einem bestimmten Tracernamen:

```
traces from service_name="my-service"
```

Diese Abfrage gibt alle Traces zurück, die den Tracernamen "my-service" haben.

2. Filterung nach einer bestimmten Operation:

```
traces where operation="getUser"
```

Diese Abfrage gibt alle Traces zurück, bei denen die Operation "getUser" ist.

3. Filterung nach Tags:

```
traces with tags.env="production" and tags.status="error"
```

Diese Abfrage gibt alle Traces zurück, die den Tag "env" mit dem Wert "production" und den Tag "status" mit dem Wert "error" haben.

4. Aggregation von Tracedaten:

```
avg(duration) by service_name
```


Diese Abfrage berechnet den Durchschnitt der Dauer für jeden Dienst und gibt die Ergebnisse zurück.

10.5 Begriffe

Bei Tempo, einem verteilten Tracing-System von Grafana, gibt es einige wichtige Begriffe, die das Verständnis des Systems und seiner Funktionalität erleichtern. Hier sind einige wichtige Begriffe bei Tempo:

1. Ingester: Der Ingester ist eine Komponente in Tempo, die für das Empfangen, Verarbeiten und Speichern von Tracedaten zuständig ist. Der Ingester akzeptiert Tracedaten von Instrumentierungsbibliotheken oder Tracing-Agenten und stellt sicher, dass sie in einem geeigneten Format gespeichert werden, um sie für die spätere Analyse verfügbar zu machen.

2. Store: Der Store ist die persistente Datenspeicherungskomponente von Tempo. Hier werden die empfangenen Tracedaten langfristig gespeichert, um sie für Abfragen und Analysen verfügbar zu machen. Der Store ermöglicht das effiziente Speichern und Abrufen von Tracedaten in einem skalierbaren und zuverlässigen System.

3. Index: Der Index ist eine Komponente, die den Zugriff auf die gespeicherten Tracedaten optimiert. Er ermöglicht das schnelle Durchsuchen und Abrufen von Traces anhand verschiedener Kriterien wie Tracernamen, Operationen oder Zeitbereichen. Der Index verbessert die Leistung bei der Abfrage von Tracedaten und unterstützt eine effiziente Analyse.

4. Query Service: Der Query Service ist eine Komponente in Tempo, die für das Ausführen von Abfragen auf die gespeicherten Tracedaten zuständig ist. Er ermöglicht es Benutzern, TraceQL-Abfragen durchzuführen, um spezifische Informationen aus den Tracedaten zu extrahieren. Der Query Service verarbeitet die Abfragen effizient und liefert die Ergebnisse an den Benutzer zurück.

5. Instrumentierung: Die Instrumentierung bezieht sich auf den Prozess des Hinzufügens von Code zu Anwendungen, um Tracedaten zu erfassen. Durch die Instrumentierung werden Traces in der Anwendung generiert und an Tempo gesendet, um eine umfassende Sichtbarkeit und Verfolgbarkeit zu ermöglichen.

6. Tracernamen: Der Tracername ist ein eindeutiger Bezeichner für einen Trace oder eine Sequenz von Ereignissen innerhalb einer Anwendung. Traces mit demselben Tracernamen gehören normalerweise zu einer bestimmten Anfrage oder Transaktion und können zur Gruppierung und Analyse verwendet werden.

11

LOKI

11 LOKI

11.1 Übersicht

LOKI ist ein Open-Source-System für Protokollaggregation und -analyse, das von Grafana entwickelt wurde. Es wurde speziell für die Bewältigung großer Mengen von Protokolldaten entwickelt, die von verteilten Anwendungen und Infrastrukturkomponenten generiert werden.

In einer verteilten Anwendungslandschaft, in der mehrere Microservices und Komponenten miteinander interagieren, wird eine effektive Protokollaggregation und -analyse immer wichtiger. LOKI bietet hier eine Lösung, indem es Protokolldaten von verschiedenen Quellen sammelt, indiziert und speichert.

Durch die Aggregation von Protokolldaten von verschiedenen verteilten Anwendungen ermöglicht LOKI eine zentrale Sichtbarkeit und Analyse der Protokolldaten. Dies ist besonders hilfreich, um Leistungsprobleme, Fehlerzustände oder ungewöhnliches Verhalten in der gesamten verteilten Anwendungslandschaft zu identifizieren und zu debuggen.

LOKI verwendet eine effiziente Speicherstruktur und Komprimierungstechniken, um große Mengen von Protokolldaten effektiv zu verwalten. Dadurch können Entwickler und Systemadministratoren schnell auf relevante Protokollereignisse zugreifen und diese analysieren, ohne von einer Flut an Daten überwältigt zu werden.

Ein weiterer Vorteil von LOKI für verteilte Anwendungen ist die Integration mit anderen Grafana-Produkten wie Grafana Dashboards und Grafana Explore. Dadurch können Benutzer Protokolldaten in Echtzeit visualisieren, benutzerdefinierte Dashboards erstellen und Metriken überwachen, um die Leistung und das Verhalten der verteilten Anwendungen zu überwachen und zu optimieren.

Insgesamt ermöglicht LOKI eine effektive Protokollaggregation und -analyse in verteilten Anwendungen, was zu einer verbesserten Fehlersuche, schnelleren Reaktionszeiten und einer besseren Leistung der Anwendungen führt. Es hilft Entwicklern und Betreibern, das Verhalten der Anwendungen zu verstehen, Engpässe zu identifizieren und die Zuverlässigkeit und Effizienz der verteilten Systeme zu verbessern.

11.2 Installation des Treibers

Die angegebene Befehlszeile dient dazu, den LOKI-Treiber für Docker zu installieren und als Plugin zu konfigurieren.

1. `docker plugin install grafana/loki-docker-driver:latest`: Dieser Teil des Befehls lädt das Docker-Plugin für den LOKI-Treiber von der offiziellen Repository-Quelle "grafana/loki-docker-driver" herunter und installiert es auf dem Docker-Host.

2. `--alias loki`: Mit dieser Option wird ein Alias "loki" für den LOKI-Treiber festgelegt. Der Alias ermöglicht es, den Treiber mit einem kürzeren Namen zu referenzieren, was die Verwendung in weiteren Docker-Befehlen erleichtert.

3. `--grant-all-permissions`: Diese Option gewährt dem LOKI-Treiber alle erforderlichen Berechtigungen, um mit dem Docker-Daemon zu interagieren und Protokolldaten effektiv zu sammeln.

Nachdem der Befehl erfolgreich ausgeführt wurde, ist der LOKI-Treiber als Docker-Plugin installiert und einsatzbereit. Der Treiber ermöglicht es Docker-Containern, ihre Protokolldaten direkt an LOKI zu senden, ohne zusätzlichen Konfigurationsaufwand innerhalb der Container selbst.

Durch die Verwendung des LOKI-Treibers in Verbindung mit Docker können Protokolldaten von den Containern zentralisiert und effizient erfasst werden. Der LOKI-Treiber fungiert als eine Art Proxy zwischen den Containern und dem LOKI-System, das die Protokolldaten speichert und analysiert. Dadurch wird die Konfiguration und Verwaltung der Protokollierung in Docker-Containern vereinfacht und die Effizienz der Protokolldatenverarbeitung verbessert.

Es ist wichtig zu beachten, dass der oben genannte Befehl abhängig von der Docker-Version, den Berechtigungen des Benutzers und anderen Umgebungsvariablen variieren kann. Es wird empfohlen, die offizielle Dokumentation des LOKI-Treibers und die spezifischen Anforderungen für die Installation und Konfiguration zu überprüfen.

11.3 Treiberkonfiguration

In diesem Auszug handelt es sich um YAML-Konfiguration für das Logging mit dem LOKI-Treiber. Hier ist eine Erläuterung der einzelnen Abschnitte:

- `x-logging: &default-logging`: Dies ist ein YAML-Alias (`&default-logging`), der verwendet wird, um die Konfiguration für das Logging zu definieren. Durch die Verwendung eines Alias kann diese Konfiguration an verschiedenen Stellen wiederverwendet werden.

- `driver: loki`: Hier wird der Treiber für das Logging angegeben. In diesem Fall wird der LOKI-Treiber verwendet, um Protokolldaten zu erfassen und an ein LOKI-System zu senden.

- `options`: Dieser Abschnitt enthält verschiedene Optionen für die Konfiguration des LOKI-Treibers.

- `loki-url: 'http://localhost:3100/api/prom/push'`: Hier wird die URL angegeben, unter der das LOKI-System erreichbar ist. Protokolldaten werden an diese URL gesendet, um in das LOKI-System eingefügt zu werden.

- `labels: namespace`: Hier wird angegeben, welche Labels (Kennzeichnungen) den Protokolldaten hinzugefügt werden sollen. In diesem Fall wird das Label "namespace" verwendet. Labels ermöglichen es, Protokolldaten mit zusätzlichen Metadaten zu versehen, die bei der Analyse und Filterung der Daten hilfreich sein können.

- `mode: non-blocking`: Hier wird der Modus für das Logging angegeben. In diesem Fall wird der nicht blockierende Modus verwendet, was bedeutet, dass das Logging asynchron erfolgt und den Hauptfluss der Anwendung nicht blockiert.

- `max-buffer-size: 4m`: Hier wird die maximale Puffergröße für die Protokolldaten angegeben. In diesem Fall ist die maximale Puffergröße auf 4 Megabyte (4m) festgelegt. Wenn der Puffer diese Größe erreicht, werden die Protokolldaten an das LOKI-System gesendet.

Durch diese Einstellungen zusammen mit der folgenden wird verhindert, das Graphana beim Herunterfahren des Containers

blockiert. Der Docker Stop – Befehl kann sonst einige Minuten dauern.

Die Zeile `loki-retries: "3"` in der Konfiguration bezieht sich auf die Anzahl der Wiederholungsversuche für das Senden von Protokolldaten an das LOKI-System.

Der Wert "3" gibt an, dass bei einem fehlgeschlagenen Versuch, die Protokolldaten an das LOKI-System zu senden, insgesamt drei Wiederholungsversuche unternommen werden sollen. Das bedeutet, dass das System bei einem Fehler beim Senden der Protokolldaten zwei weitere Male versucht, die Daten erneut zu senden, um sicherzustellen, dass sie erfolgreich an das LOKI-System übertragen werden.

Die Anzahl der Wiederholungsversuche kann je nach den Anforderungen und der Zuverlässigkeit des LOKI-Systems angepasst werden. Eine höhere Anzahl von Wiederholungsversuchen erhöht die Wahrscheinlichkeit, dass die Protokolldaten erfolgreich übertragen werden, aber es kann auch zu längeren Verzögerungen bei der Protokollierung führen. Es ist wichtig, das richtige Gleichgewicht zwischen Zuverlässigkeit und Performance zu finden, um die Anforderungen der Anwendung zu erfüllen.

Durch die Konfiguration der Wiederholungsversuche können potenzielle Netzwerkprobleme oder vorübergehende Ausfälle des LOKI-Systems abgefangen werden, um sicherzustellen, dass die Protokolldaten letztendlich erfolgreich an das Ziel übertragen werden.

11.4 LogQL

LogQL ist eine Abfragesprache, die speziell für die Abfrage von Protokolldaten in LOKI entwickelt wurde. Mit LogQL können komplexe Abfragen über Protokolldaten ausgeführt werden, um relevante Informationen zu extrahieren. Die Syntax von LogQL ähnelt der von SQL, was die Abfrageerstellung erleichtert. Sie können nach Protokollnachrichten, Labels, Zeitbereichen und anderen Kriterien filtern. Beispiele für LogQL-Abfragen sind `"{app=\"myapp\"}"` (filtert nach Protokolleinträgen mit einem bestimmten Label) und `"{level=\"error\"} |= \"timeout\""` (filtert nach Protokolleinträgen mit einem bestimmten Label oder einer bestimmten Nachricht). LogQL ermöglicht es Entwicklern und Systemadministratoren, Protokolldaten effektiv zu analysieren und

relevante Informationen zu gewinnen, um Fehler zu beheben, Performanceprobleme zu identifizieren und das Verhalten von Anwendungen und Systemen zu verstehen.

11.5 Notwendige Einstellungen in Spring Boot

11.5.1 Dependency

```
<dependency>
   <groupId>com.github.loki4j</groupId>
   <artifactId>loki-logback-appender</artifactId>
   <version>1.4.1</version>
</dependency>
```

11.5.2 lockback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <include resource="org/springframework/boot/logging/logback/base.xml" />
   <springProperty scope="context" name="appName" source="spring.application.name"/>

   <appender name="LOKI" class="com.github.loki4j.logback.Loki4jAppender">
      <http>
         <url>http://localhost:3100/loki/api/v1/push</url>
      </http>
      <format>
         <label>
            <pattern>app=${appName},host=${HOSTNAME},traceID=%X{traceId:-NONE},level=%level</pattern>
         </label>
         <message>
            <pattern>${FILE_LOG_PATTERN}</pattern>
         </message>
         <sortByTime>true</sortByTime>
      </format>
   </appender>

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

12

Graphite

12 Graphite

12.1 Übersicht

Graphite ist ein Open-Source-Tool zur Überwachung und Protokollierung von Metriken. Es bietet eine skalierbare und flexible Lösung für die Erfassung, Speicherung und Visualisierung von Metrikdaten.

1. Zeitreihendatenbank: Graphite verwendet eine spezialisierte Zeitreihendatenbank, die es ermöglicht, große Mengen an Metriken über einen längeren Zeitraum zu speichern. Dadurch können historische Metrikdaten analysiert und Trendmuster erkannt werden.

2. Skalierbarkeit: Graphite ist in der Lage, mit einer hohen Anzahl von Metriken umzugehen und diese effizient zu verarbeiten. Es unterstützt die horizontale Skalierung, sodass Sie Graphite-Instanzen hinzufügen können, um die Last zu verteilen und die Leistung zu verbessern.

3. Flexible Metrikaggregation: Graphite bietet verschiedene Aggregationsfunktionen, mit denen Metrikdaten auf verschiedene Arten zusammengefasst werden können. Sie können beispielsweise Metriken über verschiedene Intervalle aggregieren, um einen Überblick über kurzfristige oder langfristige Trends zu erhalten.

4. Grafische Darstellung: Graphite stellt eine Vielzahl von Diagrammtypen zur Verfügung, um Metrikdaten visuell darzustellen. Sie können Linien-, Balken- und Flächendiagramme erstellen, um Muster und Abweichungen in den Metriken leichter zu erkennen.

5. Abfrage- und Filtermöglichkeiten: Mit Graphite können Sie Metriken anhand von Tags, Mustern und Filtern abfragen und filtern. Dadurch können Sie spezifische Metriken extrahieren und detaillierte Analysen durchführen.

6. Integration mit anderen Tools: Graphite kann nahtlos mit anderen Monitoring- und Logging-Tools integriert werden. Es unterstützt verschiedene Protokolle und APIs, um Metriken von verschiedenen Quellen zu erfassen und anzuzeigen.

12.2 Einbindung in Spring Boot

12.2.1 application.properties

```
management.metrics.export.graphite.step=1s
#management.metrics.export.graphite.host=127.0.0.1
#management.metrics.export.graphite.port=2004
```

12.2.2 Dependency

```
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-registry-graphite</artifactId>
</dependency>
```

13

Grafana

13 Grafana

13.1 Übersicht

Grafana ist eine Open-Source-Plattform für die Visualisierung und Analyse von Daten. Sie bietet eine benutzerfreundliche Oberfläche zur Erstellung von Dashboards, Diagrammen und grafischen Darstellungen, um Daten aus verschiedenen Quellen zu visualisieren und zu überwachen.

Mit Grafana können Benutzer Daten aus unterschiedlichen Datenquellen wie Datenbanken, Zeitreihendatenbanken, Überwachungssystemen und Protokolldateien abrufen und visualisieren. Es unterstützt eine Vielzahl von Datenquellen, darunter InfluxDB, Prometheus, Graphite, Elasticsearch und viele andere.

13.2 Merkmale

Die wichtigsten Merkmale von Grafana sind:

1. Dashboard-Erstellung: Grafana ermöglicht die Erstellung ansprechender Dashboards, auf denen mehrere Diagramme, Messwerte und Textelemente angezeigt werden können. Benutzer können Diagramme nach Bedarf anpassen, Achsenbeschriftungen, Farben und andere visuelle Eigenschaften ändern.

2. Datenquellenintegration: Grafana kann Daten aus verschiedenen Datenquellen abrufen und darstellen. Es bietet spezielle Konnektoren und Plugins für gängige Datenquellen wie InfluxDB, Prometheus, Graphite, Elasticsearch und viele andere.

3. Abfrage-Editor: Grafana enthält einen integrierten Abfrage-Editor, mit dem Benutzer Abfragen und Filter erstellen können, um spezifische Daten aus den Datenquellen abzurufen. Es unterstützt Abfragesprachen wie PromQL für Prometheus oder SQL für relationale Datenbanken.

4. Benutzerdefinierte Panels und Plugins: Grafana ermöglicht die Integration benutzerdefinierter Panels und Plugins, um zusätzliche

Funktionalitäten hinzuzufügen oder spezifische Anforderungen zu erfüllen. Benutzer können Plugins aus einer umfangreichen Sammlung von Community- und Entwicklerressourcen installieren.

5. Alarmierung und Benachrichtigungen: Grafana unterstützt die Einrichtung von Warnungen und Benachrichtigungen, um Benutzer über kritische Ereignisse oder Schwelleüberschreitungen zu informieren. Benutzer können Regeln und Schwellenwerte definieren und die Benachrichtigungen per E-Mail, Slack oder andere Kanäle erhalten.

Grafana wird häufig in DevOps-, IT-Operations- und Überwachungsszenarien eingesetzt, um Daten aus verschiedenen Quellen zu visualisieren und Echtzeit-Einblicke in die Leistung und Verfügbarkeit von Systemen zu erhalten. Es bietet eine flexible und anpassbare Plattform für die Datenvisualisierung und ist sowohl für kleine als auch für große Umgebungen geeignet.

13.3 Plug-Ins

Plugins sind Erweiterungen oder Zusatzmodule, die in verschiedene Softwareanwendungen integriert werden können, um zusätzliche Funktionen oder Integrationen bereitzustellen. Hier ein paar Beispiele für die von uns verwendeten Technologien:

1. Prometheus Plugins:

- Storage Plugins: Prometheus unterstützt verschiedene Storage Plugins, die alternative Speicheroptionen für die Metriken bieten. Ein Beispiel ist das Prometheus Remote Storage Plugin, das es ermöglicht, Metriken an externe Speicherlösungen wie InfluxDB oder Cortex zu senden.

- Exporter Plugins: Prometheus Exporter Plugins erweitern die Fähigkeit von Prometheus, Metriken von verschiedenen Quellen zu sammeln. Es gibt eine Vielzahl von Exporter Plugins für verschiedene Anwendungen und Systeme, z.B. den Node Exporter für das Sammeln von Systemmetriken oder den MySQL Exporter für das Sammeln von Metriken aus einer MySQL-Datenbank.

2. Tempo Plugins:

- Instrumentation Plugins: Tempo unterstützt Instrumentation Plugins, die es ermöglichen, Traces aus verschiedenen Quellen zu sammeln und

zu verarbeiten. Ein Beispiel ist das OpenTelemetry Plugin, das es ermöglicht, Traces aus Anwendungen zu erfassen und an Tempo zu senden.

- Storage Plugins: Tempo bietet Storage Plugins, die verschiedene Speicheroptionen für Traces unterstützen. Ein Beispiel ist das Tempo File Storage Plugin, das Traces auf der Festplatte speichert.

3. Loki Plugins:

- Logging Plugins: Loki unterstützt verschiedene Logging Plugins, die es ermöglichen, Protokolldaten aus verschiedenen Quellen zu sammeln und zu verarbeiten. Ein Beispiel ist das Loki Promtail Plugin, das Protokolldaten von Anwendungen und Systemen sammelt und an Loki sendet.

- Storage Plugins: Loki bietet Storage Plugins, die alternative Speicheroptionen für Protokolldaten bieten. Ein Beispiel ist das Loki S3 Plugin, das Protokolldaten in einem Amazon S3-Bucket speichert.

4. Graphite Plugins:

- Data Source Plugins: Graphite unterstützt verschiedene Datenquelle Plugins, die es ermöglichen, Metriken von verschiedenen Quellen abzurufen. Ein Beispiel ist das Graphite InfluxDB Plugin, das es ermöglicht, Metriken aus einer InfluxDB-Datenbank abzurufen.

- Render Plugins: Graphite bietet Render Plugins, die die Art und Weise steuern, wie Metriken visualisiert und gerendert werden. Ein Beispiel ist das Graphite D3 Plugin, das erweiterte Diagramme und Visualisierungen für Metriken bereitstellt.

Diese Plugins erweitern die Funktionalität der jeweiligen Anwendungen und ermöglichen es den Benutzern, zusätzliche Datenquellen zu integrieren, Daten zu speichern, spezifische Protokolle oder Metriken zu sammeln und spezifische Visualisierungsoptionen zu nutzen. Sie bieten Flexibilität und Anpassbarkeit für spezifische Anforderungen und ermöglichen es den Benutzern, die Anwendungen nach ihren Bedürfnissen zu erweitern.

13.4 Explore und Dashboard

"Explore" und "Dashboard" sind zwei wichtige Funktionen in Grafana, die zur Datenvisualisierung und -analyse verwendet werden.

1. Explore:

Die Funktion "Explore" ermöglicht es Benutzern, Daten aus verschiedenen Datenquellen zu durchsuchen, Abfragen zu erstellen und Ergebnisse in Echtzeit anzuzeigen. Mit "Explore" können Benutzer schnell auf ihre Daten zugreifen und diese interaktiv analysieren, um Erkenntnisse zu gewinnen. Die Funktion unterstützt verschiedene Datenquellen wie Prometheus, InfluxDB, Elasticsearch und viele andere. Benutzer können Abfragen erstellen, Filter anwenden, Diagramme generieren und Daten in verschiedenen Formaten anzeigen.

2. Dashboard:

Das Dashboard ist ein benutzerdefinierter Bildschirm in Grafana, auf dem verschiedene Diagramme, Messwerte und visuelle Elemente angezeigt werden können. Benutzer können Dashboards erstellen und anpassen, um ihre Daten in einem einzigen übersichtlichen Bildschirm zu visualisieren. Dashboards können aus verschiedenen Paneltypen bestehen, z.B. Zeitreihendiagramme, Balkendiagramme, Tabellen usw. Benutzer können Diagramme nach Bedarf anpassen, Achsenbeschriftungen, Farben und andere visuelle Eigenschaften ändern. Dashboards können auch verschiedene Zeitspannen und Filter unterstützen, um Daten basierend auf bestimmten Kriterien anzuzeigen.

Der Hauptunterschied zwischen "Explore" und "Dashboard" liegt in ihrer Verwendung und ihrem Zweck. "Explore" konzentriert sich auf die interaktive Exploration von Daten, während "Dashboard" darauf abzielt, Daten in einem strukturierten und anpassbaren Format anzuzeigen und zu überwachen. "Explore" eignet sich gut für ad-hoc-Analysen und das Testen von Abfragen, während "Dashboard" für die langfristige Überwachung und Präsentation von Daten verwendet wird.

14

Docker und Git

14 Docker und Git

14.1 Git

Alle Beispiele und Dokumentationen finden sich folgenden Git-Repository

https://github.com/LimagoHub/atruvia-ops-juni-2023

14.2 Docker

Hier findet sich auch der Link zum Docker-Compose File, sowie allen Configurationsdateien

https://github.com/LimagoHub/atruvia-ops-juni-2023/tree/main/DockerScripts/Grafana_LOKI_Tempo_Prom

14.3 Nützliche Links

https://micrometer.io/docs/concepts

https://docs.spring-boot-admin.com/current/getting-started.html

https://docs.spring.io/spring-boot/docs/2.0.x/actuator-api/html/

https://spring.io/blog/2018/03/16/micrometer-spring-boot-2-s-new-application-metrics-collector

https://tanzu.vmware.com/developer/guides/observability-reactive-spring-boot-3/

https://github.com/Netflix/atlas/wiki/Getting-Started

https://github.com/Netflix/atlas/wiki/Stack-Language

https://www.baeldung.com/micrometer

http://localhost:7101/api/v1/graph?e=2023-06-15T00:00&q=name,personen.all,:eq&s=e-2d

http://localhost:7101/api/v1/graph?e=2023-06-15T00:00&q=name,personen,:eq&s=e-2d

https://netflix.github.io/atlas-docs/api/graph/basics/

https://github.com/DataDog/datadog-agent/tree/main/Dockerfiles/agent

docker run -d --name dd-agent -v /var/run/docker.sock:/var/run/docker.sock:ro -v /proc/:/host/proc/:ro -v /sys/fs/cgroup/:/host/sys/fs/cgroup:ro -e DD_API_KEY=877dcf043e998fdef95acfdd64848159 -e DD_SITE="datadoghq.eu" gcr.io/datadoghq/agent:7

https://docs.datadoghq.com/agent/basic_agent_usage/windows/?tab=gui

java -javaagent:dd-java-agent.jar -Ddd.logs.injection=true -jar C:\Users\JoachimWagner\git\Limago\spring-operartions\wepapp17\target\wepapp-0.0.1-SNAPSHOT.jar

https://spring.io/blog/2022/10/12/observability-with-spring-boot-3

https://neilwhite.ca/spring-boot-3-observeability/

https://www.baeldung.com/spring-boot-self-hosted-monitoring

https://github.com/blueswen/spring-boot-observability

https://refactorfirst.com/distributed-tracing-with-opentelemetry-jaeger-in-spring-boot

https://github.com/blueswen/spring-boot-observability/blob/main/readme.md

https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/

https://refactorfirst.com/distributed-tracing-with-opentelemetry-jaeger-in-spring-boot

https://fossies.org/linux/grafana/devenv/docker/blocks/tempo/docker-compose.yaml

https://grafana.com/blog/2022/04/26/set-up-and-observe-a-spring-boot-application-with-grafana-cloud-prometheus-and-opentelemetry/