



МОНИТОРЬ ЭТО

WARNER BROS. PRESENTS
IN ASSOCIATION WITH VILLAGE ROADSHOW PICTURES AND NPV ENTERTAINMENT A BALTIMORE / SPRING CREEK PICTURES / FACE / TRIBECA PRODUCTION A HAROLD RAMIS FILM
ROBERT DE NIRO BILLY CRYSTAL LISA KUDROW "ANALYZE THIS" JOE VITERELLI AND CHAZZ PALMINTERI MUSIC BY HOWARD SHORE COSTUME DESIGNER LEN AMATO EDITOR CHRISTOPHER TELLEFSEN
PRODUCTION DESIGNER WYNN THOMAS EXECUTIVE PRODUCERS STUART DRYBURGH PRODUCED BY BILLY CRYSTAL CHRIS BRIGHAM AND BRUCE BERMAN DIRECTED BY KENNETH LONERGAN AND PETER TOLAN SCREENPLAY BY PETER TOLAN
VILLAGE ROADSHOW PICTURES R MPAA R RESTRICTED PARENTS STRONGLY CAUTIONED AND HAROLD RAMIS AND KENNETH LONERGAN PRODUCED BY PAULA WEINSTEIN JANE ROSENTHAL WRITTEN BY HAROLD RAMIS
www.analyzethis.com

Обо мне

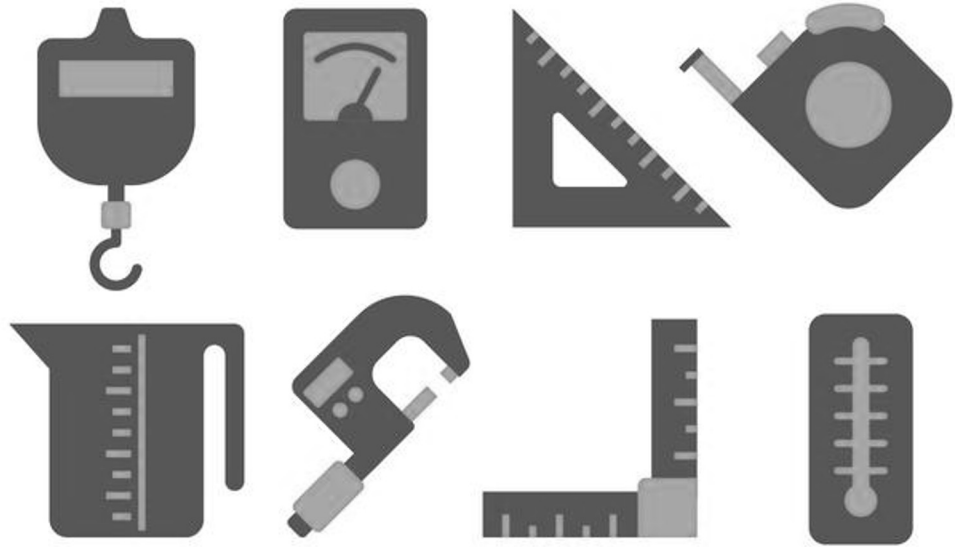


Рустам Курамшин
Java Team Lead
MAGNIT TECH

- Занимаюсь backend-разработкой на Java, Kotlin и Spring
- Люблю Linux, Docker и Kubernetes
- Спикер конференций JPoint, HighLoad++ etc

О чем пойдет речь

- Введение в мониторинг `Spring Boot`
- `Micrometer` и `Spring Boot Actuator`
- Практика разработки кастомных метрик для `Spring Boot`
- Дашборды в `Grafana` и особенности `PromQL`
- Разбор популярных ошибок



I. Введение в МОНИТОРИНГ

Современный мониторинг JVM и Spring Boot микросервисов

Типовой стек мониторинга **Spring Boot**



Micrometer – фасад над клиентами инструментирования. SLF4J для observability



Spring Boot Actuator – модуль **Spring Boot**, предоставляющий готовые к использованию эндпоинты для мониторинга



Prometheus – база данных для сбора, хранения и анализа метрик в реальном времени



Grafana – платформа для визуализации, мониторинга и анализа данных

Метрики

Spring Boot

HTTP, Endpoints,
Latency, RPS



Основные метрики **Spring Boot**

- **Latency**: средние задержки в обработке HTTP-запросов в разрезе REST endpoints
- **RPS**: кол-во HTTP-запросов в единицу времени на конкретные REST endpoints
- **Status Code**: распределение кол-ва статус-кодов HTTP в разрезе REST endpoints
- **Logs**: распределение лог-записей по типам сообщений (error, warn, info), используя LogQL

Представление в Prometheus

Метрика	Что измеряет
<code>http_server_requests_seconds</code>	latency HTTP-запросов
<code>http_server_requests_seconds_count</code>	количество HTTP-запросов
<code>http_server_requests_seconds_sum</code>	суммарное время обработки запросов
<pre>rate(http_server_requests_seconds_count {uri="/api/orders"}[1m])</pre>	в разрезе endpoint

Интересный факт про LogQL #1

- Логи можно агрегировать как метрики, считать количество ошибок, строить распределения и динамику без изменения кода приложения
- Можно извлекать структуру из “сырого” текста `json`, `grep` и форматирование позволяют выделить `logger`, тип ошибки и другие поля прямо в запросе
- Получаем топ проблем в системе, группировка по `logger` и `short_message` показывает какие ошибки происходят чаще всего и где искать проблему

Интересный факт про LogQL #2

```
1 sum by(logger_name, short_message) (  
2   count_over_time(  
3     {cluster="prod", namespace="mm-core-bff", container="mm-core-bff", level="error"}  
4     | json  
5     | line_format "{{.logger_name}} | {{.message}}"  
6     | error!="JSONParserErr"  
7     | regexp "^(?P<loggerName>[^|]+)\\|(?P<short_message>[^:|]{1,45})"  
8     [10m]  
9   )  
10 )
```

Интересный факт про LogQL #2

```
1 sum by(logger_name, short_message) (  
2   count_over_time(  
3     {cluster="prod", namespace="mm-core-bff", container="mm-core-bff", level="error"}  
4     | json  
5     | line_format "{{.logger_name}} | {{.message}}"  
6     | error!="JSONParserErr"  
7     | regexp "^(?P<loggerName>[^|]+)\\|(?P<short_message>[^:|]{1,45})"  
8     [10m]  
9   )  
10 )
```

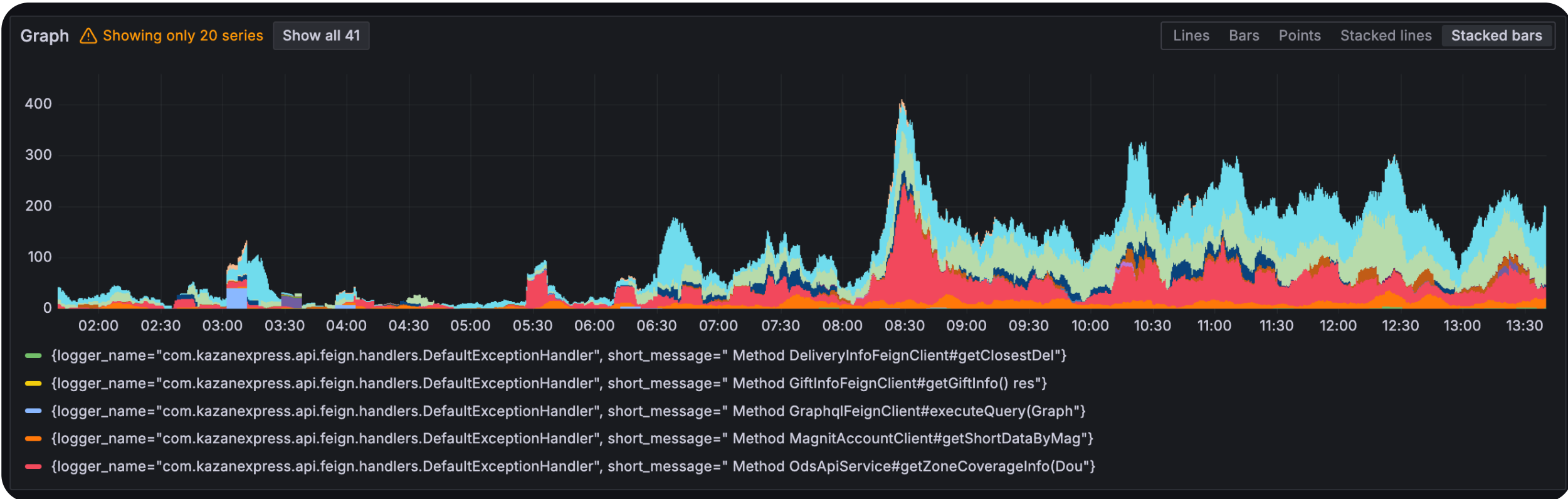
Интересный факт про LogQL #2

```
1 sum by(logger_name, short_message) (  
2   count_over_time(  
3     {cluster="prod", namespace="mm-core-bff", container="mm-core-bff", level="error"}  
4     | json  
5     | line_format "{{.logger_name}} | {{.message}}"  
6     | error!="JSONParserErr"  
7     | regexp "^(?P<loggerName>[^|]+)\|(?P<short_message>[^:|]{1,45})"  
8     [10m]  
9   )  
10 )
```

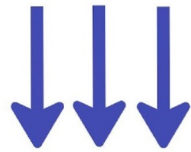
Интересный факт про LogQL #2

```
1 sum by(logger_name, short_message) (  
2   count_over_time(  
3     {cluster="prod", namespace="mm-core-bff", container="mm-core-bff", level="error"}  
4     | json  
5     | line_format "{{.logger_name}} | {{.message}}"  
6     | error!="JSONParserErr"  
7     | regexp "^(?P<loggerName>[^|]+)\\|(?P<short_message>[^:|]{1,45})"  
8     [10m]  
9   )  
10 )
```

Интересный факт про LogQL #3

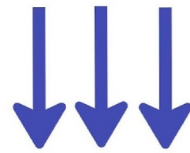


Актuator умеет больше, чем кажется #1

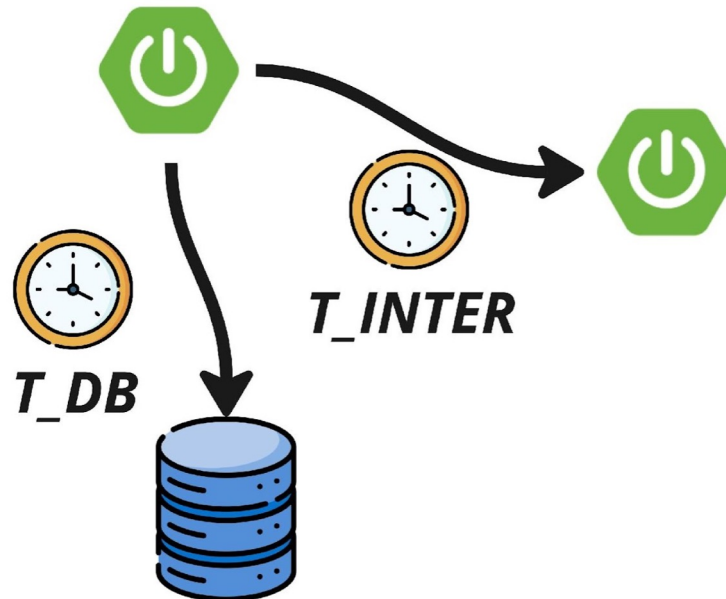


$$T \sim T_{DB} + T_{INTER}$$

Актuator умеет больше, чем кажется #1



$$T \sim T_{DB} + T_{INTER}$$



Метрики запросов в базу данных

Метрика	Что измеряет
<code>spring_data_repository_invocations</code>	количество вызовов методов Spring Data репозиториев
<code>spring_data_repository_invocations_seconds</code>	время выполнения методов репозиториев (latency)
<code>spring_data_repository_invocations_seconds_count</code>	количество вызовов методов
<code>spring_data_repository_invocations_seconds_sum</code>	суммарное время выполнения

Метрики http клиентов

Метрика	Что измеряет
<code>http_client_requests_seconds</code>	latency HTTP-клиентов
<code>http_client_requests_seconds_count</code>	количество исходящих HTTP-запросов
<code>http_client_requests_seconds_sum</code>	суммарное время внешних вызовов



+



II. Micrometer и Spring Boot Actuator

Завод по производству метрик под капотом Spring Boot

Что такое Micrometer

- Micrometer – vendor-neutral фасад для популярных систем мониторинга. Это SLF4J для мониторинга
- Интегрируется с популярными фреймворками: *Spring, Quarkus, Micronaut*
- Поддерживает публикацию метрик в *Prometheus, Datadog, Dynatrace, Graphite, Influx, New Relic* и многие другие

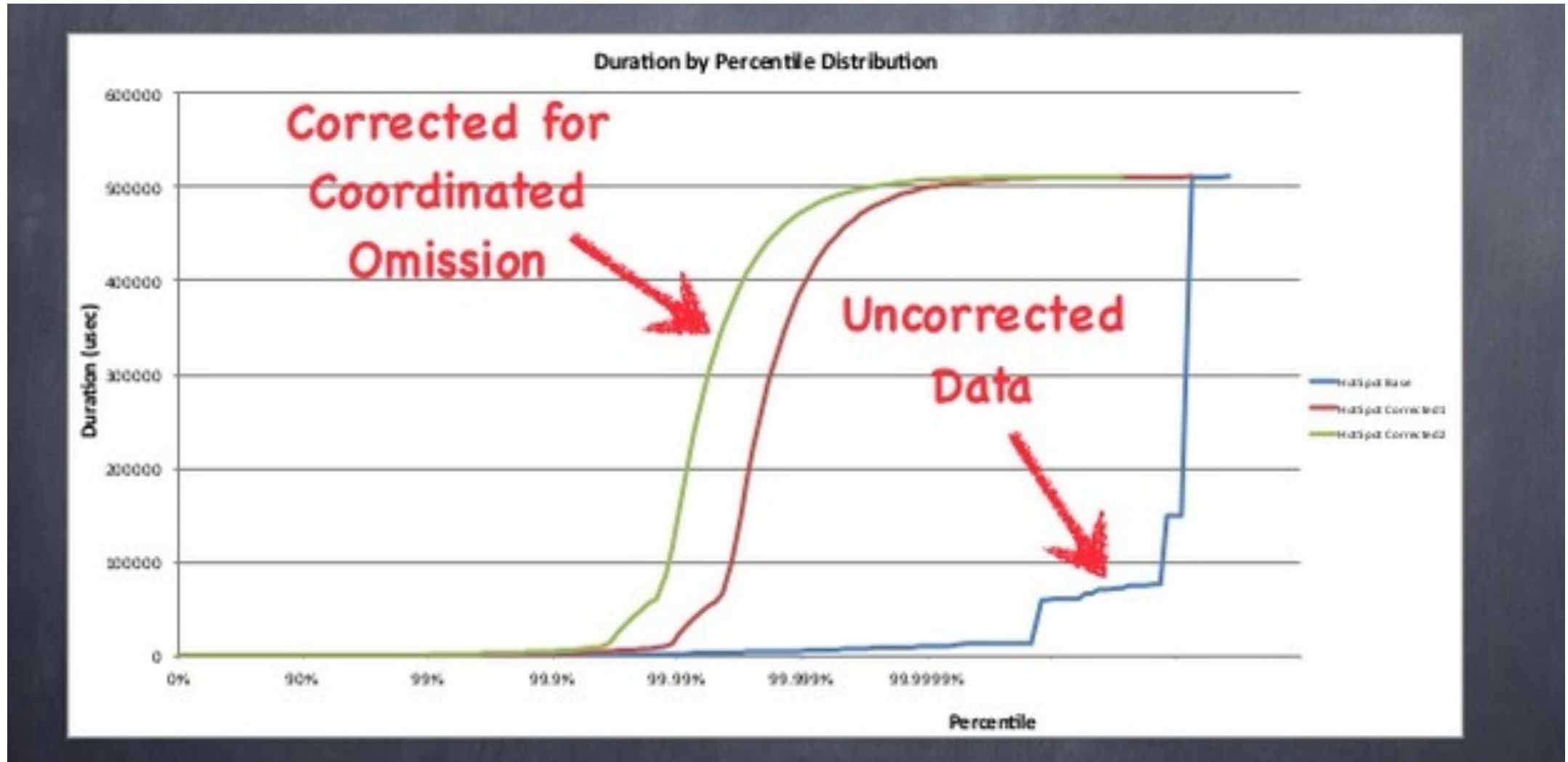
Micrometer под капотом

- Micrometer интегрируется с JVM через Java Management Extensions (JMX), Java Flight Recorder (JFR), Garbage Collector Logs и Runtime API
- Содержит инструментальные SPI (Service Provider Interface) для поддержки различных систем мониторинга
- Имеет "детектор пауз", который использует LatencyUtils для компенсации "*coordinated omission*" – дополнительной задержки из-за пауз GC и процессов внутри VM, которые занижают метрики

Что такое *coordinated omission*

- Мы думаем, что измеряем реальное время ответа системы
- На самом деле измеряем время ответа только тогда, когда система уже “в состоянии отвечать”
- “пропущенные” запросы - это и есть *coordinated omission*

Gil Tene - "How NOT To Measure Latency"



Флоу формирования метрик

1. Application code

```
Counter counter = Counter.builder("http.client.requests")  
    .tag("client", "billing")  
    .description("Number of HTTP client requests")  
    .register(meterRegistry);
```



2. Micrometer API

`Counter.builder()`

`Timer.builder()`


`Gauge.builder()`

...

3. MeterRegistry (метрика создаётся и управляется через registry)

1 Собирает Meter.Id

- name = "http.client.requests"
- tags = ["client=billing"]
- baseUnit = ...
- description = ...



2 Прогоняет Meter.Id через MeterFilter



MeterFilter


- может изменить имя метрики
- может добавить/изменить теги
- может задать baseUnit/description
- может управлять гистограммами/перцентиями
- может принять или отклонить метрику



ACCEPT
метрика разрешена

DENY
метрика отклонена
(возвращается NoopMeter)


3 Проверяет, есть ли уже meter с таким id




Есть

Нет


4 Создаёт backend-specific реализацию meter



5 Кладёт meter во внутреннее хранилище registry



6 Возвращает существующий или новый meter в приложение



Флоу формирования метрик

4. Meters (универсальные типы)

Counter

Timer

Gauge

DistributionSummary

LongTaskTimer

...

5. Backend-specific registry

(адаптер под конкретную систему мониторинга)

PrometheusMeterRegistry

DatadogMeterRegistry

OtlpMeterRegistry

GraphiteMeterRegistry

AtlasMeterRegistry

...

6. Monitoring systems



Prometheus



Datadog



OTLP
(OpenTelemetry)



Graphite



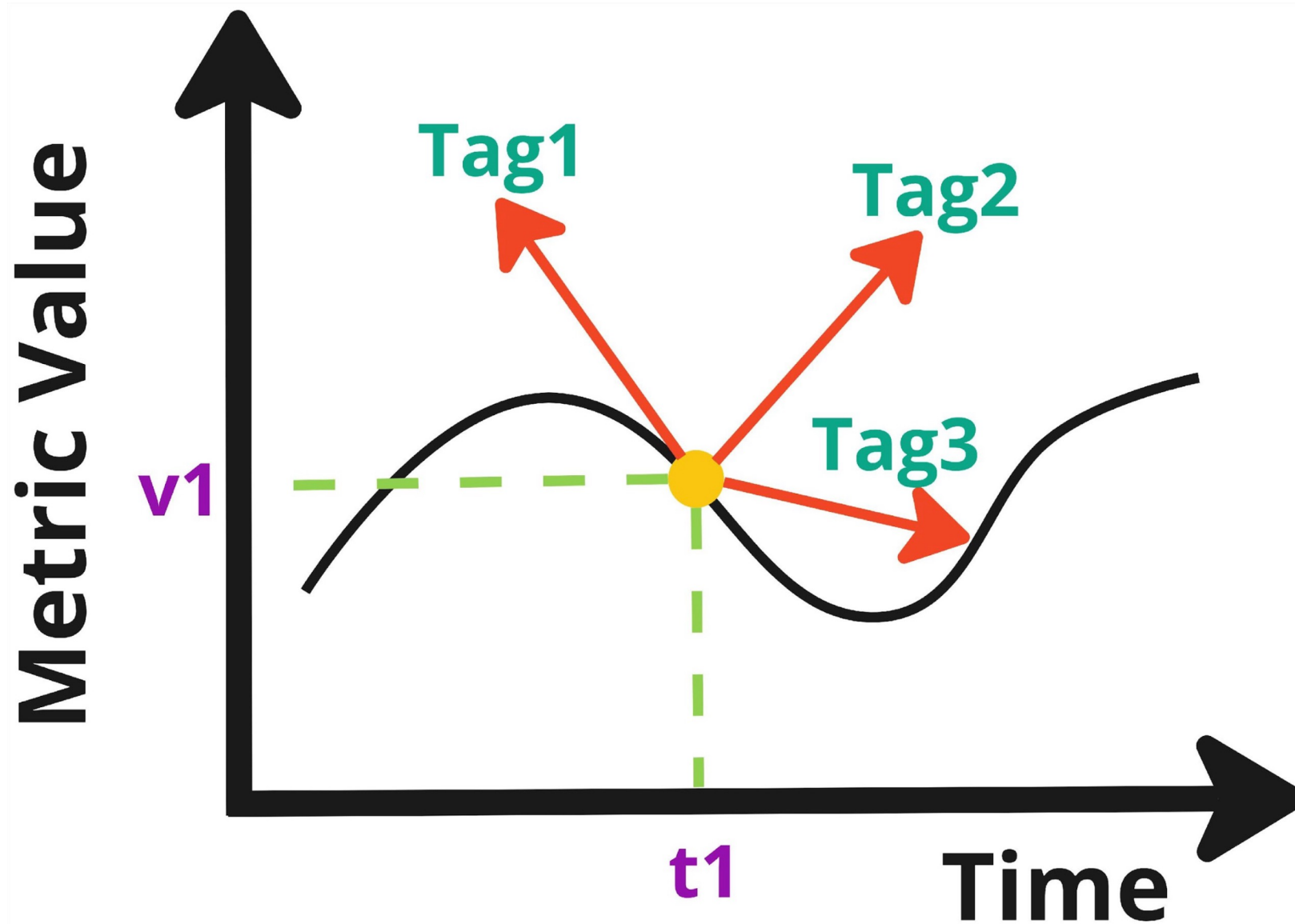
Atlas

...

Micrometer API

- ***Meter*** - интерфейс для сбора метрик. Реализации: *Timer*, *Counter*, *Gauge*, *DistributionSummary*, *LongTaskTimer*, *FunctionCounter*, *FunctionTimer*, и *TimeGauge*
- ***Tags*** - метки, которые можно присваивать метрикам, чтобы различать их по какому-либо критерию, например, по региону или версии сервера
- ***Meters*** в Micrometer создаются и хранятся в ***MeterRegistry***. Каждая поддерживаемая система мониторинга имеет реализацию ***MeterRegistry***. Способ создания реестра различается для каждой реализации

Метрика и теги



Виды счетчиков: Counter

- **Counter** – это монотонно увеличивающаяся метрика, которая используется для отслеживания количественных значений, таких как количество запросов к серверу, ошибок или любых других событий, которые можно подсчитать
- **Простота:** Предназначен только для инкрементирования, не поддерживает уменьшение
- **Использование:** Идеален для отслеживания событий, количество которых со временем растёт

Counter

```
1 @RestController
2 public class HelloController {
3
4     private final Counter counter;
5
6     public HelloController(MeterRegistry registry) {
7         this.counter = Counter.builder("hello.requests.count")
8             .description("Total number of Hello requests")
9             .tags("hello", "counterexample")
10            .register(registry);
11    }
12
13    @GetMapping("/hello")
14    public String handleRequest() {
15        counter.increment();
16        return "hello";
17    }
18 }
```

Виды счетчиков: Gauge

- **Gauge** – это метрика, которая представляет собой мгновенный снимок некоторого числового значения. Это может быть количество элементов в коллекции, значение температуры, объем памяти и т.п.
- **Гибкость значения:** Значение может как увеличиваться, так и уменьшаться
- **Прямая зависимость:** Часто **Gauge** связан с объектами или переменными, отслеживая их текущее состояние

Gauge

```
1 @RestController
2 public class ListController {
3
4     private List<String> list = Collections.synchronizedList(new LinkedList<>());
5
6     public ListController(MeterRegistry registry) {
7         Gauge.builder("list.size", list, List::size)
8             .description("The size of the list")
9             .register(registry);
10    }
11
12    @GetMapping("/add")
13    public String addItem() {
14        list.add("item");
15        return "Added item, list size now " + list.size();
16    }
17
18    @GetMapping("/remove")
19    public String removeItem() {
20        if (!list.isEmpty()) {
21            list.remove(0);
22        }
23        return "Removed item, list size now " + list.size();
24    }
25 }
```

Виды счетчиков: Timer

- **Timer** – это счетчик для измерения длительности операций и частоты их выполнения
- Измерение времени выполнения: **Timer** измеряет длительность операций в наносекундах и автоматически агрегирует данные о времени выполнения
- Количество и частота: Помимо длительности, **Timer** также подсчитывает количество произведенных операций, что позволяет измерять частоту событий

Timer

```
1 @RestController
2 public class HelloController {
3
4     private final Timer responseTimer;
5
6     public HelloController(MeterRegistry registry) {
7         this.responseTimer = Timer.builder("hello.response.time")
8             .description("Time taken to return a response")
9             .register(registry);
10    }
11
12    @GetMapping("/hello")
13    public String handleRequest() {
14        return responseTimer.record(() -> {
15            try {
16                TimeUnit.MILLISECONDS.sleep(200); // задержка 200 мс
17            } catch (InterruptedException e) {
18                Thread.currentThread().interrupt();
19            }
20            return "hello";
21        });
22    }
23 }
```

Виды счетчиков:

`DistributionSummary`

- `DistributionSummary` – это инструмент для сбора статистики по распределению измеряемых значений
- `DistributionSummary` – помогает эффективно разгрузить ресурсы Prometheus-сервера потому что использует агрегацию данных
- Сбор статистики: Автоматически рассчитывает среднее, сумму, максимум и другие статистические параметры. Может быть настроен для автоматического подсчета квантилей и построения гистограмм распределения значений

Distribution Summaries

```
1 private final DistributionSummary payloadSize;
2
3 public OrderService(MeterRegistry registry) {
4     this.payloadSize = DistributionSummary.builder("http.response.size")
5         .description("Response payload size")
6         .baseUnit("bytes")
7         .publishPercentiles(0.5, 0.95, 0.99)
8         .register(registry);
9 }
10
11 public List<OrderDto> getOrders() {
12     List<OrderDto> result = loadOrders();
13
14     payloadSize.record(estimateSize(result));
15
16     return result;
17 }
```

Distribution Summaries

- Какие метрики появляются в Prometheus:

http_response_size_count

http_response_size_sum

http_response_size_max

http_response_size{quantile="0.95"}

Теги не бесплатны для Prometheus

- Каждый уникальный набор тегов создает отдельную метрику, что приводит к увеличению количества временных рядов
- Таким образом, увеличение количества значений тегов приводит к экспоненциальному росту данных, которые должны быть сохранены в Prometheus

Оценка кол-ва временных рядов в Prometheus

`http_requests_total{method="GET", status="200"}`

Кол-во тегов	Значения тегов	Кол-во временных рядов
0		1
1 (method)	GET, POST	2
2 (method, status)	GET, POST, 200, 404	4
...

Как убить Prometheus за 5 минут

- Добавлять динамические значения из запроса *userId*, *requestId*, *sessionId* выглядят полезно, но это создает новые временные ряды на каждый запрос
- Использовать “сырые” значения без агрегации эндпойнтов с параметрами, *email*, *UUID*, что приводит к неконтролируемому росту *cardinality*
- Не думать о количестве комбинаций - каждый новый *tag* умножает количество временных рядов

Динамический URL в тэгах

```
1 @Component
2 @RequiredArgsConstructor
3 public class HttpMetricsFilter extends OncePerRequestFilter {
4
5     private final MeterRegistry meterRegistry;
6
7     @Override
8     protected void doFilterInternal(HttpServletRequest request,
9                                     HttpServletResponse response,
10                                    FilterChain filterChain)
11         throws ServletException, IOException {
12
13         String uri = request.getRequestURI();
14
15         Counter.builder("http_requests_total")
16             .tag("uri", uri)
17             .tag("method", request.getMethod())
18             .register(meterRegistry)
19             .increment();
20
21         filterChain.doFilter(request, response);
22     }
23 }
```



Динамический URL в тэгах

```
1 @Component
2 @RequiredArgsConstructor
3 public class HttpMetricsFilter extends OncePerRequestFilter {
4
5     private final MeterRegistry meterRegistry;
6
7     @Override
8     protected void doFilterInternal(HttpServletRequest request,
9                                     HttpServletResponse response,
10                                    FilterChain filterChain)
11         throws ServletException, IOException {
12
13         String uri = request.getRequestURI();
14
15         Counter.builder("http_requests_total")
16             .tag("uri", uri)
17             .tag("method", request.getMethod())
18             .register(meterRegistry)
19             .increment();
20
21         filterChain.doFilter(request, response);
22     }
23 }
```



Как сломать cardinality в бизнес-метриках

```
1 @Service
2 @RequiredArgsConstructor
3 public class PaymentMetricsService {
4
5     private final MeterRegistry meterRegistry;
6
7     public void recordPayment(String orderId, String email, String externalStatus)
8     {
9         Timer.builder("payment_processing_seconds")
10            .tag("orderId", orderId)
11            .tag("email", email)
12            .tag("externalStatus", externalStatus)
13            .register(meterRegistry)
14            .record(() -> processPayment(orderId));
15
16     private void processPayment(String orderId) {
17         // business logic
18     }
19 }
```



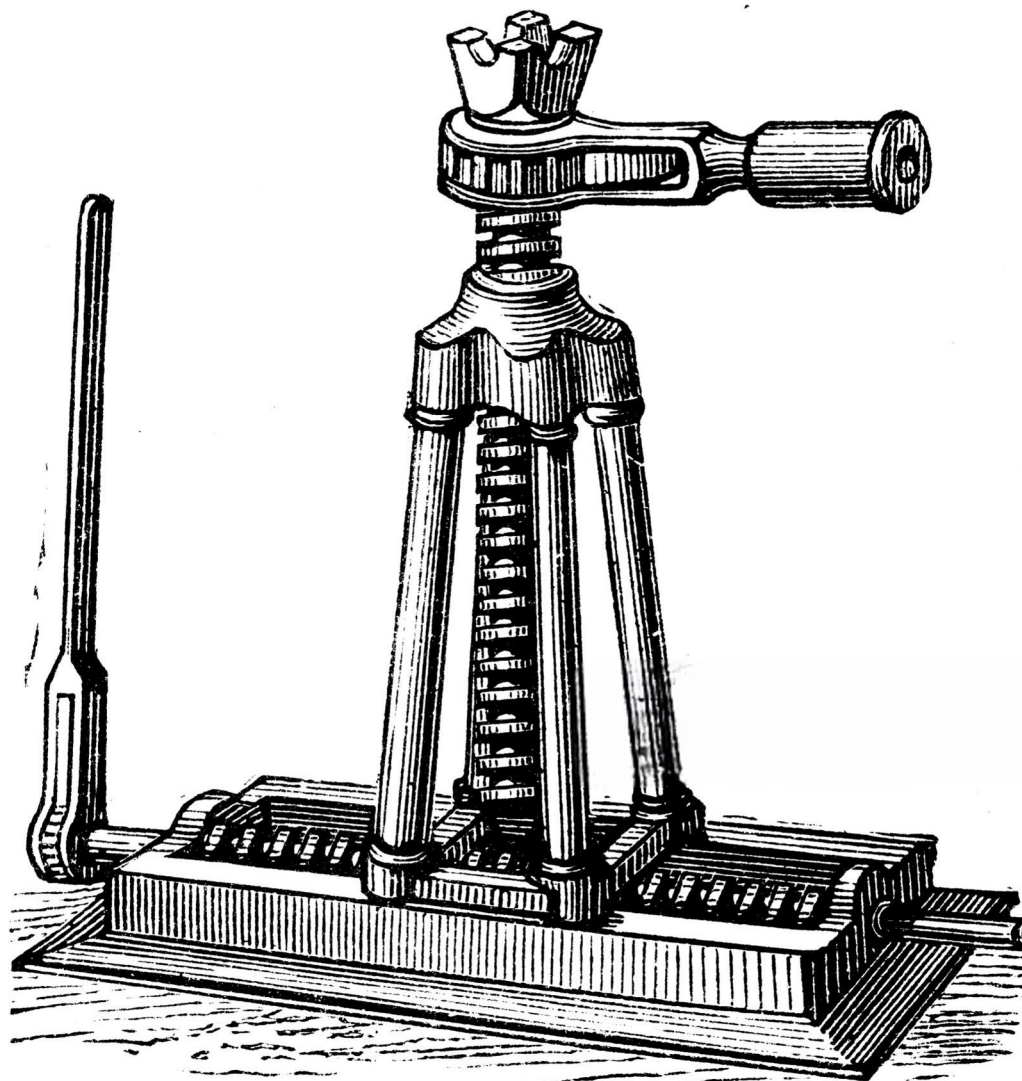
Как сломать cardinality в бизнес-метриках

```
1 @Service
2 @RequiredArgsConstructor
3 public class PaymentMetricsService {
4
5     private final MeterRegistry meterRegistry;
6
7     public void recordPayment(String orderId, String email, String externalStatus)
8     {
9         Timer.builder("payment_processing_seconds")
10            .tag("orderId", orderId)
11            .tag("email", email)
12            .tag("externalStatus", externalStatus)
13            .register(meterRegistry)
14            .record(() -> processPayment(orderId));
15
16     private void processPayment(String orderId) {
17         // business logic
18     }
19 }
```



Spring Boot Actuator

Актуатор – это механическое устройство. Актуаторы могут превращать небольшое движение в значительное



Что такое **Spring Boot** Actuator

- **Spring Boot** Actuator является одним из ключевых инструментов мониторинга Spring Boot. Он предоставляет Production-ready Features, которые помогают вам мониторить ваше приложение через HTTP или JMX
- **Spring Boot** включает в себя ряд встроенных actuator endpoints и позволяет вам добавлять свои собственные. Например, **GET /actuator/prometheus** публикует метрики для Prometheus

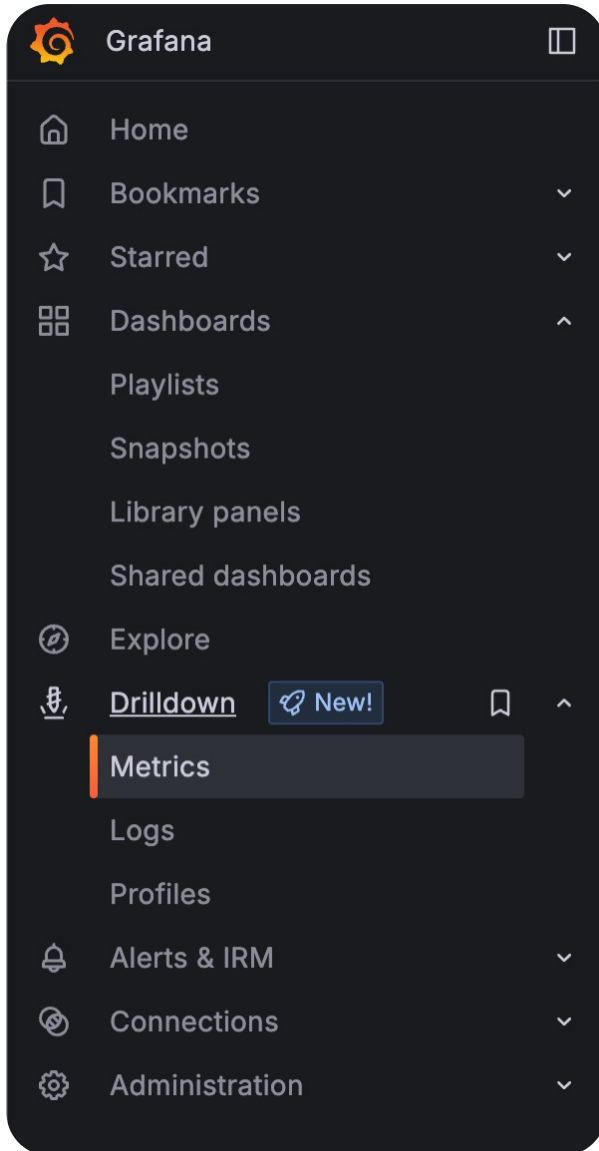
Метрики доступные через Actuator



- JVM Metrics
- System Metrics
- Application Startup Metrics
- Logger Metrics
- Task Execution and Scheduling Metrics
- JMS Metrics
- Spring MVC Metrics
- Spring WebFlux Metrics
- Jersey Server Metrics
- HTTP Client Metrics
- Tomcat Metrics
- Cache Metrics
- Spring Batch Metrics
- Spring GraphQL Metrics
- DataSource Metrics
- Hibernate Metrics
- Spring Data Repository Metrics
- RabbitMQ Metrics
- Spring Integration Metrics
- Kafka Metrics
- MongoDB Metrics
- Jetty Metrics
- Redis Metrics

<https://docs.spring.io/spring-boot/reference/actuator/metrics.html#actuator.metrics.supported>

Grafana Drilldown (ex. Metric Explorer)



III. Практика разработки кастомных метрик для Spring Boot

Когда нужно что-то измерить, но не знаешь как



**Считать
метрику
in-place**



**Написать
аннотацию и
считать
метрику в аспекте**

Основные подходы к разработке метрик

- Измерение `in-place`: "на месте", там где вам нужно в коде использовать `Counter`, `Gauge`, `Timer` и т.п. (см. предыдущий раздел)
- Измерение как аспект (AOP): разработка аннотации и аспекта для проведение измерений в различных частях `Spring Boot` приложения

Почему не профайлинг или анализ дампов



Пример из практики:

Расследование *OutOfMemoryError*

- Некое API имеет множество endpoints, которые возвращают списки сущностей
- Возникла потребность сделать графики в Grafana, которые бы могли ответить на вопросы на каких endpoints возвращаются самые большие списки и кто за ними приходит

Аннотация для разметки измеряемых методов



```
1 @Target( ElementType.METHOD )  
2 @Retention( RetentionPolicy.RUNTIME )  
3 public @interface MeasureSizeOfList {  
4  
5 }
```

Аспект, перехватывающий MeasureSizeOfList

```
1 @Aspect
2 @Component
3 @RequiredArgsConstructor
4 public class AspectMeasuringSizeOfList {
5
6     private final MeterRegistry meterRegistry;
7
8     @Around("@annotation(MeasureSizeOfList)")
9     public Object measureSizeOfList(ProceedingJoinPoint joinPoint) throws Throwable {
10         Object result = joinPoint.proceed();
11
12         if (result instanceof BaseListDto<?> list) {
13             HttpServletRequest request = ((ServletRequestAttributes)
14                 RequestContextHolder.currentRequestAttributes()).getRequest();
15
16             String sourceIp = request.getRemoteAddr();
17
18             meterRegistry.counter("list_endpoints_size", "source_ip", sourceIp).increment(list.getItems().size())
19         }
20
21         return result;
22 }
```

Пример из практики:

Анализ `@Transactional` методов и CRUD-операций с БД

- Найти самые длительные `@Transactional`-методы
- Статистика по `@Transactional`-методам и CRUD-операциям с БД

Аспект, измеряющий длительность @Transactional- методов

```
1 @Aspect
2 @Component
3 public class TransactionalTimerAspect {
4
5     private final MeterRegistry meterRegistry;
6
7     @Autowired
8     public TransactionalTimerAspect(MeterRegistry meterRegistry) {
9         this.meterRegistry = meterRegistry;
10    }
11
12    @Pointcut("@annotation(org.springframework.transaction.annotation.Transactional)")
13    public void transactionalMethods() {
14    }
15
16    @Around("transactionalMethods()")
17    public Object measureMethodExecutionTime(ProceedingJoinPoint pjp) throws Throwable {
18        MethodSignature signature = (MethodSignature) pjp.getSignature();
19        String methodName = signature.getMethod().getName();
20
21        Timer.Sample sample = Timer.start(meterRegistry);
22        try {
23            return pjp.proceed();
24        } finally {
25            sample.stop(Timer.builder("transactions.timer")
26                .description("Duration of transactional methods")
27                .tags("method", methodName)
28                .register(meterRegistry));
29        }
30    }
31 }
```

Аспект измеряющий длительность CRUD-операций с БД на основе статистики Hibernate (Actuator не выставляет такие метрики)

```
1 @Aspect
2 @Component
3 public class HibernateStatisticsAspect {
4
5     private final Statistics statistics;
6     private final MeterRegistry meterRegistry;
7
8     @Autowired
9     public HibernateStatisticsAspect(SessionFactory sessionFactory, MeterRegistry meterRegistry) {
10         this.statistics = sessionFactory.getStatistics();
11         this.meterRegistry = meterRegistry;
12     }
13
14     @Pointcut("@annotation(transactional)")
15     public void transactionalMethods(Transactional transactional) {
16     }
17
```

Аспект измеряющий длительность CRUD-операций с БД на основе статистики Hibernate (Actuator не выставляет такие метрики)

```
18 @Around("transactionalMethods(transactional)")
19 public Object recordStatistics(ProceedingJoinPoint joinPoint, Transactional transactional) throws Throwable {
20     MethodSignature signature = (MethodSignature) joinPoint.getSignature();
21     String methodName = signature.getMethod().getDeclaringClass().getSimpleName() + "." + signature.getMethod().getName();
22     List<Tag> tags = Arrays.asList(Tag.of("method", methodName));
23
24     long startEntityLoadCount = statistics.getEntityLoadCount();
25     long startEntityInsertCount = statistics.getEntityInsertCount();
26     long startEntityUpdateCount = statistics.getEntityUpdateCount();
27     long startEntityDeleteCount = statistics.getEntityDeleteCount();
28
29     try {
30         return joinPoint.proceed();
31     } finally {
32         long endEntityLoadCount = statistics.getEntityLoadCount();
33         long endEntityInsertCount = statistics.getEntityInsertCount();
34         long endEntityUpdateCount = statistics.getEntityUpdateCount();
35         long endEntityDeleteCount = statistics.getEntityDeleteCount();
36
37         meterRegistry.counter("db.entity.load", tags).increment(endEntityLoadCount - startEntityLoadCount);
38         meterRegistry.counter("db.entity.insert", tags).increment(endEntityInsertCount - startEntityInsertCount);
39         meterRegistry.counter("db.entity.update", tags).increment(endEntityUpdateCount - startEntityUpdateCount);
40         meterRegistry.counter("db.entity.delete", tags).increment(endEntityDeleteCount - startEntityDeleteCount);
41
42         // statistics.clear();
43     }
44 }
45 }
```

Пример из практики: Измерение длительности `suspend`-функций `Kotlin`

Support for observing suspending functions with `@Observed` annotation #4827

ilya40umov opened on Mar 4, 2024

Problem Statement

`@observed` annotation, which is powered by `ObservedAspect`, does not seem to be able to correctly handle suspending methods. The aspect is able to invoke the suspending call, but the newly created observation does not become "current" within the suspending method.

Rationale

The issue #4754 is covering additions to the Observation API, so that it can be used for suspending functions. However, a lot of Spring users may prefer to use the annotation `@observed` in their existing code.

Additional context

Seems like Spring Framework does support AOP on suspending code since `SpringFramework2462`. E.g. `AppUtils` has additional logic for dealing with invoking suspending functions.

Also, looking at [this comment](#) has made me think that `ObservedAspect` could be modified to use `ProceedingJoinPoint.proceed()` and then put the newly created observation on `ObservationThreadLocalAccessor.KEY`.

Raised initially in [issues/4754#issuecomment-1977193040](#) by [@ArvindErryov](#)

Timer API does not support Kotlin suspend function (coroutines) #4455

AleksanderBrzozowski opened on Dec 7, 2023

Please describe the feature request.

For standard (non suspend functions) it is possible to use `record` function to measure method invocation time:

```
Timer.builder("method.time")  
    .register(meterRegistry)  
    .record { fn() } // fn() is a standard, non suspend function
```

When using `record` function to measure time needed to call suspend function, the observation is stopped before the suspend function is called. Suspend functions can be called only within coroutine body.

It would be nice if the `record` function could be used to measure time needed to call suspend function. This would be useful for measuring time needed to call suspend function, and it is currently not possible to measure time needed to call suspend function.

API, and it is currently not possible to measure time needed to call suspend function.

How do I prevent Micrometer observations from being stopped when a suspending function is called?

Asked 1 year, 4 months ago Modified 1 year, 4 months ago Viewed 759 times

1

I have a Spring application written in Kotlin that uses the Micrometer observation library (1.10.3). The application executes a `suspend` function that is annotated with Micrometer's `@Observed` annotation. The function calls a Kotlin coroutine `await` function within it.

I have noticed while inspecting the logs of observation events that the observation started by the `@Observed` annotation is stopped (and any open scopes for the observation are closed) right before it calls the Kotlin coroutine `await` function. This causes any observation events created after the `await` function to be created in the wrong observation (or in no observation at all). How do I prevent this from happening?

As a side note, I've also noticed that within the `ObservedAspect` class's code, there is handling for methods that have a `CompletionStage` return type. However, modifying the return type of my `@Observed` function changes nothing, and the return type is always evaluated as just `java.util.Object`. I've noticed that this only happens if the function being `@Observed` has the `suspend` keyword, and the return type evaluates correctly if there is no `suspend`.

`kotlin` `kotlin-coroutines` `micrometer` `spring-micrometer`

Share Improve this question Follow

asked Sep 7, 2023 at 4:38
Charles Lee
21 # 3

Пример из практики:

Измерение длительности *suspend*-функций *Kotlin*

- Работа с *suspend*-функциями *Kotlin* известный *challenge* для *Micrometer*. *@Timed* не работает адекватно
- Чтобы произвести измерения длительности выполнения кода нужно получить время до начала выполнения и после его завершения *suspend*-функции
- Когда мы говорим про *suspend*-функции стоит вспомнить что дождаться их завершения из *java*-кода крайне сложно. Интероп *suspend*'ов оставляет желать лучшего

Метод обертка для измерений

```
11 suspend fun <T: Any> coroutineMetricsWithNullable(  
12     suspendFunc: suspend () -> T?,  
13     metricName: String,  
14     moreTags: Map<String, String> = emptyMap(),  
15     timeBuckets: Array<Duration> = DEFAULT_TIME_BUCKETS,  
16     meterRegistry: MeterRegistry  
17 ): T? {  
18     require(timeBuckets.isNotEmpty()) { "timeBuckets are mandatory to create latency distribution histogram" }  
19     val timer = statisticTimerBuilder(  
20         metricsLabelTag = metricName,  
21         moreTags = moreTags,  
22         timeBuckets = timeBuckets  
23     )  
24     .register(meterRegistry)  
25     val clock = meterRegistry.config().clock()  
26     val start = clock.monotonicTime()  
27     try {  
28         return suspendFunc.invoke()  
29     } finally {  
30         val end = clock.monotonicTime()  
31         timer.record(end - start, TimeUnit.NANOSECONDS)  
32     }  
33 }  
34 }
```

Теперь можно замерять suspend'ы

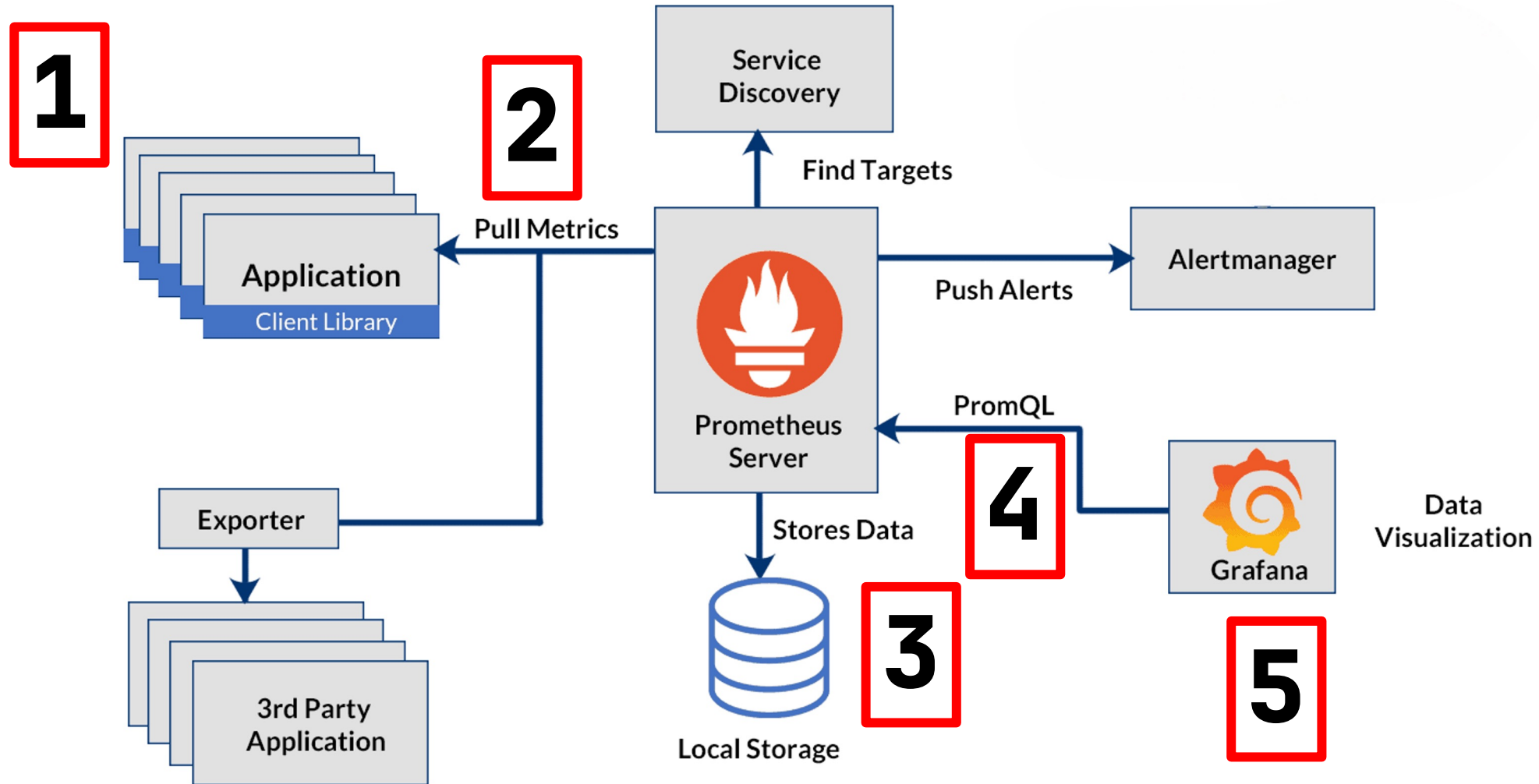
```
1     coroutineMetrics(  
2         suspendFunc = suspend {  
3             anyApiCall.executeAsync(...).awaitSingle()  
4         },  
5         metricName = "api.call",  
6         moreTags = mapOf("method" to "anyApiCall", "entity" to "cat"),  
7         meterRegistry = meterRegistry  
8     )
```



IV. Дашборды в Grafana и особенности PromQL

Когда графики дают ответы

Гrafana работает в связке с Prometheus и PromQL



Важные PromQL запросы для метрик Micrometer и Spring Boot

- Просто выборка для счетчика Counter:

rate(counter[10s])


- Latency: Средняя задержка для счетчика Timer:

rate(timer_sum[10s])/rate(timer_count[10s])

- RPS: Пропускная способность для счетчика Timer:

rate(timer_count[10s])

PromQL best practice #1

- Не путать сырой счетчик и скорость изменения
- Частая ошибка: вывести **_count* или **_total* напрямую
- Такой график будет только расти и плохо отвечает на вопрос “что происходит сейчас”
-  Пример PromQL: *http_server_requests_seconds_count*

PromQL best practice #2

- Понимать разницу между `rate()` и `increase()`
- `rate()` хорош для графиков RPS, throughput, error rate
- `increase()` хорош для ответа “сколько событий произошло за период”
- Для дашбордов чаще нужен `rate()`, алертов по количеству событий иногда удобнее `increase()`

PromQL best practice #3

- Percentile лучше считать через histogram, а не через среднее

```
1 histogram_quantile(  
2  
3     0.95,  
4  
5     sum by (le, uri) (  
6  
7         rate(http_server_requests_seconds_bucket[5m])  
8  
9     )  
10  
11 )
```

PromQL best practice #4

- Не использовать высококардинальные labels в группировках
- Даже если метрика уже содержит опасные labels, не стоит делать:
sum by (userId, requestId) (...)
- Это убьет читаемость Grafana и нагрузит Prometheus

VI. ВЫВОДЫ

Что мы узнали

- `Micrometer` и `Actuator` решают основные задачи с доставкой метрик в `Prometheus`
- Можно (нужно) писать свои кастомные метрики для дебага на проде. Для дашбордов используется несложные `PromQL` запросы с соблюдением `best practice`
- Для этого удобно использовать подход основанный на аннотациях и AOP-аспектах

Подпишитесь на Spring АйО



Популярный в Индонезии канал



Задать мне вопросы или посмотреть другие доклады



СПАСИБО ЗА ВНИМАНИЕ! 🤝 🙏

Можем обсудить вопросы:

- Как написать свою метрику для X?
- Когда профайлинг или свои метрики?
- Как не ошибиться в PromQL?
- Разобрать кейс из вашей практики