# мониторь это

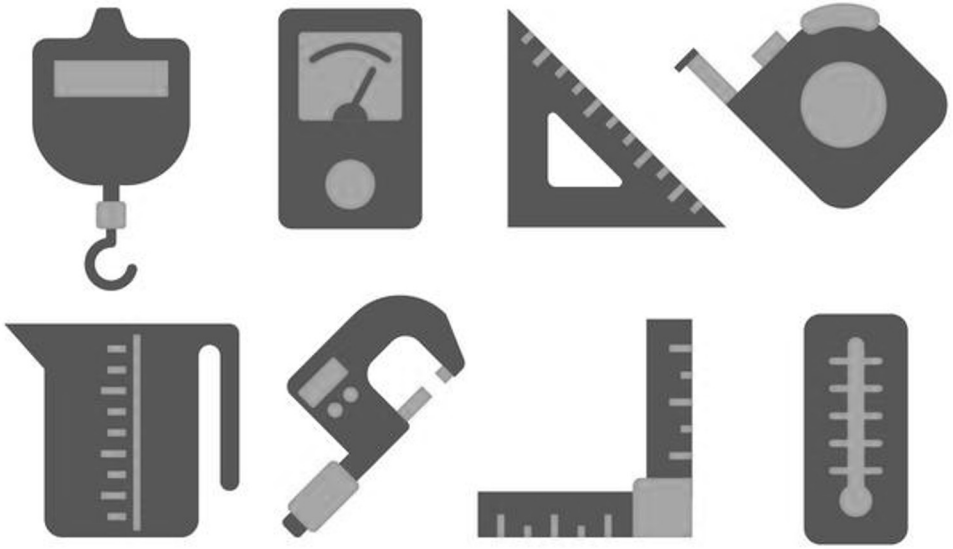
# Обо мне

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 |
| --- | --- |
| http_server_requests_seconds | latency HTTP-запросов |
| http_server_requests_seconds_count | количество HTTP-запросов |
| http_server_requests_seconds_sum | суммарное время обработки запросов |
| rate(<br>http_server_requests_seconds_count<br>{uri="/api/orders"}[1m]<br>) | 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, regexp 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

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

# Интересный факт про LogQL #2

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

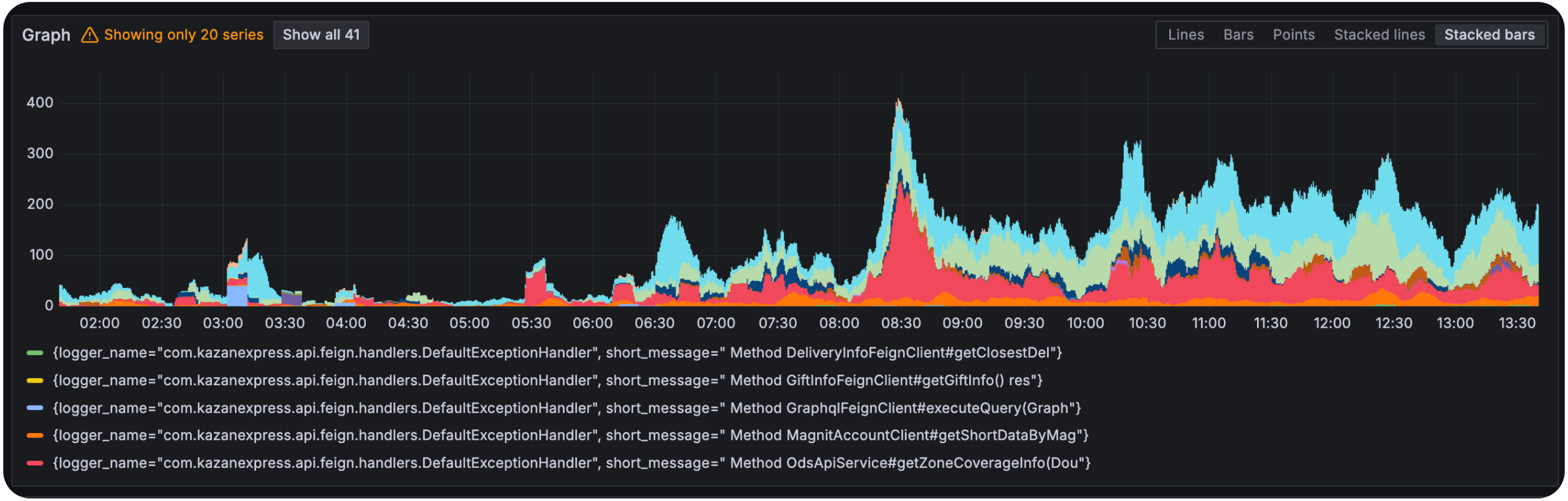
# Интересный факт про LogQL #2

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

# Интересный факт про LogQL #2

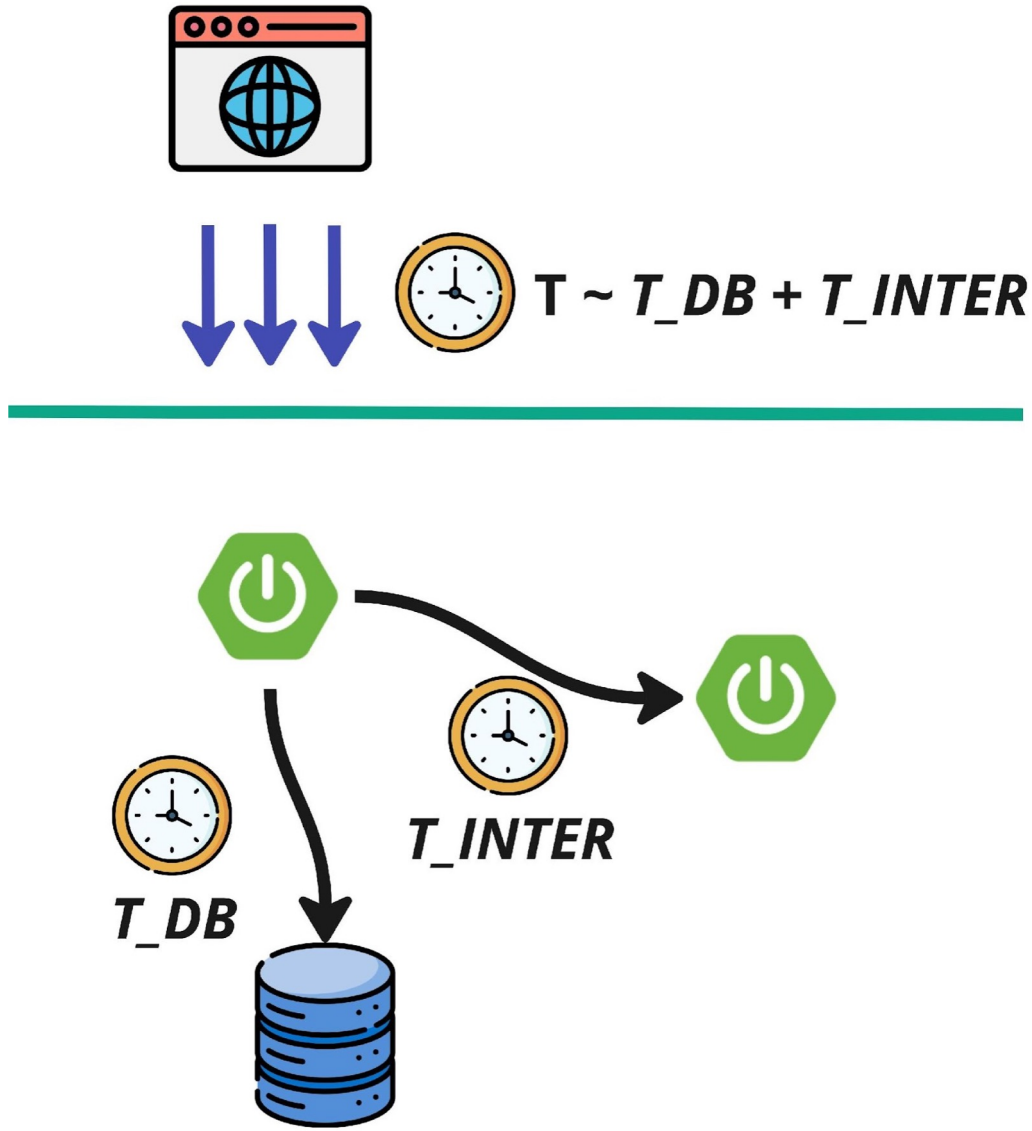
```
sum by(logger_name, short_message) (
  count_over_time(
    {cluster="prod", namespace="mm-core-bff", container="mm-core-bff", level="error"}
    | json
    | line_format "{{.logger_name}} | {{.message}}"
    | error!="JSONParserErr"
    | regexp "^(?P<loggerName>[^|]+)\\|(?P<short_message>[^:|]{1,45})"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

# Интересный факт про LogQL #3



# Actuator умеет больше, чем кажется #1

# Actuator умеет больше, чем кажется #1



# Метрики запросов в базу данных

| Метрика | Что измеряет |
| --- | --- |
| spring_data_repository_invocations | количество вызовов методов Spring Data репозиториев |
| spring_data_repository_invocations_seconds | время выполнения методов репозиториев (latency) |
| spring_data_repository_invocations_seconds_count | количество вызовов методов |
| spring_data_repository_invocations_seconds_sum | суммарное время выполнения |

# Метрики http клиентов

| Метрика | Что измеряет |
| --- | --- |
| http_client_requests_seconds | latency HTTP-клиентов |
| http_client_requests_seconds_count | количество исходящих HTTP-запросов |
| http_client_requests_seconds_sum | 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

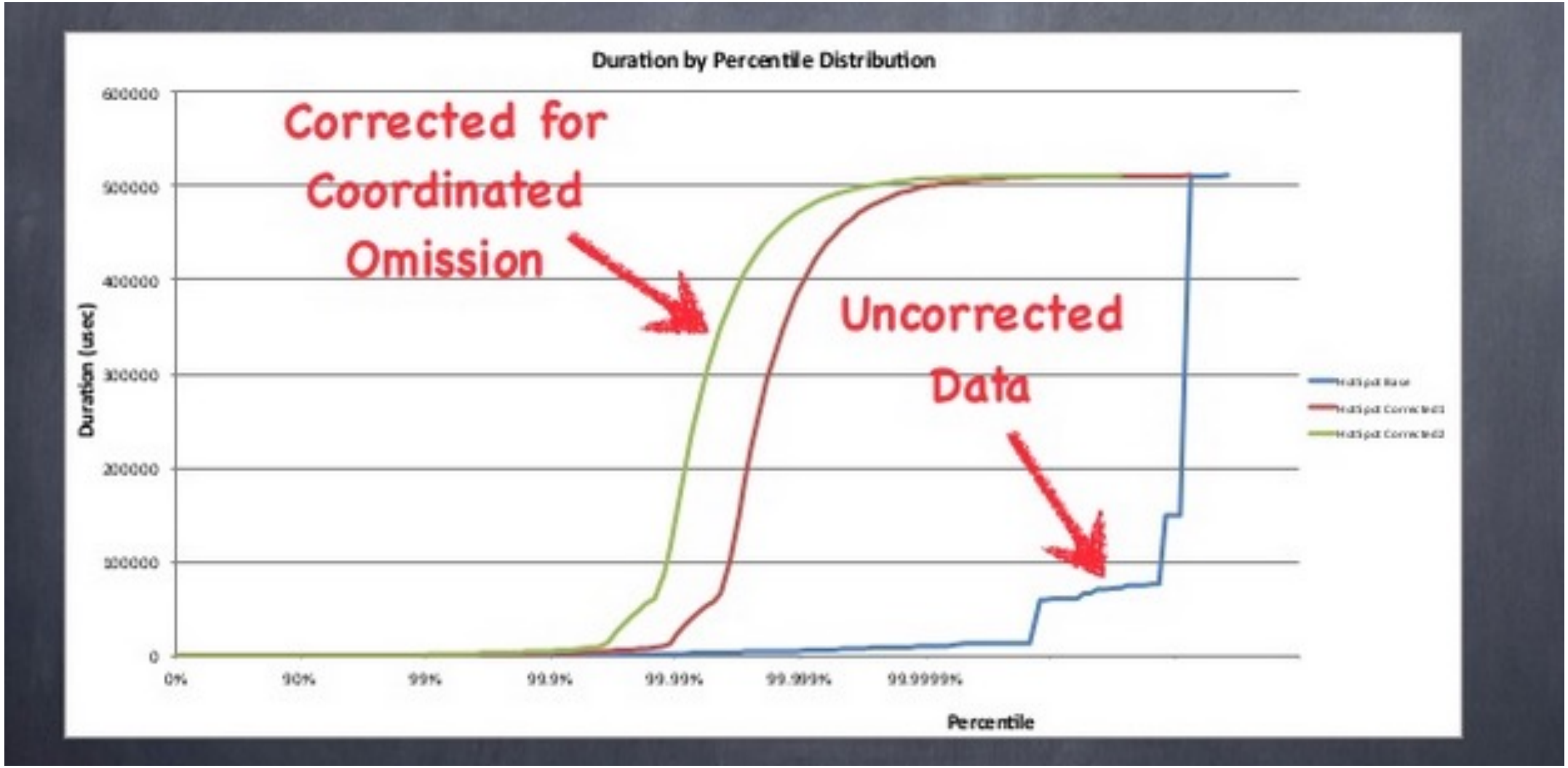
- Содержит инструментальные **SPI** (Service Provider Interface) для поддержки различных систем мониторинга

- 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);
```

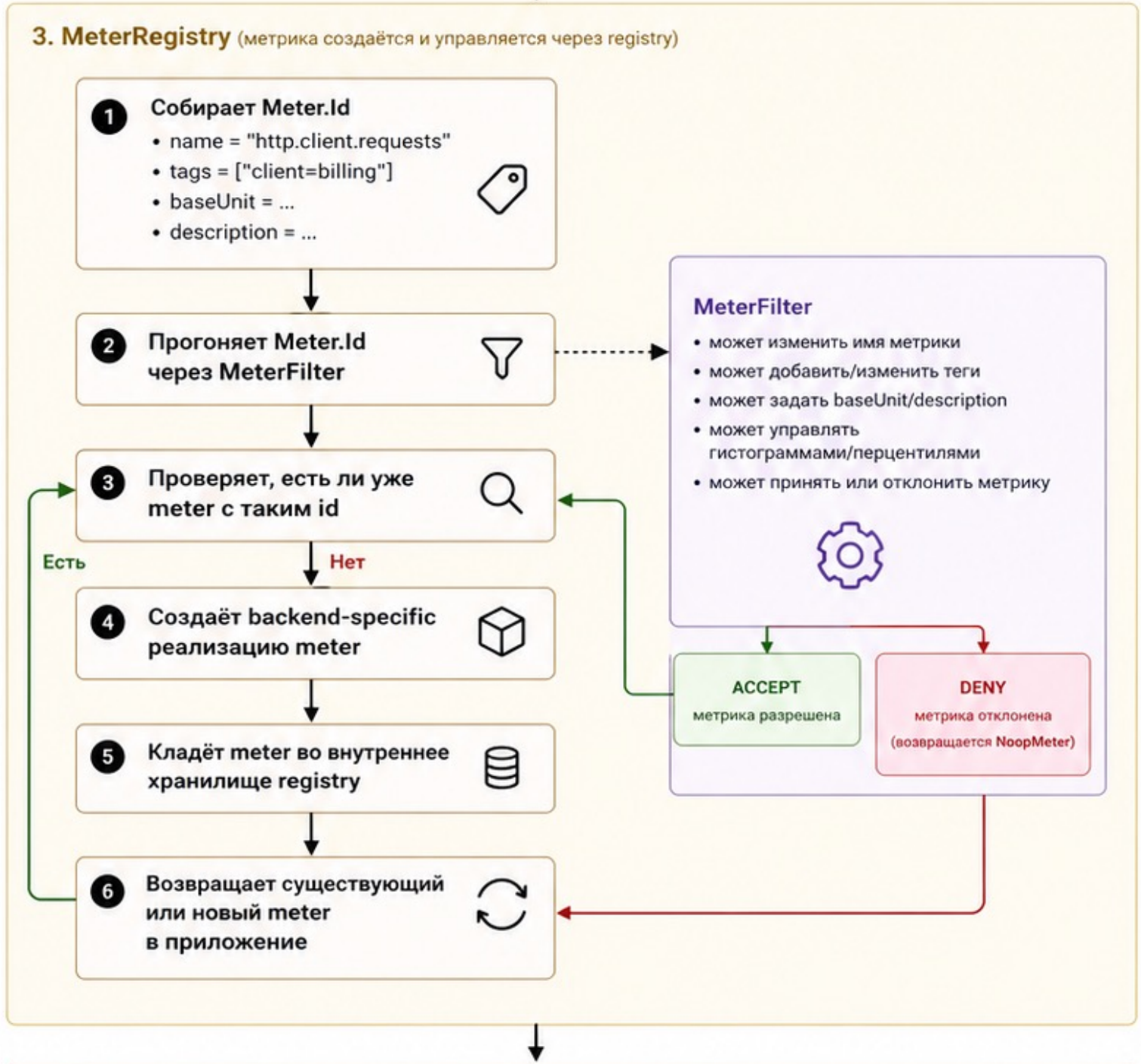
## 2. Micrometer API

Counter.builder()

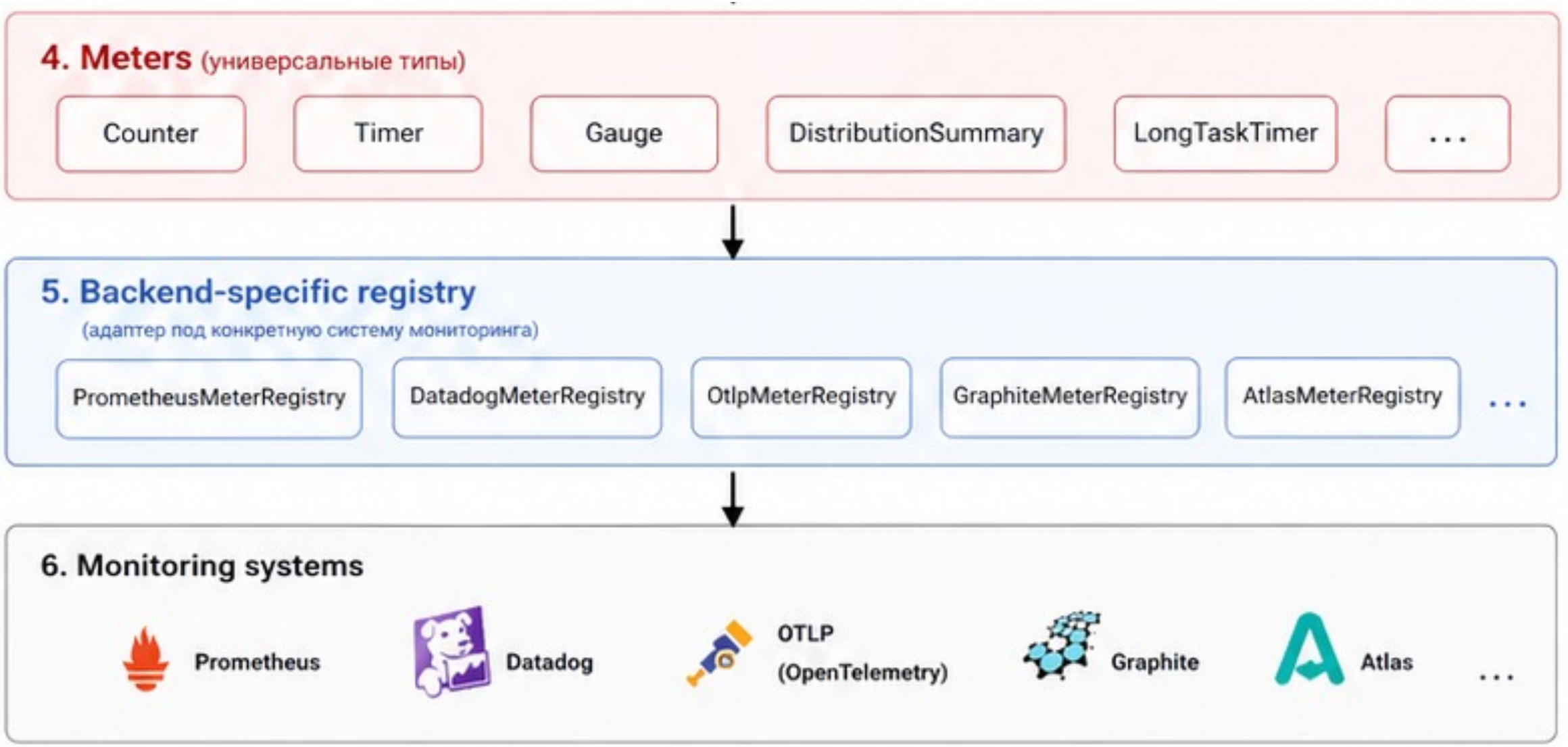
Timer.builder()

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
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
MeterFilter
• может изменить имя метрики
• может добавить/изменить теги
• может задать baseUnit/description
• может управлять гистограммами/перцентилями
• может принять или отклонить метрику
ACCEPT
метрика разрешена
DENY
метрика отклонена
(возвращается NoopMeter)

# Флоу формирования метрик



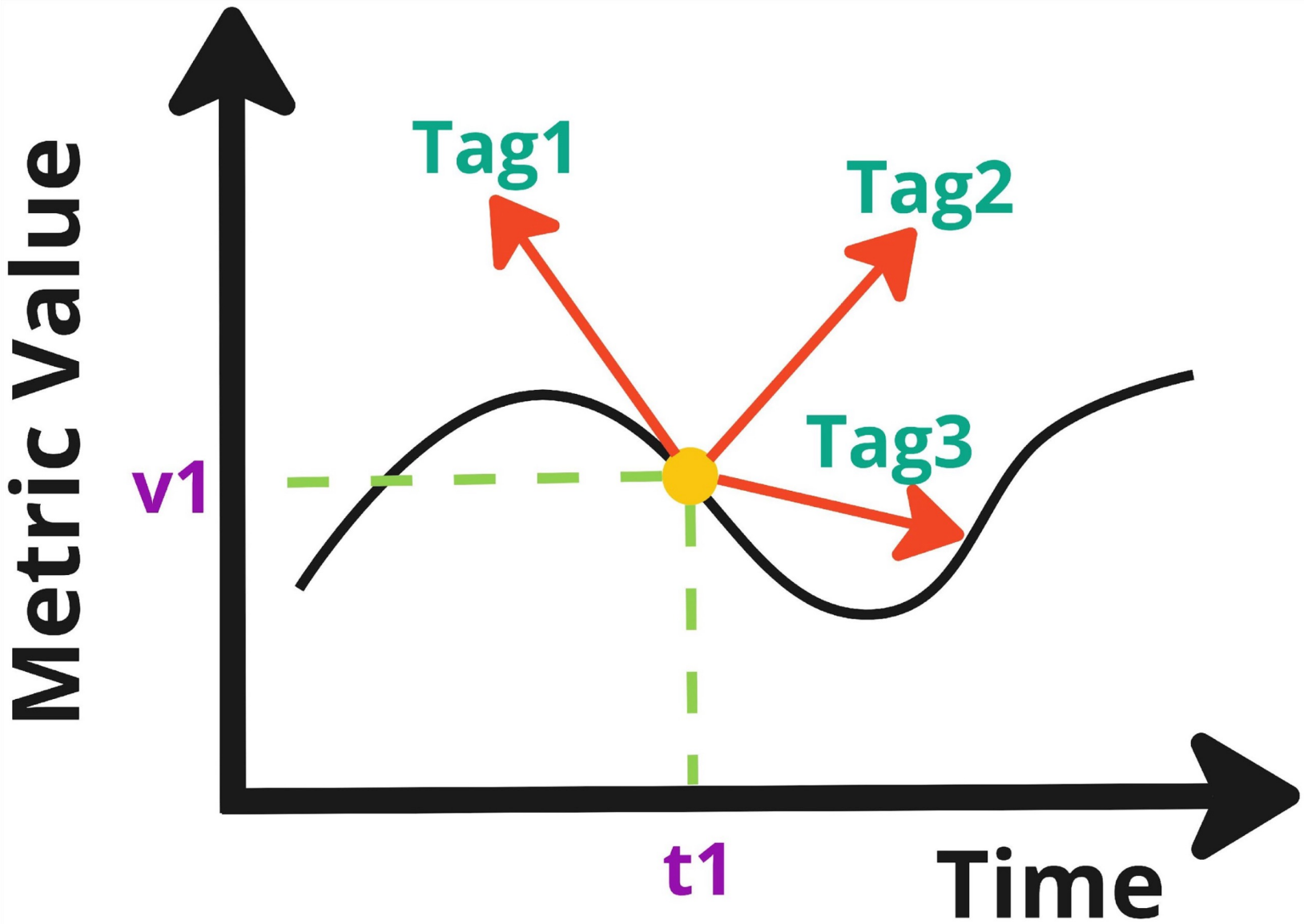
# Micrometer API

- ***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 растет

# Counter

```java
@RestController
public class HelloController {

    private final Counter counter;

    public HelloController(MeterRegistry registry) {
        this.counter = Counter.builder("hello.requests.count")
                              .description("Total number of Hello requests")
                              .tags("hello", "counterexample")
                              .register(registry);
    }

    @GetMapping("/hello")
    public String handleRequest() {
        counter.increment();
        return "hello";
    }
}
```

# 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

```java
@RestController
public class ListController {

    private List<String> list = Collections.synchronizedList(new LinkedList<>());

    public ListController(MeterRegistry registry) {
        Gauge.builder("list.size", list, List::size)
             .description("The size of the list")
             .register(registry);
    }

    @GetMapping("/add")
    public String addItem() {
        list.add("item");
        return "Added item, list size now " + list.size();
    }

    @GetMapping("/remove")
    public String removeItem() {
        if (!list.isEmpty()) {
            list.remove(0);
        }
        return "Removed item, list size now " + list.size();
    }
}
```

# 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

```java
@RestController
public class HelloController {

    private final Timer responseTimer;

    public HelloController(MeterRegistry registry) {
        this.responseTimer = Timer.builder("hello.response.time")
                                  .description("Time taken to return a response")
                                  .register(registry);
    }

    @GetMapping("/hello")
    public String handleRequest() {
        return responseTimer.record(() -> {
            try {
                TimeUnit.MILLISECONDS.sleep(200); // задержка 200 мс
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "hello";
        });
    }
}
```

# 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

```java
@Component
@RequiredArgsConstructor
public class HttpMetricsFilter extends OncePerRequestFilter {

    private final MeterRegistry meterRegistry;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
            throws ServletException, IOException {

        String uri = request.getRequestURI();

        Counter.builder("http_requests_total")
            .tag("uri", uri)
            .tag("method", request.getMethod())
            .register(meterRegistry)
            .increment();

        filterChain.doFilter(request, response);
    }
}
```

# Динамический URL в тэгах

```java
@Component
@RequiredArgsConstructor
public class HttpMetricsFilter extends OncePerRequestFilter {

    private final MeterRegistry meterRegistry;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
            throws ServletException, IOException {

        String uri = request.getRequestURI();

        Counter.builder("http_requests_total")
            .tag("uri", uri)
            .tag("method", request.getMethod())
            .register(meterRegistry)
            .increment();

        filterChain.doFilter(request, response);
    }
}
```

# Как сломать cardinality в бизнес-метриках

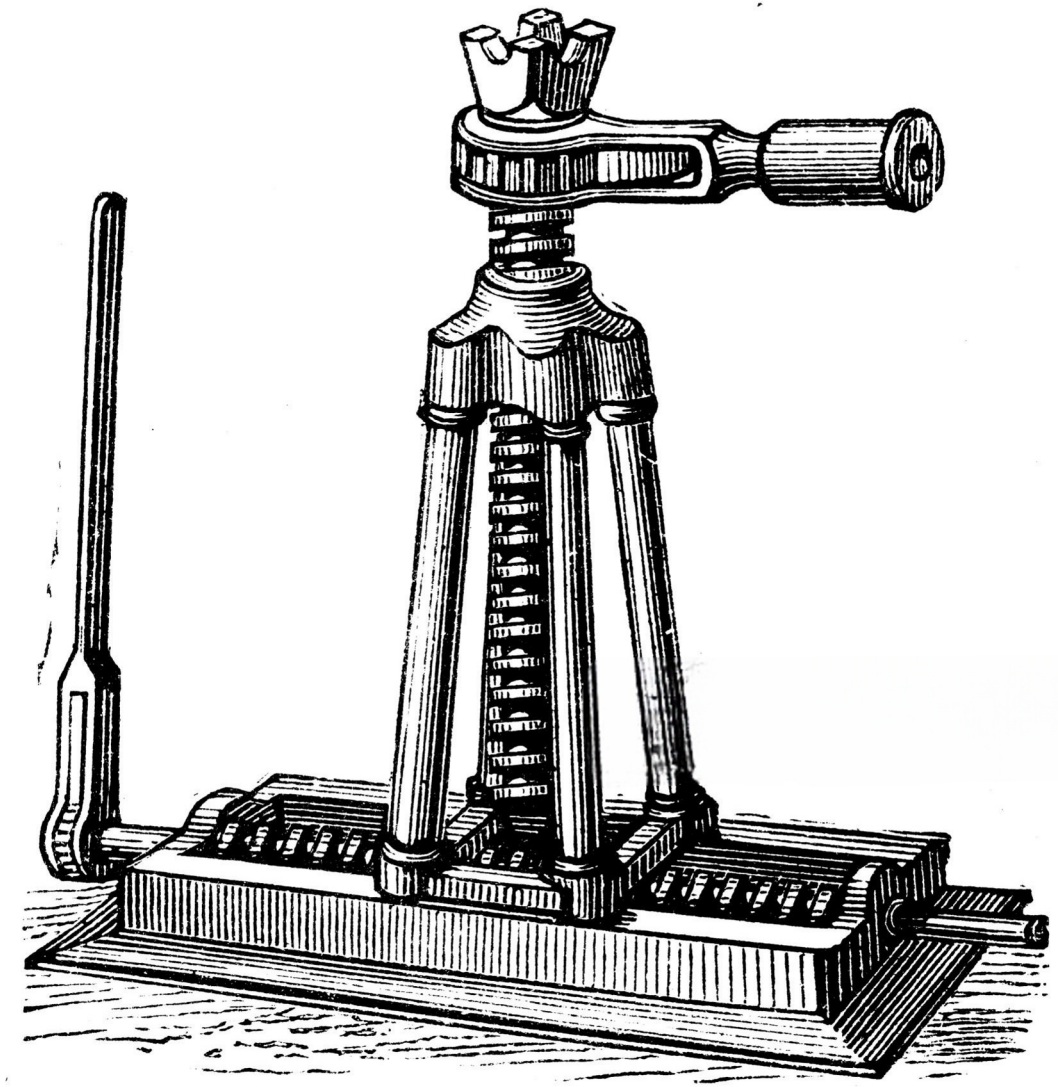
```java
@Service
@RequiredArgsConstructor
public class PaymentMetricsService {

    private final MeterRegistry meterRegistry;

    public void recordPayment(String orderId, String email, String externalStatus)
{
        Timer.builder("payment_processing_seconds")
            .tag("orderId", orderId)
            .tag("email", email)
            .tag("externalStatus", externalStatus)
            .register(meterRegistry)
            .record(() -> processPayment(orderId));
    }

    private void processPayment(String orderId) {
        // business logic
    }
}
```

# Как сломать cardinality в бизнес-метриках

```java
@Service
@RequiredArgsConstructor
public class PaymentMetricsService {

    private final MeterRegistry meterRegistry;

    public void recordPayment(String orderId, String email, String externalStatus)
{
        Timer.builder("payment_processing_seconds")
            .tag("orderId", orderId)
            .tag("email", email)
            .tag("externalStatus", externalStatus)
            .register(meterRegistry)
            .record(() -> processPayment(orderId));
    }

    private void processPayment(String orderId) {
        // business logic
    }
}
```

# 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

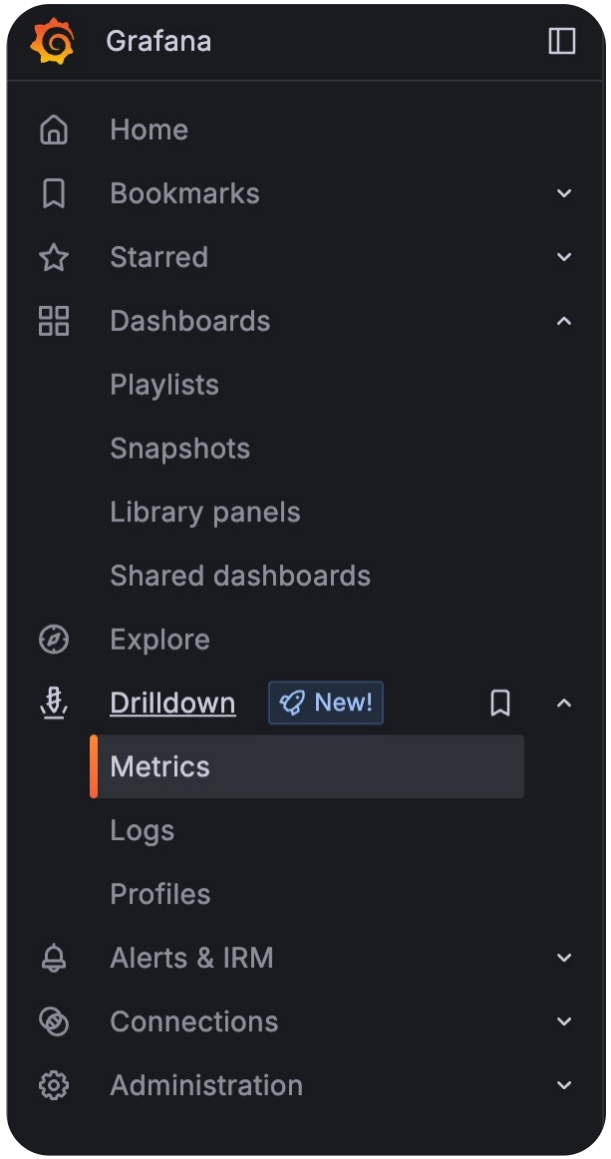
- Spring Boot включает в себя ряд встроенных actuator endpoints и позволяет вам добавлять свои собственные. Например, **GET /actuator/prometheus** публикует метрики для Prometheus

# 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

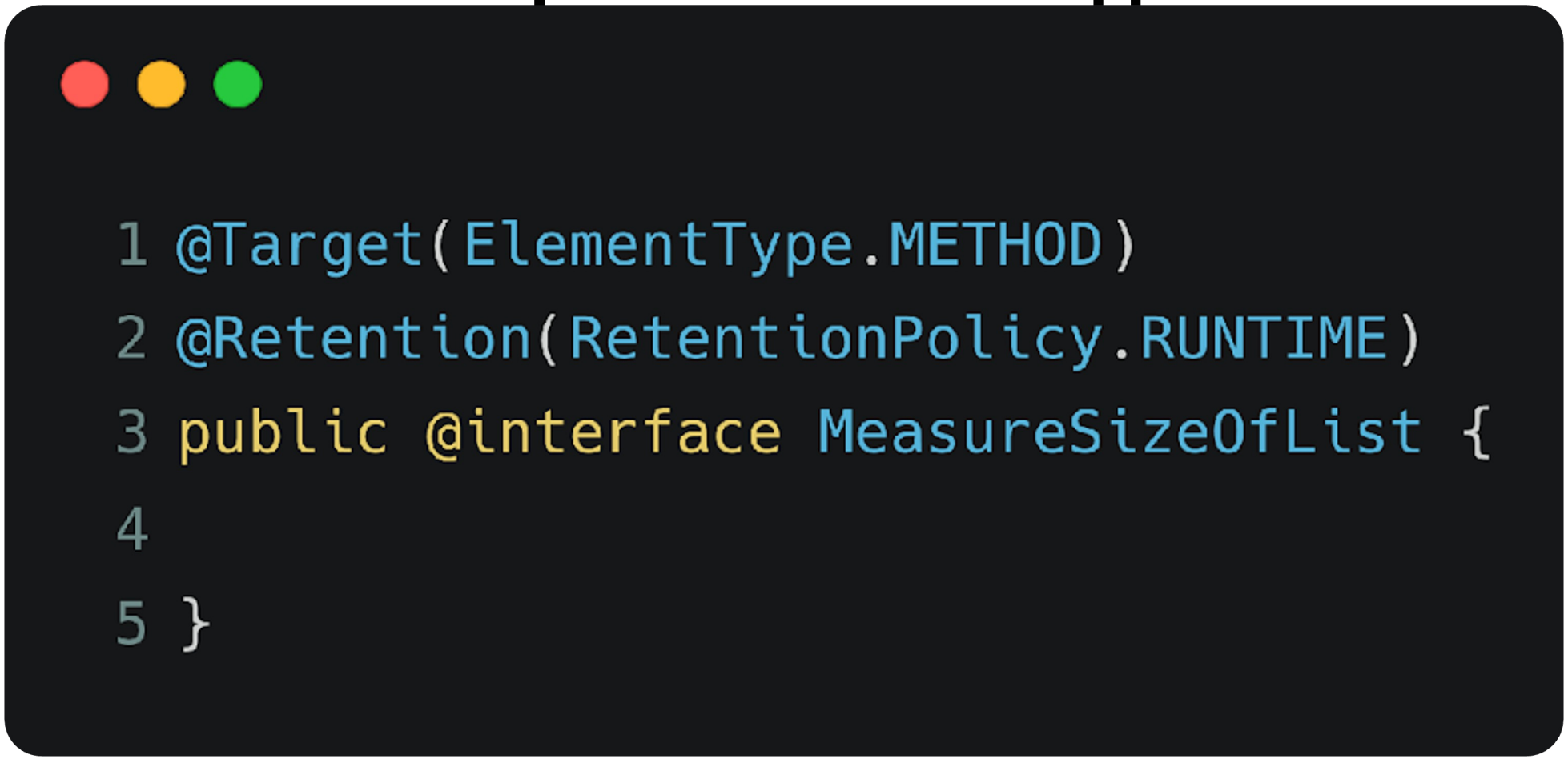
# Почему не профайлинг или анализ дампов

# *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

```java
@Aspect
@Component
@RequiredArgsConstructor
public class AspectMeasuringSizeOfList {

    private final MeterRegistry meterRegistry;

    @Around("@annotation(MeasureSizeOfList)")
    public Object measureSizeOfList(ProceedingJoinPoint joinPoint) throws Throwable {
        Object result = joinPoint.proceed();

        if (result instanceof BaseListDto<?> list) {
            HttpServletRequest request = ((ServletRequestAttributes)
RequestContextHolder.currentRequestAttributes()).getRequest();

            String sourceIp = request.getRemoteAddr();

            meterRegistry.counter("list_endpoints_size", "source_ip", sourceIp).increment(list.getItems().size())
        }

        return result;
    }
}
```

# *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

```java
@Aspect
@Component
public class TransactionalTimerAspect {

    private final MeterRegistry meterRegistry;

    @Autowired
    public TransactionalTimerAspect(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    @Pointcut("@annotation(org.springframework.transaction.annotation.Transactional)")
    public void transactionalMethods() {
    }

    @Around("transactionalMethods()")
    public Object measureMethodExecutionTime(ProceedingJoinPoint pjp) throws Throwable {
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        String methodName = signature.getMethod().getName();

        Timer.Sample sample = Timer.start(meterRegistry);
        try {
            return pjp.proceed();
        } finally {
            sample.stop(Timer.builder("transactions.timer")
                    .description("Duration of transactional methods")
                    .tags("method", methodName)
                    .register(meterRegistry));
        }
    }
}
```

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

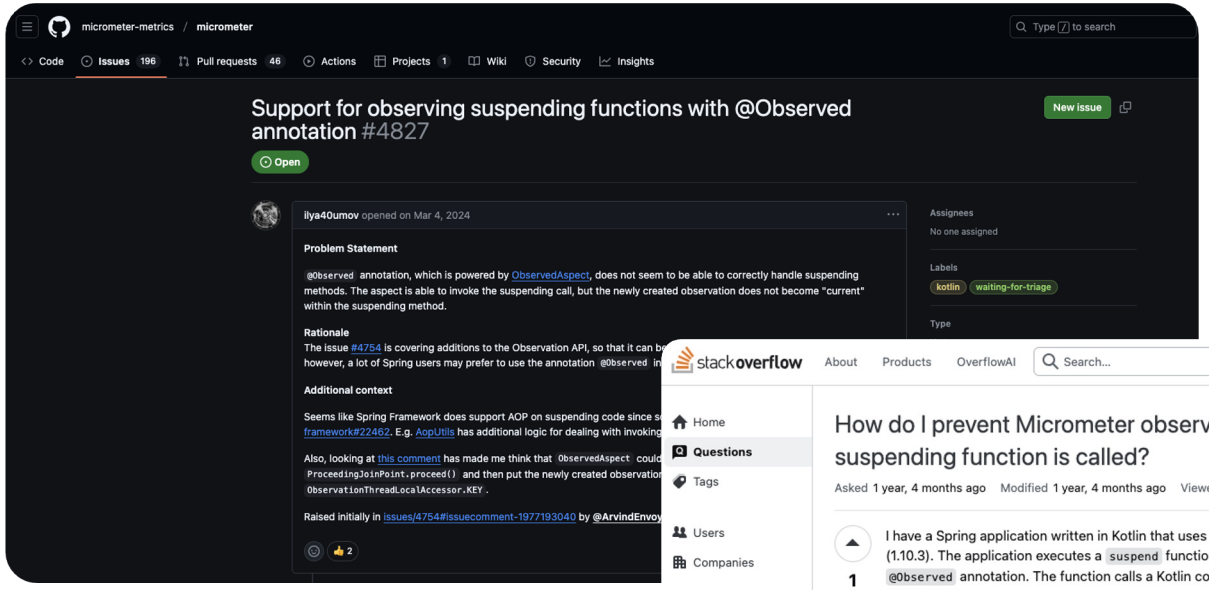
```java
@Aspect
@Component
public class HibernateStatisticsAspect {

    private final Statistics statistics;
    private final MeterRegistry meterRegistry;

    @Autowired
    public HibernateStatisticsAspect(SessionFactory sessionFactory, MeterRegistry meterRegistry) {
        this.statistics = sessionFactory.getStatistics();
        this.meterRegistry = meterRegistry;
    }

    @Pointcut("@annotation(transactional)")
    public void transactionalMethods(Transactional transactional) {
    }

```

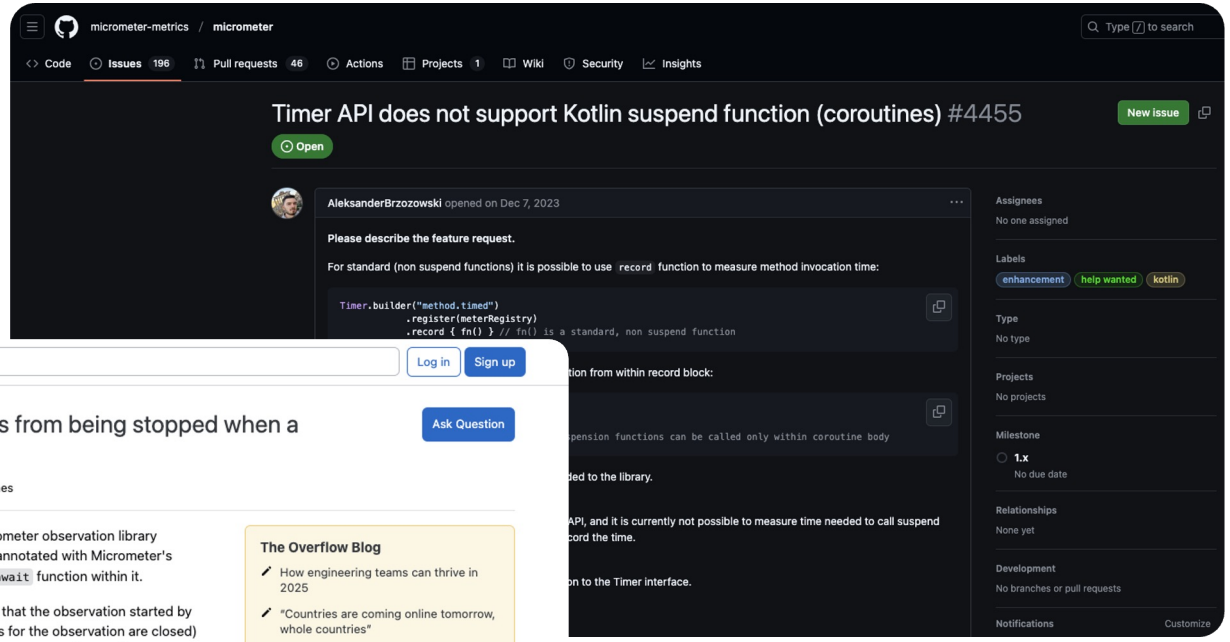
*продолжение на следующем слайде ->*

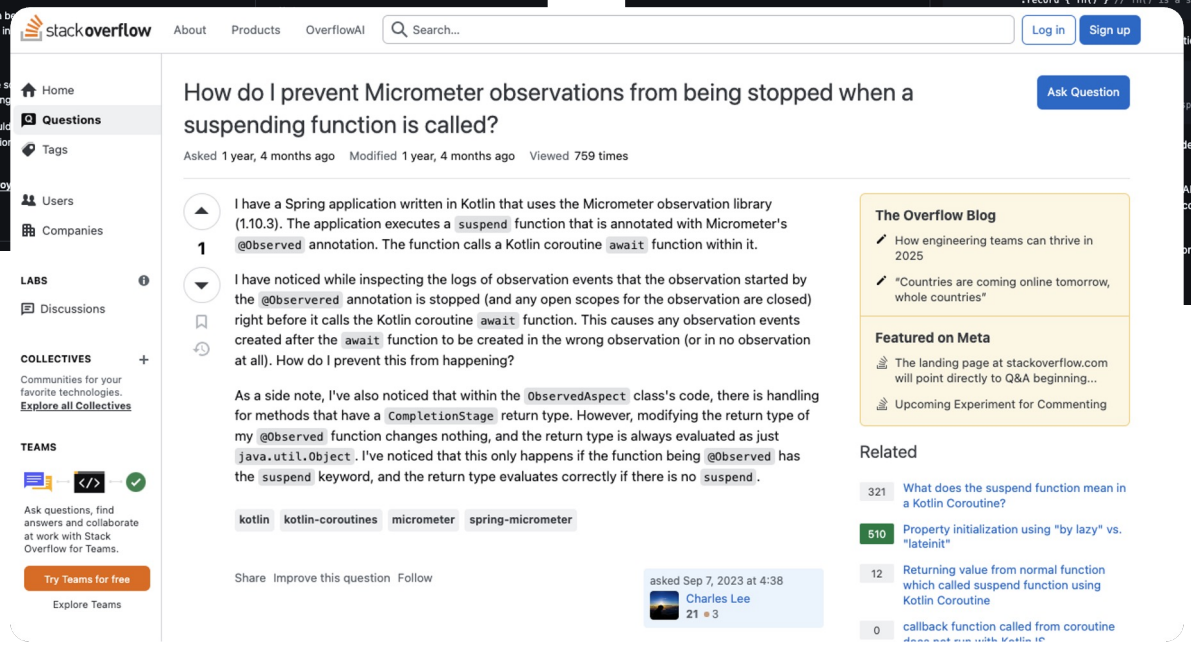
# Аспект измеряющий длительность CRUD-операций с БД на основе статистики Hibernate (Actuator не выставляет такие метрики)

```
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



# *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  }
```

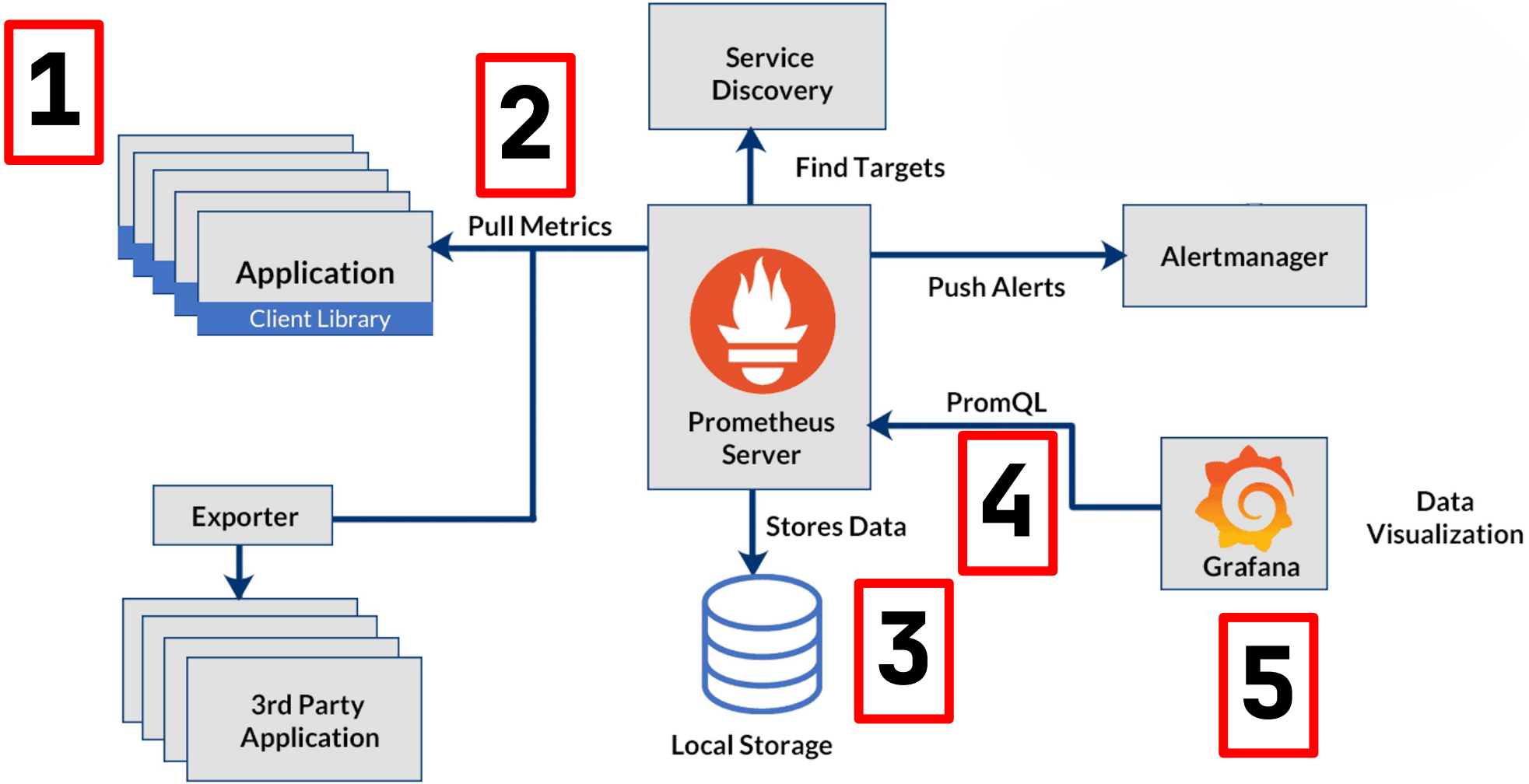
# Теперь можно замерять suspend'ы

```
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

# Grafana 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

```
histogram_quantile(

  0.95,

  sum by (le, uri) (

    rate(http_server_requests_seconds_bucket[5m])

  )

)
```

# 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