Руководство прикладного разработчика

компонента Объединенный мониторинг Unimon (код компонента: MONA)

продукта Platform V Monitor (код продукта: OPM)

ОГЛАВЛЕНИЕ

Руководство прикладного разработчика

Документ предназначен для персонала, осуществляющего подключение, конфигурирование, миграцию на текущую версию и разработку приложений с использованием программного компонента Объединенный мониторинг Unimon (MONA) в составе продукта Platform V Monitor (OPM), далее Unimon.

Unimon предназначен для сбора разных типов метрик:

- метрики, отражающие информацию о работе приложений и оборудования опорной инфраструктуры;
- метрики для контроля бизнес-показателей, отражающие активность и опыт конечного пользователя или конечной точки подключения. Позволяют количественно и качественно оценить качество сервиса;
- метрики для тестирования и разработки приложений, статусы и состояния модулей системы, получаемые с различных интерфейсов, различные технические логи и т.д.

Собранные метрики могут быть визуализированы с помощью Grafana.

Термины и определения

Расшифровка терминов указана в общих документах на продукт.

Системные требования

Перед началом работы необходимо установить Unimon. Подробнее про установку и системные требования к установке см. Руководство по установке.

Когда Unimon установлен, к нему можно подключить прикладное приложение. Приложение может быть написано на любом языке. Основные требования к приложениям: возможность запуска в среде контейнеризации Kubernetes или Red Hat OpenShift 4+ (опционально) и наличие выставленных http endpoint с метриками по стандарту Prometheus. Можно использовать любые возможности для публикации метрик по стандарту Prometheus.

Пример метрики simple_counterWithTags_total по стандарту Prometheus:

```
# HELP simple_counterWithTags_total Just a simple counter with tags # TYPE simple_counterWithTags_total counter simple_counterWithTags_total{terbank="sib",vsp="111",} 3.0 simple_counterWithTags_total{terbank="msk",vsp="111",} 4.0
```

, где simple_counterWithTags_total — имя метрики, {terbank="sib",vsp="111",} — labels — метки метрики, 3.0 — value — значение метрики.

Подключение и конфигурирование

Перед тем как подключить Unimon, необходимо создать подключение. В результате будет получен UnimonId — идентификатор подключения к Unimon. Подробнее о создание подключения см. Руководство по администрированию. UnimonId должен присутствовать в

каждой метрике в качестве label. По label UnimonId определяется соответствующий топик Kafka, в который отправляются метрики. Подробное описание приведено ниже.

Мультитенантность

Для удобства потребителей в Unimon реализована возможность маршрутизации метрик в нужный индекс хранилища Abyss на основе идентификатора UnimonId.

Ключ	Описание
rn	Собственный resourceName (ресурс уровня проекта), используемый для интеграции с мониторингом. Обязательное поле, но есть возможность для обратной совместимости публиковать метрику без rn (вариант отправки в метрики в топик по умолчанию).
UnimonId	Идентификатор подключения к мониторингу, соответствует определенному месту хранения метрики. Обязательное поле, но есть возможность для обратной совместимости публиковать метрику без UnimonId (вариант отправки в метрики в топик по умолчанию).

При выполнении метода по созданию подключения в БД мониторинга записывается соответствие rn → UnimonId → объекты хранилище Abyss. При создании подключения, в качестве ответа возвращается идентификатор UnimonId. Первое подключение по проекту по умолчанию определяется, как основное. Это позволяет определить маршрут для метрик в случае, если передан только rn, т.е. отправлять метрики по основному подключению к мониторингу. Для этого потребитель должен подключиться к Unimon и получить индивидуальный UnimonId, который соответствует топику Kafka Abyss.

В общем случае для одного заказчика будет достаточно одного (основного) подключения Unimon. Дополнительные подключения могут быть созданы для случаев, когда нужно разделить метрики одного заказчика по месту хранения. Это может быть полезно, если метрики сильно отличаются по набору labels (меток) или у заказчика есть какое-то внутренне деление источников метрик. Например, можно выделить отдельный проект (namespace) для технических сервисов платформы, создать для них один rn и сделать подключения для каждого технического сервиса. Тогда метрики каждого сервиса будут храниться отдельно, что оптимизирует их хранение и улучшит скорость работы.

Для обратной совместимости останется возможность отправки метрик в заранее определенный топик Kafka по умолчанию (автономный режим работы), в этом случае rn может добавляться в метрику, но маршрутизация по UnimonId не выполняется. Выполнять процедуру создания подключения в этом случае не надо, но часть возможностей Unimon будут недоступны, например фильтрация. Если настройка на топик не указана и в метрике отсутствуют UnimonId, либо rn, то метрика не будет сохранена.

Пример иерархии:

rn	UnimonId	main_connection
Потребитель-1	connect1_1	true
Потребитель-1	connect1_2	false
Потребитель-2	connect2_1	true
Потребитель-3	connect3_1	true

Функционал маршрутизации начинает работать при выполнении условий:

1. Выполнено подключение к Unimon.
2. В метрику добавлен label идентификатор rn или UnimonId (можно указать оба идентификатора или какой-то один из них). При этом в случае добавления только rn отправка идет только по «основному» подключению Unimon.

Существует несколько вариантов использования Unimon

- **Подключение потребителей на Виртуальных машинах** Для подключения Unimon, требуется получить идентификатор подключения UnimonId в рамках своего проекта (rn) и указывать rn и UnimonId в качестве label в targets.json. Если UnimonId не указывать, отправка метрики будет производиться в топик по умолчанию (параметр Kafka.topic в namespace мониторинга). Если параметр Kafka.topic будет не заполнен либо отсутствовать, метрика будет игнорироваться. Для того чтобы метрики с виртуальных машин отображались на дашбордах в Grafana, а также для идентификации потребителя, необходимо чтобы для каждого набора таргетов присутствовал label app и version, в котором указывается имя приложения, публикующего метрики. Пример targets.json:

```
[       {
   "labels": {
                "app": "app1",
                "version": "1.0.0",
                "UnimonId": "tenant1",
                "rn": "loga",
                "__metrics_path__": "/metrics"          },
 "targets": ["10.53.6.109:9100", "10.53.6.110:9100"]
 }
]
```

- **Подключение через API** Предоставляется возможность отправлять метрики через API напрямую в приложение Unimon-sender. Unimon-sender устанавливается в namespace клиента или отдельно в namespace системы мониторинга. Подробное описание API в /apis/. Прием метрик осуществляется в нескольких форматах:
  - json - метод API - /external/send
  - текстовый формат Prometheus - метод API - /push (функциональность по умолчанию включена, можно отключить с помощью параметра unimon-sender.push-api.enabled)
  - xxx-from-url-encoded - метод API - /push/encoded (функциональность по умолчанию включена, можно отключить с помощью параметра unimon-sender.push-api.enabled)

  Данная возможность предоставляется для агрегированных метрик или метрик с короткоживущих процессов, а также для приложений, развернутых не в среде

контейнеризации. Для отображения метрик в UI (в том числе для фильтрации), а также для корректной маршрутизации необходимо добавить в метрики обязательные labels:

- app - имя приложения;
- version - версия приложения;
- UnimonId - идентификатор подключения к unimon;
- rn (для API, который не содержит данный параметр; для API, который в параметре содержит rn - в labels добавляется автоматически).

В остальных случаях рекомендуется использовать вариант подключения 3.

- **Подключение для приложений в среде контейнеризации** Для приложений, которые могут запускаться в среде контейнеризации Kubernetes или Red Hat Openshift 4+ (опционально) и выставлять http endpoint с метриками по стандарту Prometheus, необходимо выполнить следующие настройки:

1. Настройки в манифестах в среде контейнеризации Большинство обязательных labels метрики будут добавлены автоматически на этапе сбора Unimon-agent. Но в прикладное приложение необходимо самостоятельно добавить следующие labels:
   - app - имя приложения;
   - version - версия приложения; Их необходимо добавить в labels POD вашего приложения.
2. Добавление в метрики UnimonId, rn (необходимо при неавтономном режиме работы) Для того, чтобы значения UnimonId, rn появились во всех метриках пользователя есть несколько вариантов.
   - UnimonId задать, как label в pod'е прикладного сервиса.
   - UnimonId/rn задать, как label в метрике.
   - rn задать в файле конфигурации Unimon-sender (metric.label.rn=).

Варианты указаны в соответствии с приоритетностью применения значений. То есть, если метка задана первым и последним вариантом и значения не равны, то в метрики будет добавлено значение, указанное в первом варианте. Также, если указан несуществующий UnimonId/rn в метриках, выводится сообщение в лог (режим debug), что эти метрики не будут направлены в хранилище.

Если в метрике указан только rn и параметр работы Unimon указан, как автономная работа, метрика будет отправлена в топик по умолчанию, указанный в Unimon-sender.

Если в метрике указан только rn, параметр указан, как неавтономная работа и на текущий момент отсутствует связь с сервером, то есть невозможно получить основной UnimonId для rn и смаршрутизировать в соответствующий топик, метрика не будет отправлена!

Пример Deployment.yaml

Порты и названия приложений могут меняться на стороне потребителя.

```
apiVersion: apps/v1 kind: Deployment metadata: name: unimon-sandbox-micrometer annotations: description: Defines how to deploy Micrometer-Prometheus App labels: app: unimon-sandbox-micrometer # имя прикладного приложения version: 1.0.0 # версия прикладного приложения testLabel: test spec: selector: matchLabels: app: unimon-sandbox-micrometer version: 1.0.0 replicas: 1 strategy: type: Recreate recreateParams: timeoutSeconds: 600 activeDeadlineSeconds: 21600 template: metadata: name: unimon-sandbox-micrometer annotations: description:
```

Micrometer Prometheus App pod labels: app: unimon-sandbox-micrometer version: 1.0.0 testLabel: test spec: containers: - name: unimon-sandbox-micrometer image: "registry.sigma.sbrf.ru/efs/ci01976100/ci02702773_opm_unimon/unimon-sandbox-micrometer:DEV-SNAPSHOT" imagePullPolicy: IfNotPresent ports: - name: http containerPort: 8080 - name: actuator containerPort: 8081 readinessProbe: httpGet: path: /actuator/health port: 8081 scheme: HTTP periodSeconds: 3 timeoutSeconds: 3 failureThreshold: 3 initialDelaySeconds: 30 resources: limits: cpu: 1 memory: 1Gi requests: cpu: 500m memory: 512Mi restartPolicy: Always

3. Настройка DevOps для того, чтобы Unimon-agent начал собирать метрики с вашего приложения, нужно добавить аннотации prometheus.io в ваш Service. При конфигурации сервиса в среде контейнеризации необходимо указать следующие параметры:

Пример Service.yaml

apiVersion: v1 kind: Service metadata: # Пример названия сервиса name: unimon-sandbox-micrometer annotations: description: "Exposes Micrometer-Prometheus App by CLuster Ip" # Аннотация для включения сбора метрик prometheus.io.scrape: "true" # Аннотация для указания HTTP endpoint с метриками приложения # Путь указывается полностью, например если при тестирование # вы переходите на endpoint http://localhost:8081/actuator/prometheus prometheus.io.path: "/actuator/prometheus" # Аннотация для указания порта подключения к HTTP endpoint с метриками # Внимание!!! порт может меняться на стороне потребителя, "8081" указан как пример prometheus.io.port: "8081" labels: app: unimon-sandbox-micrometer spec: ports: - name: http port: 8080 targetPort: 8080 - name: http-actuator port: 8081 targetPort: 8081 selector: app: unimon-sandbox-micrometer

Маршрутизация метрик

Наличие rn в метрике перед отправкой в хранилище	**Наличие UnimonId в метрике перед отправкой в хранилище**	**Kafka_topic дефолтный топик в настройках Unimon-sender**	**Маршрутизация метрики**
+	-	есть	Если установлен флаг автономной работы, то метрика будет направлена в дефолтный топик, иначе метрика не пройдет валидацию (не будет отправлена в хранилище)

+	+	есть	Метрика будет направлена в топик, соответствующий UnimonId
-	+	есть	Метрика будет направлена в топик, соответствующий UnimonId
-	-	есть	Если установлен флаг автономной работы, то метрика будет направлена в дефолтный топик, иначе метрика не пройдет валидацию (не будет отправлена в хранилище)
+	-	нет	Метрика не пройдет валидацию (не будет отправлена в хранилище)
+	+	нет	Метрика будет направлена в топик, соответствующий UnimonId
-	+	нет	Метрика будет направлена в топик, соответствующий UnimonId
-	-	нет	Метрика не пройдет валидацию (не будет отправлена в хранилище)

На текущий момент для метрик заданы следующие правила валидации:

- Проверка соответствия rn и UnimonId. Если они относятся к разным проектам, метрика не будет проходить валидацию.
- Проверка наличия UnimonId в метрике. Если метки нет и признак автономной работы не установлен, то метрика не будет проходить валидацию.

При отсутствии в метриках одного из значений rn/UnimonId, будет выполнена попытка обогащения метрик:

- Если не указан UnimonId, то будет выполнена попытка по rn найти его (по признаку основное подключение) и добавить в метрику.
- Если не указан rn, то будет выполнен поиск его в БД по указанному в метрике UnimonId, если найден то будет добавлен в метрику.
- Если в метрике и в БД Unimon нет rn, но в файле конфигурации Unimon-sender задан, он не будет добавлен в метрику.

Миграция на текущую версию

Для миграции на текущую версию дополнительные настройки по подключению Unimon не требуются.

Разработка первого приложения с использованием программного продукта

Ниже приведен пример выставления метрик в формате Prometheus из прикладного приложения на Spring Boot.

1. Зависимости

Добавить в Apache Maven зависимости на actuator и micrometr:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
 <dependency>
 <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-core</artifactId>
 </dependency>
<dependency>
<groupId>io.micrometer</groupId>
 <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

2. Настройки

В application.properties указать порт actuator (значение по умолчанию 8080) и фильтр включенных endpoint.

Пример application.properties

```
management.server.port=8081
management.endpoints.web.exposure.include=*
```

3. Создание прикладных метрик в коде

Создать прикладные метрики в проекте с помощью micrometer. Подробнее см.: https://www.baeldung.com/micrometer. Если будут выставлены метрики со значением NaN, то есть "value": "NaN", они будут собраны, но валидацию не пройдут и не будут направлены в хранилище.

Ниже приведены примеры различных типов метрик.

Пример метрики Counter

```
@RestController public class CounterController { private static final String COUNTER_WITH_TAG_NAME = "simple.counterWithTags"; private static final String COUNTER_WITH_TAG_DESCRIPTION = "Just a simple counter with tags"; private static final String COUNTER_WITH_USER_TAG_NAME = "simple.counterWithUserTag"; private static final String TAG_NAME_1 = "terbank"; private static final String TAG_NAME_2 = "vsp"; private static final String TAG_USER = "userText"; private static final String TAG_RN = "rn"; private static final String RN = "unimon-sandbox-micrometer"; private static final String TAG_UNIMON_UD = "UnimonId"; private static final String UNIMON_ID = "unimon-sandbox-micrometer"; @Autowired MeterRegistry meterRegistry; private Counter simpleCounter; private Counter simpleCounterWithTags; private Counter simpleCounterWithTags2; private Counter simpleCounterWithUserTag; /** * Инициальзация метрик типа Counter */ @PostConstruct public void init() { simpleCounter = Counter.builder("simple.counter") .description("Just a simple counter") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); simpleCounterWithTags = Counter.builder(COUNTER_WITH_TAG_NAME) .description(COUNTER_WITH_TAG_DESCRIPTION) .tag(TAG_NAME_1, "sib") .tag(TAG_NAME_2, "111") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); simpleCounterWithTags2 = Counter.builder(COUNTER_WITH_TAG_NAME) .description(COUNTER_WITH_TAG_DESCRIPTION) .tag(TAG_NAME_1, "msk") .tag(TAG_NAME_2, "111") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); } /** * Увеличение счетчика на 1 */ @GetMapping("/counter") @ResponseBody public String incrementCounter() { simpleCounter.increment(); return String.format("Counter has been increases\nCurrent value: %s", simpleCounter.count()); } /** * Увеличение на 1 счетчика с разрезом фиксации "terbank" и значением "sib" */ @GetMapping("/counterWithTags/terbank/sib") @ResponseBody public String incrementCounterTags1() { simpleCounterWithTags.increment(1); return String.format("Counter has been increases\nCurrent value: %s", simpleCounterWithTags.count()); } /** * Увеличение на 2 счетчика с разрезом фиксации "terbank" и значением "msk" */ @GetMapping("/counterWithTags/terbank/msk") @ResponseBody public String incrementCounterTags2() { simpleCounterWithTags2.increment(2); return String.format("Counter has been increases\nCurrent value: %s", simpleCounterWithTags2.count()); } @GetMapping("/counterWithUserTagValue") @ResponseBody public String incrementCounterWithUserTag(@RequestParam String val) { simpleCounterWithUserTag = Counter.builder(COUNTER_WITH_USER_TAG_NAME) .tag(TAG_USER, val) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry);
```

```
simpleCounterWithUserTag.increment(1); return String.format("Counter has been increases\nCurrent value: %s", simpleCounterWithUserTag.count()); } }
```

Пример метрики DistributionSummary

```
@RestController public class DistributionSummaryController { @Autowired MeterRegistry meterRegistry; private DistributionSummary simpleSummary; private static final String TAG_RN = "rn"; private static final String RN = "unimon-sandbox-micrometer"; private static final String TAG_UNIMON_UD = "UnimonId"; private static final String UNIMON_ID = "unimon-sandbox-micrometer"; /** * Инициальзация метрик типа DistributionSummary */ @PostConstruct public void init() { simpleSummary = DistributionSummary.builder("simple.distributionSummary") .description("Just a simple distributionSummary") .publishPercentiles(0.5, 0.75, 0.9) .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); } /** * Добавление случайного значения в множество */ @GetMapping("/distributionSummary") @ResponseBody public String addRecord() { simpleSummary.record(Math.random()); return String.format("value added to the distributionSummary \n Current count: %s", simpleSummary.count()); } }
```

Пример метрики Gauge

```
@RestController public class GaugeController { private static final String GAUGE_NAME = "simple.gauge"; private static final String GAUGE_DESCRIPTION = "Just a simple gauge"; private static final String GAUGE_WITH_TAG_NAME = "simple.gaugeWithTags"; private static final String GAUGE_WITH_TAG_DESCRIPTION = "Just a simple gauge with tags"; private static final String TAG_NAME_1 = "listName"; private static final String TAG_NAME_2 = "returnValue"; private static final String TAG_RN = "rn"; private static final String RN = "unimon-sandbox-micrometer"; private static final String TAG_UNIMON_UD = "UnimonId"; private static final String UNIMON_ID = "unimon-sandbox-micrometer"; @Autowired MeterRegistry meterRegistry; private Gauge simpleGauge; private Gauge simpleGaugeWithTags; private Gauge simpleGaugeWithTags2; private ServiceAvailabilityState availabilityState = new ServiceAvailabilityState(); private List<Integer> list1 = new ArrayList<>(4); private List<Integer> list2 = new ArrayList<>(4); /** * Инициальзация метрик типа Gauge */ @PostConstruct public void init() { simpleGauge = Gauge.builder(GAUGE_NAME, availabilityState, ServiceAvailabilityState::getStateFlag) .description(GAUGE_DESCRIPTION) .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); simpleGaugeWithTags = Gauge.builder(GAUGE_WITH_TAG_NAME, list1, List::size) .description(GAUGE_WITH_TAG_DESCRIPTION) .tag(TAG_NAME_1, "list1") .tag(TAG_NAME_2, "size") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); simpleGaugeWithTags2 = Gauge.builder(GAUGE_WITH_TAG_NAME, list2, List::size) .description(GAUGE_WITH_TAG_DESCRIPTION) .tag(TAG_NAME_1, "list2") .tag(TAG_NAME_2, "size") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); } /** * Установка значения метрики * @param val - значение метрики */ @GetMapping("/gauge") @ResponseBody public String setGauge(@RequestParam String val) { try { availabilityState.setStateFlag(Integer.valueOf(val)); } catch (NumberFormatException e) { return "Integer value should be provided to set this Gauge"; } return String.format("Gauge has been set\nCurrent value: %s", simpleGauge.value()); } /** * Добаляет к list1 значение, при этом изменяется значение метрики (размер списка) с разррезом ListName = list1 */ @GetMapping("/gaugeWithTags/listName/list1") @ResponseBody public String addElementList1() { list1.add(1); return String.format("Gauge has
```

```
been updated\nCurrent value: %s", simpleGaugeWithTags.value()); } /** * Добаляет к list2 значение, при этом изменяется значение метрики (размер списка) с разррезом ListName = list2 */ @GetMapping("/gaugeWithTags/listName/list2") @ResponseBody public String addElementList2() { list2.add(1); return String.format("Gauge has been updated\nCurrent value: %s", simpleGaugeWithTags2.value()); } } class ServiceAvailabilityState { Integer stateFlag = 0; public Integer getStateFlag() { return stateFlag; } public void setStateFlag(Integer stateFlag) { this.stateFlag = stateFlag; } }
```

Пример метрики Timer

```
@RestController public class TimerController { @Autowired MeterRegistry meterRegistry; private Timer simpleTimer; private Timer simpleTimerWithPercentile; private static final String TAG_RN = "rn"; private static final String RN = "unimon-sandbox-micrometer"; private static final String TAG_UNIMON_UD = "UnimonId"; private static final String UNIMON_ID = "unimon-sandbox-micrometer"; /** * Инициальзация метрик типа Timer */ @PostConstruct public void init() { simpleTimer = Timer.builder("simple.timer") .description("Just a simple timer") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .register(meterRegistry); simpleTimerWithPercentile = Timer.builder("simple.timerWithPercentile") .description("Just a simple timer with percentile") .tag(TAG_RN, RN) .tag(TAG_UNIMON_UD, UNIMON_ID) .publishPercentiles(0.5, 0.9, 0.99) .register(meterRegistry); } /** * Добавляет в метрику типа Timer значение длительности вызова sleep() */ @GetMapping("/timer") @ResponseBody public String runTimer() { simpleTimer.record(3000, TimeUnit.MILLISECONDS); return String.format("Timer has been run\nCurrent value: %s \nCurrentCount: %s", simpleTimer.totalTime(TimeUnit.MILLISECONDS), simpleTimer.count()); } /** * Добавляет в метрику типа Timer с включенной публикацией процентилей, значение длительности вызова sleep() */ @GetMapping("/timerWithPercentile") @ResponseBody public String runTimerWithPercentile() { simpleTimerWithPercentile.record(() -> { try { TimeUnit.SECONDS.sleep((int) (Math.random() * 4)); } catch (InterruptedException ignored) { } }); return String.format("Timer has been run\nCurrent value: %s \nCurrentCount: %s", simpleTimerWithPercentile.totalTime(TimeUnit.MILLISECONDS), simpleTimerWithPercentile.count()); } }
```

Подробнее с примерами можно ознакомиться в тестовом приложении Unimon-sandbox-micrometer. Приложение находится в дистрибутиве Unimon: samples/unimon-sandbox-micrometer.zip.

4. Настройки для подключения Unimon

Выполнить настройки из пункта «Подключение и конфигурирование». Просмотреть метрики, выставленные тестовым приложением в формате Prometheus можно по ссылке: http://localhost:8081/actuator/prometheus.

Использование программного продукта

Unimon предназначен для сбора метрик, которые затем могут быть визуализированы. Типы метрик, которые собирает Unimon:

- метрики, отражающие информацию о работе приложений и оборудования опорной инфраструктуры;

- метрики для контроля бизнес-показателей, отражающие активность и опыт конечного пользователя или конечной точки подключения. Позволяют количественно и качественно оценить качество сервиса;
- метрики для тестирования и разработки приложений, статусы и состояния модулей системы, получаемые с различных интерфейсов, различные технические логи и т.д.

Основной сценарий использования программного компонента Unimon описан в пункте **Разработка первого приложения с использованием программного продукта**. Поскольку назначение Unimon — сбор метрик, сценарии использования отличаются только типом собираемых метрик.

Часто встречающиеся проблемы и пути их устранения

Не отображаются метрики в Grafana

1. Проверить работоспособность Grafana. Например, выбрать больший временной период отображения метрик или зайти на другие дашборды. Если на дашбордах есть сообщения об ошибках (обычно в правом верхнем углу), это говорит о проблемах с Grafana. Если метрики или графики за любой период отображаются, Grafana доступна.
2. Убедиться, что метрики выставляются. Для проверки в терминале POD можно выполнить следующую команду:

```
curl 127.0.0.1:8081/actuator/prometheus
```

3. Проверить, что в Service вашего приложения указаны аннотации Prometheus.
4. Проверить работоспособность Unimon-agent и Unimon-sender по логам. Необходимо убедиться в отсутствии сообщений об ошибках в Pod **Unimon-sender** и **Unimon-agent**. В Pod **Unimon-sender** должна быть информация о входящих запросах. Например:

```
2020-10-23 07:42:04,194 [http-nio-0.0.0.0-8080-exec-23] [INFO]
(com.sbt.opsmon.unimon.api.MonitoringController)
[com.sbt.opsmon.unimon.api.MonitoringController::sendMetrics:50] mdc:()| Metrics
received: 90
```

Остальные шаги выполняются при наличии доступов в Kafka и Druid

5. Проверить метрики нужного приложения в топике.
6. Проверить метрики нужного приложения в Druid (работоспособность перекладчика Kafka → Druid).