

Мониторинг операционных систем и приложений

На примере Prometheus и Grafana

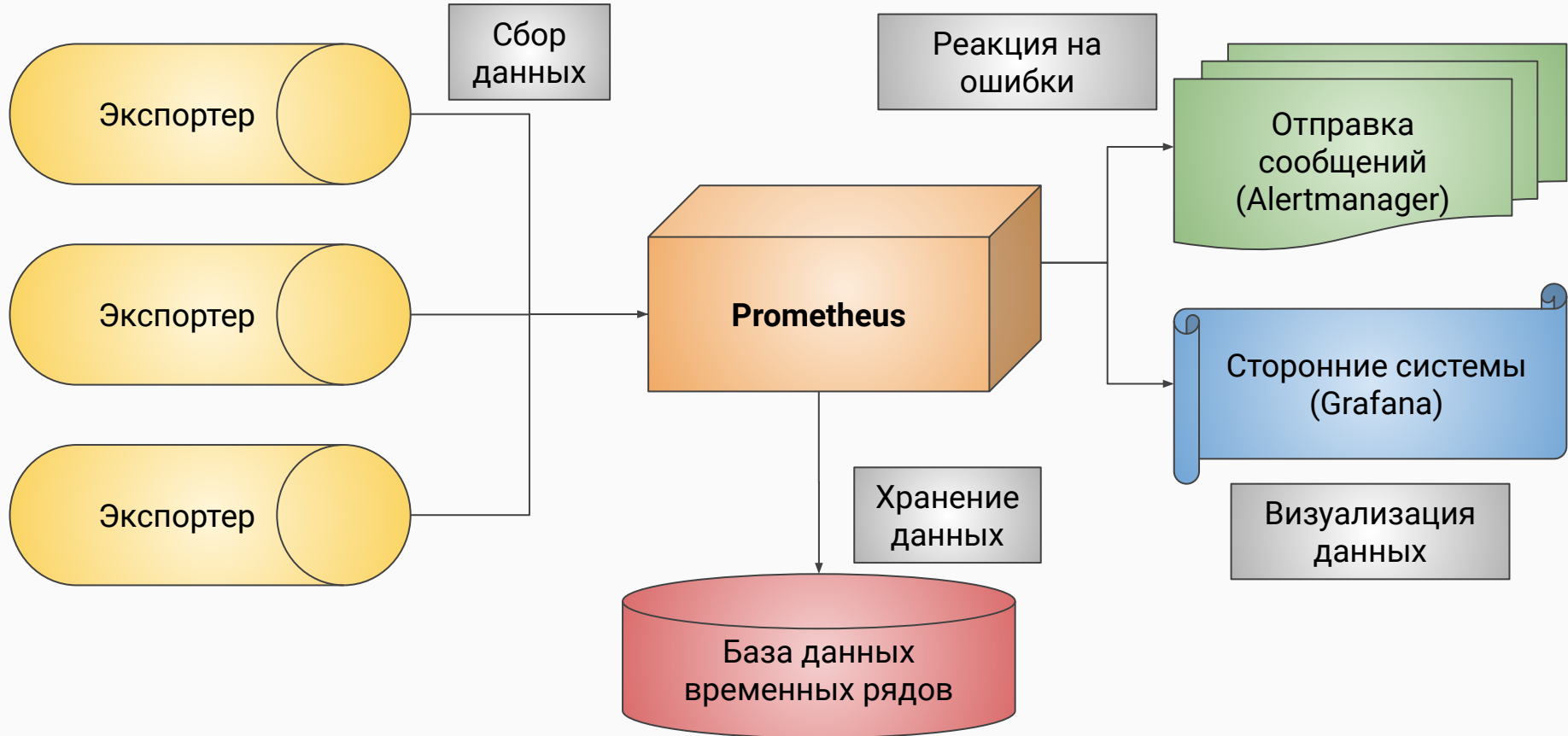
Введение в Prometheus

- Автоматизация контроля качества и развертывания является очень важным аспектом работы DevOps инженера. Однако не менее важным является контроль за состоянием приложения и его окружения (операционной системы) уже после того, как это приложение стало доступным конечному пользователю. Очевидно, что могут произойти всякие неожиданности - критически вырасти нагрузка, увеличиться или уменьшиться количество посетителей, само приложение может со временем начать работать очень странно или вообще выйти из строя.
- Для отслеживания состояния приложения используются специальные системы мониторинга. Они периодически собирают информацию (метрики) о работе важнейших узлов системы, хранят её у себя во внутренней базе данных, позволяют использовать информацию для составления удобных графиков и предоставляют возможности для интеграции с другими программами. Одним из самых популярных таких решений на данный момент является программа Prometheus. Она была создана в 2012 году на языке программирования Go, является бесплатным программным обеспечением, поддерживает все упомянутые выше действия, а также многие другие - например, информирование администратора о критических ситуациях и свой специальный язык запросов для получения информации и формирования графиков.

Упрощенная архитектура Prometheus

- Внутри Prometheus можно условно выделить несколько частей. Во-первых, это сам Prometheus, который является веб-приложением (т.е. его интерфейс доступен через браузер). Обычно используется порт 9090.
- Основная задача Prometheus - сбор информации из внешних источников. В конфигурационном файле прописываются соответствующие ссылки - именно по этим ссылкам с определенным интервалом Prometheus отправляет запросы. По ссылке должен быть доступна информация в определенном формате (о нем мы поговорим позже). Отображение информации можно настроить вручную, но для популярных технологий уже есть готовые решения, называемые экспортерами.
- Сами данные Prometheus хранит в специальной базе данных временных рядов, которая оптимизирована для работы с интервальными значениями.
- Дополнительной необязательной опцией является использование менеджера сообщений (**Alertmanager**), который Prometheus сможет задействовать для связи с администратором (если приложение не будет доступно по ссылке).
- Также Prometheus интегрируется со сторонними программами - самой известной из них является Grafana, предоставляющая интерфейс для работы с графиками.

Упрощенная архитектура Prometheus (визуализация)



Виды метрик

- Как уже говорилось, Prometheus периодически собирает информацию, которую собирают экспортеры - правильнее называть такую информацию метриками. Всего существует четыре вида метрик:
 - **Counter (счетчик)** - показывает некое возрастающее значение, например, количество секунд с момента старта приложения, общее количество полученных запросов и т.д.
 - **Gauge (датчик)** - отображает произвольно изменяющееся значение, например, потребление памяти каким-либо процессом (может уменьшаться или увеличиваться) или загрузка ядер процессора.
 - **Histogram (гистограмма)** - сразу несколько возрастающих значений с разделением на временные интервалы. Например, количество обработанных задач за 1 секунду, 15 секунд, 20 секунд и т.д.
 - **Summary (сводка)** - метрика, похожая на гистограмму, но предоставляет не только сырые данные, но и статистические обобщения - квантили (доля от общего числа, составляющая величины с размером ниже заданного значения).

Виды метрик (пример для счетчиков и датчиков)

```
# HELP http_requests_total Total number of HTTP requests.  
# TYPE http_requests_total counter  
http_requests_total{method="GET", status="200"} 100  
http_requests_total{method="POST", status="200"} 50
```

метрика-счетчик - общее количество запросов к приложению

```
# HELP memory_usage_bytes Current memory usage in bytes.  
# TYPE memory_usage_bytes gauge  
memory_usage_bytes{app="web-server"} 536870912
```

метрика-датчик - используемая оперативная память в момент обращения

Виды метрик (пример для гистограмм и сводок)

```
# HELP http_request_duration_seconds Histogram of HTTP request durations.
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.1"} 20
http_request_duration_seconds_bucket{le="0.5"} 50
http_request_duration_seconds_bucket{le="1.0"} 100
http_request_duration_seconds_sum 120
http_request_duration_seconds_count 100
```

метрика-гистограмма - количество запросов, отработавших за время в интервале [0, 0.1] секунды, [0, 0.5] секунд, [0, 1] секунды

```
# HELP http_request_duration_seconds Summary of HTTP request durations.
# TYPE http_request_duration_seconds summary
http_request_duration_seconds{quantile="0.5"} 0.6
http_request_duration_seconds{quantile="0.9"} 1.2
http_request_duration_seconds_sum 120
http_request_duration_seconds_count 100
```

метрика-обобщение - часть запросов (0.5), отработавших за время в интервале [0, 0.5] секунд и другая часть (0.9) - в интервале [0, 1.2] секунд

Структура метрик

- Из примеров на предыдущих страницах мы видим, что все метрики состоят из двух основных частей - названия метрики (например, **http_requests_total**) и ее значения (**100**). Также название метрики может быть дополнено метками, которые предоставляют вспомогательную информацию (например, у **http_requests_total**) сразу две метки - метод запроса и статус ответа - **{method="GET", status="200"}**) - эти метки могут использоваться для построения более сложных графиков.
- Для того, чтобы Prometheus при сборе данных понимал начало и конец метрик, то непосредственно перед метрикой располагается структура, которая начинается с **"# TYPE"** - далее идет название метрики (**http_requests_total**), а затем указание ее типа (**counter**).
- Также метрика обычно дополняется комментарием (объяснение того, что эта метрика вообще отображает) - для этого используется структура, который начинается с **"# HELP"**, затем указывается название метрики (**http_requests_total**), а в самом конце - непосредственно комментарий.

Установка Prometheus

- Установка Prometheus может быть произведена разными способами. Можно скачать установочный файл с официального сайта для нашей операционной системы - <https://prometheus.io/download/>, а можно использовать Docker образ, который доступен на DockerHub - <https://hub.docker.com/r/prom/prometheus/>. Для простоты мы воспользуемся Docker образом.
- Перед применением образа создадим специальную сеть - с ней в будущем будет удобно подключаться к разным узлам. Назовем ее **prometheus-network**:

```
docker network create --driver=bridge --subnet=172.20.0.0/24  
--gateway=172.20.0.1 prometheus-network
```

- Теперь установим Prometheus с использованием порта 9090 и подключим его к созданной сети (выдадим также статический IP - 172.20.0.11):

```
docker run -p 9090:9090 -d --net=prometheus-network  
--ip=172.20.0.11 --name=prometheus prom/prometheus
```

Конфигурационные настройки (обзор)

- После запуска Prometheus мы должны взглянуть на конфигурационный файл **/etc/prometheus/prometheus.yaml** - для этого надо зайти внутрь контейнера:

```
docker exec --it prometheus sh
```

- Конфигурация состоит из 4-х блоков. Во-первых, это глобальные настройки (свойство **global**), в которые входят два внутренних свойства - **scrape_interval** (периодичность сбора данных) и **evaluation_interval** (периодичность, с которой запускаются особые правила, реагирующие на состояния данных).
- Второй главный блок (**alerting**) связан с оповещением администратора в случае некой чрезвычайной ситуации - о нем мы подробно поговорим позже.
- В третьем главном блоке (**rule_files**) задаются пути к файлам, где можно вписать специальные правила для отправки сообщений администратору - этот блок также будет рассмотрен немного позже.
- Блок **scrape_configs** содержит массив ссылок хостов, данные которых собирает Prometheus. Внутреннее свойство **job_name** - название хоста, **static_configs** - адреса хоста, **metrics_path** - путь на хосте, где расположены метрики.

Конфигурационные настройки (визуализация)

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: "prometheus"

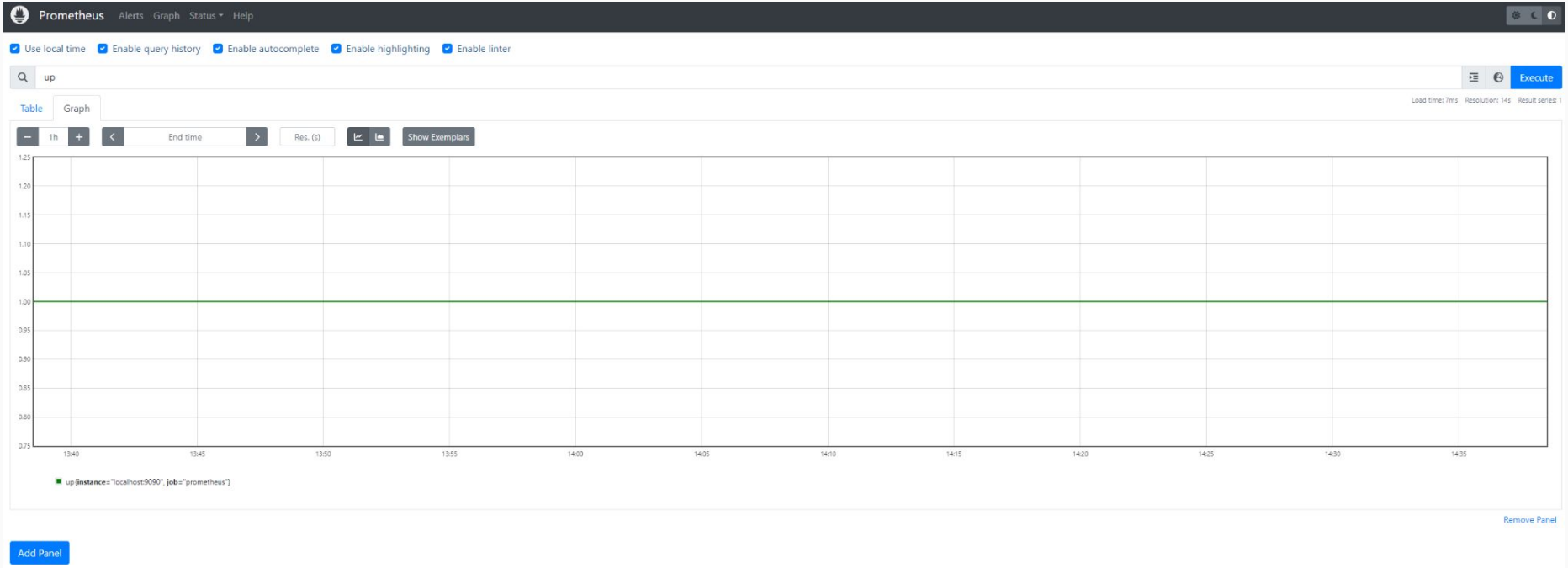
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]
```

Внутренний интерфейс (введение)

- Для непосредственного взаимодействия с Prometheus мы должны открыть браузер и перейти по адресу <http://localhost:9090>. Программа не имеет никаких ограничений (логина или пароля), поэтому мы сразу окажемся внутри системы.
- Внутри Prometheus нам доступны четыре больших раздела. **Alerts** (Оповещения) - предназначен для отображения информации, которая связана с оповещениями администратора в случае некой чрезвычайной ситуации.
- **Graph** (Графики) - в этом разделе можно делать запросы, чтобы получить статистическую информацию об отслеживаемых хостах. Информация может быть отражена в виде таблицы (**Table**) или графика (**Graph**). Таблица покажет для каждого параметра либо его последнее значение, либо значение на момент установленного времени. График покажет множество значений в контексте временных интервалов.
- Раздел **Status** (Статусы) предоставляет информацию о состоянии системы, конфигурациях и подключенных хостах.
- Раздел **Help** (Помощь) является простой ссылкой на сайт официальной документации Prometheus.

Внутренний интерфейс (пример)



Настройка времени сохранения данных

- Мы познакомились с базовыми разделами Prometheus и даже увидели файл настроек. Однако еще была освещена тема настроек, которые могут быть заданы только при создании конвейера. Самой важной такой настройкой является время хранения данных, которая по умолчанию равняется 15 дням. Можно понять, что мы можем оказаться в неприятной ситуации, если захотим получить данные, например, за целый год.
- Для того, чтобы задать эту настройку, нам надо удалить старый контейнер и создать новый, но уже с перечислением необходимой нам настройки (поставим срок хранения в 730 дней):

```
# удаление старого контейнера
```

```
docker rm -f prometheus
```

```
# удаление нового контейнера с перечислением настроек
```

```
docker run -p 9090:9090 -d --net=prometheus-network
```

```
--ip=172.20.0.11 --name=prometheus prom/prometheus
```

```
--config.file=/etc/prometheus/prometheus.yml
```

```
--storage.tsdb.path=/prometheus
```

```
--storage.tsdb.retention.time=730d
```


Установка экспортеров

Установка Node exporter

- Сразу же после установки Prometheus начинает собирать информацию только об одном приложении - самом себе. Это можно увидеть в конфигурационном файле, где внутри свойства **scrape_configs** находится элемент с названием **prometheus**, ссылающийся на **localhost:9090** (т.е. на тот адрес, где установлен Prometheus).
- Очевидно, что данный элемент вставлен только для примера - в реальном мире мы захотим следить за более серьезными вещами. Как уже было сказано, данные для Prometheus предоставляют специальные программы - экспортеры. Их существует великое множество, но для начального примера мы возьмем программу под названием Node exporter, которая предоставляет метрики для систем на ядре Linux - загрузку процессора, свободную память и т.д.
- Установка Node exporter на операционную систему подробно описана в руководстве Prometheus - <https://prometheus.io/docs/guides/node-exporter/>, но мы сделаем нашу жизнь еще проще - возьмем Docker образ, создадим из него контейнер и подключим контейнер (IP - 172.20.0.12) к одной сети с Prometheus:

```
docker run -d --name=node-exporter --net=prometheus-network  
--ip=172.20.0.12 -p 9100:9100  
quay.io/prometheus/node-exporter:latest
```

Подключение **Node exporter** к Prometheus

- То, что мы установили Node exporter, еще не означает, что Prometheus его автоматически увидит. Чтобы все же подключить Node exporter к Prometheus, нам необходимо перейти в конфигурационный файл Prometheus и внутри свойства-массива **scrape_configs** вставить следующий элемент:

```
- job_name: "linux"
  static_configs:
    - targets: ["172.20.0.12:9100"]
```

- Теперь мы должны перезагрузить конфигурацию Prometheus, чтобы изменения вступили в силу. Для этого внутри Prometheus контейнера нам следует выполнить следующую команду (приказ всем процессам, связанным с Prometheus, остановиться и запуститься вновь):

```
killall -HUP prometheus
```

Установка **django-prometheus**

- Далее мы установим экспортер **django-prometheus** - из названия понятно, что он предоставит данные для приложения, написанного на фреймворке Django. В данном случае это пакет (<https://github.com/korfuri/django-prometheus>), который можно установить при помощи менеджера зависимостей PIP.

```
pip install django-prometheus
```

- Затем в файле **settings.py** мы должны прописать следующие конструкции:

```
INSTALLED_APPS = [  
    ...  
    'django_prometheus',  
    ...  
]  
  
MIDDLEWARE = [  
    'django_prometheus.middleware.PrometheusBeforeMiddleware',  
    ...  
    'django_prometheus.middleware.PrometheusAfterMiddleware'  
]
```

- После этого в файле **urls.py** надо прописать ссылки для отображения метрик:

```
urlpatterns = [  
    ...  
    path('', include('django_prometheus.urls')),  
]
```

Подключение **django-prometheus** к Prometheus

- После того, как наше Django приложение уже написано, напишем для него специальный Dockerfile (он будет предоставлен в процессе обучения), а затем создадим из него Docker образ **django-prometheus**:

```
docker build -t django-prometheus .
```

- Следующий шаг - создание контейнера из образа в той же сети, где работает Prometheus (IP - 172.20.0.13):

```
docker run -d --name=django-prometheus  
--net=prometheus-network --ip=172.20.0.13 -p 8888:80  
django-prometheus
```

- В конце добавляем новый адрес в конфигурацию и перезагружаем Prometheus:

```
- job_name: "django"  
  static_configs:  
    - targets: ["172.20.0.13:80"]
```

```
killall -HUP prometheus
```

Отображение статуса отслеживаемым приложениям

- После того, как мы запустим все приложения, следует перейти в Prometheus во вкладку **Status -> Targets** - если все сделано правильно, то будут отображены все три отслеживаемых системы - сам Prometheus (**prometheus**), операционная система на ядре Linux (**linux**) и сайт на Django (**django**):

The screenshot shows the Prometheus web interface. At the top is a navigation bar with 'Prometheus', 'Alerts', 'Graph', 'Status', and 'Help'. The 'Targets' section is active. Below the navigation bar, there's a filter bar with 'All scrape pools', 'Unhealthy', 'Collapse All', and a search input. To the right are status indicators: 'Unknown', 'Unhealthy', and 'Healthy'. The main content area displays three target groups: 'django (1/1 up)', 'linux (1/1 up)', and 'prometheus (1/1 up)'. Each group has a 'show logs' button and a table of targets.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.20.0.13/metrics	UP	instance="172.20.0.13:80" job="django"	3.715s ago	7.514ms	
http://172.20.0.12:9100/metrics	UP	instance="172.20.0.12:9100" job="linux"	12.831s ago	15.302ms	
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	8.850s ago	9.883ms	

Работа с данными Prometheus

Введение в язык PromQL

- Чтобы получать данные в нужном для нас формате, надо перейти в приложение Prometheus и открыть вкладку **Graph**. В открывшемся окне у нас будет поле для команд (рядом с кнопкой **Execute**), в результате исполнения которых и будут отображаться либо таблицы, либо графики. Интересный момент - можно добавить несколько полей - в результате выполнения их команд на странице появится несколько графиков.
- Для написания команд в Prometheus используется язык PromQL. Он не особо сложен - в основном мы указываем название метрики и, если понадобится, одну из меток, которая связана с метрикой (если нас интересует только та информация, которая связана с этой меткой). Например, если мы хотим получить общее количество запросов для Django, то команда выглядит так:

```
django_http_requests_total_by_method_total
```

<code>django_http_requests_total_by_method_total{instance="172.20.0.13:80", job="django", method="GET"}</code>	20
<code>django_http_requests_total_by_method_total{instance="172.20.0.13:80", job="django", method="POST"}</code>	5

Выборка данных с применением базовой фильтрации

- Приведем пример запроса в том случае, если нам необходимо получить данные по конкретной метке. У нас есть экспортер **Node exporter**, предоставляющий данные по операционной системе. Возьмем метрику **node_cpu_seconds_total**, которая посвящена количеству секунд, которые ядра процессора потратили на работу в разных режимах и получим информацию только по состоянию **idle** (т.е. только те моменты, когда ядра не работали):

```
node_cpu_seconds_total{mode="idle"}
```

node_cpu_seconds_total{cpu="0", instance="172.20.0.12:9100", job="linux", mode="idle"}	1812.07
node_cpu_seconds_total{cpu="1", instance="172.20.0.12:9100", job="linux", mode="idle"}	1816.01

- Важно понимать, что мы не ограничены указанием только одной метки - через запятую можно указывать все необходимые нам метки, например, номер ядра:

```
node_cpu_seconds_total{mode="idle", cpu="0"}
```

node_cpu_seconds_total{cpu="0", instance="172.20.0.12:9100", job="linux", mode="idle"}	1812.07
--	---------

Продвинутые способы фильтрации

- При фильтрации по меткам мы можем применять регулярные выражения, но для этого следует немного изменить синтаксис оператора сравнения - вместо `=` следует применять `=~`. Применим наши новые знания для того, чтобы получить данные о времени работы первого ядра процессора как в режиме исполнения задач пользователя (**user**), так и в режиме исполнения задач системы (**system**):

```
node_cpu_seconds_total{mode=~"user|system", cpu="0"}
```

<code>node_cpu_seconds_total{cpu="0", instance="172.20.0.12:9100", job="linux", mode="user"}</code>	1.475
<code>node_cpu_seconds_total{cpu="0", instance="172.20.0.12:9100", job="linux", mode="system"}</code>	2.505

- Также при фильтрации нам доступен оператор отрицания `!=`, инвертирующий логику поиска - поиск происходит по тем значениям, которые отличаются от представленного значения. Найдем время работы первого ядра процессора, которое занимались любой деятельностью отлично от `idle`:

```
node_cpu_seconds_total{mode!="idle", cpu="0"}
```

Применение математических операций

- Каждое значение, возвращаемое PromQL запросом, может быть включено в состав математического выражения. Поддерживаются следующие операции: + – сложение, - – вычитание, * – умножение, / – деление, % – деление по модулю, ^ – возведение в степень. Далее приведем пример, в котором участвуют две метрики - **process_open_fds** (количество открытых файловых дескрипторов) и **process_max_fds** (максимально возможное число открытых файловых дескрипторов) - посчитаем процент использованных дескрипторов относительно максимально разрешенного значения для контейнера **linux**:

```
process_open_fds{job="linux"} / process_max_fds{job="linux"} * 100
```

- Математические операции могут быть осуществлены не только между одиночными значениями, но и между наборами значений. Пары наборов значений будут подобраны с учетом полного совпадения меток векторов у обоих операндов. В подтверждение этих слов посчитаем процент использованных дескрипторов для каждого контейнера (т.е. вообще не будем указывать название контейнера в метриках):

```
process_open_fds / process_max_fds * 100
```

Применение операций сравнений

- Помимо математических вычислений, Prometheus поддерживает следующие операции сравнения: `==` – равно, `!=` – не равно, `>` – больше, `<` – меньше, `>=` – больше или равно, `<=` – меньше или равно. Важно знать, что операции сравнения не используются для получения каких-либо конкретных логических значений, а только для фильтрации - это значит, что в финальной выборке останутся только те метрики, значение которое подпадает под истинность той или иной операции.
- Допустим, мы хотим получить информацию (на графике) о том, когда количество запросов определенного типа (с методом GET, POST и т.д.) превысило показатель 1000. Для этого метрику **`django_http_requests_total_by_method_total`** необходимо сравнить с числом 1000 и переключиться на тип отображения **Graph** - отображение графика начнется именно с того времени, когда число запросов сравнялось с 1000:

```
django_http_requests_total_by_method_total >= 1000
```

Получение векторов диапазонов

- До этого момента мы получали структуры, которые называются **мгновенными векторами**. Это означает, что мы получали список неких атрибутов (по сути, это были комбинации меток), и каждый элемент списка имел одно значение. Однако Prometheus позволяет получать т.н. **вектора диапазонов** - мы указываем тот период времени, за который мы хотим получить данные, а в ответ нам вернется тоже список, но в качестве значений будет возвращено несколько значений - т.е. те значения, которые входят в диапазон. Важно знать, что вектор диапазонов не может быть отображен на графике - в этом случае надо будет использовать дополнительные конструкции, о которых мы поговорим позже.
- Для пример получим вектор диапазонов, который отражает нарастание количества запросов в Django приложении с диапазоном 30 секунд. В качестве селектора диапазонов применяются квадратные скобки, внутри которых указывается время (**1d** - 1 день, **5h** - 5 часов, **15m** - 15 минут, **30s** - 30 секунд):

```
django_http_requests_total_by_method_total[30s]
```

Агрегатные функции

- Иногда нам могут понадобиться некий обобщенный результат - среднее значение для всех результатов списка, минимальный и максимальный элемент в списке, а также количество элементов самого списка. Обобщенно этот процесс называется агрегацией - превращением нескольких значений в одно. Для осуществления агрегации используются специальные конструкции - функции, которые оборачивают те выражения, которые мы использовали до сих пор.
- Для примера получим средний размер диска в отслеживаемой операционной системе на ядре Linux - сама метрика называется **node_filesystem_size_bytes**, а функция, которая отслеживает среднее значение - **avg**:

```
avg(node_filesystem_size_bytes)
```

- Далее применим функцию **sum**, чтобы получить свободное место всех дисков:

```
sum(node_filesystem_avail_bytes)
```

- Другие важные функции, которые могут понадобиться для агрегации - **min** (минимальное значение), **max** (максимальное значение) и **count** (количество элементов в выборке).

Осуществление группировки

- У простой агрегации есть один существенный недостаток - мы не можем выделить отдельные группы внутри списка и посчитать некое обобщенное значение внутри этих групп. Например, нам может потребоваться посчитать общее время, которое затратило каждое ядро процессора на решение задач. Однако при осуществлении простой агрегации мы получим время, затраченное всеми ядрами! К счастью, Prometheus поддерживает выделение групп (мы сами можем указать, каких именно) и просчитывание обобщенных значений для этих групп - для этого применяется ключевое слово **by**, после которого в скобках указываются те метки, уникальные значения которых должны входить в группу.
- Приведем пример для времени, затраченном на работу ядрами процессора - за это отвечает метрика **node_cpu_seconds_total**, а за номер ядра - метка **cpu**:

```
sum by (cpu) (node_cpu_seconds_total)
```

- Также при помощи ключевого слова **without** можно применять исключающую группировку - т.е. Группировка будет происходить по всем параметрам за исключением указанных:

```
sum without (cpu) (node_cpu_seconds_total)
```

Функции для векторов диапазонов (введение)

- Нам всегда следует помнить об одном исключении - все перечисленные до этого функции умеют работать только с мгновенными векторами, но они не могут быть применены к векторам диапазонов. Кроме того, мы не должны забывать, что вектора диапазонов создают еще одну проблему - они не могут быть отображены с помощью графиков. Однако есть решение проблемы - существуют функции, предназначенные для работы с исключительно векторами диапазонов, более того, результаты работы этих функций можно отображать на графике.
- Прежде всего функции для векторов диапазонов дополняют стандартные функции. Для **max** это **max_over_time**, **min** - **min_over_time**, **sum** - **sum_over_time**, **count** - **count_over_time** и **avg** - **avg_over_time**. Остальные функции такого типа занимаются тем, что считают производную для доступных данных, либо вычисляют разницу между первым и последним результатом в диапазоне. Можно сделать вывод, что эти конструкции хорошо подходят в том случае, если нам надо получить информацию о всплесках активности, а также быстром приросте или убывании некоего показателя.

Функции для векторов диапазонов (примеры)

- Для первой мы применим функцию **rate**, которая считает производную (как быстро увеличивалось значение в каком либо диапазоне) для метрик типа **counter**. Посчитаем производную для количества запросов к нашему Django приложению с шагом в 5 минут - можно вручную сделать много запросов и увидеть, как изменения отобразятся на графике (вкладка **Graph**):

```
rate(django_http_requests_total_by_method_total[5m])
```

- Далее рассмотрим функцию **deriv**, которая считает производную для метрик типа **gauge**. В качестве примера возьмем метрику **node_filesystem_avail_bytes**, которая отображает свободный размер диска. Шаг установим в 2 часа. Чтобы увидеть изменения, внутри контейнера с операционной системой можно создавать файлы со случайным содержимым с помощью следующей команды:

```
dd if=/dev/urandom of=sample.txt bs=1G count=1 iflag=fullblock
```

Сама команда **deriv** для получения соответствующей производной:

```
deriv(node_filesystem_avail_bytes[2h])
```

Использование менеджера сообщений (Alertmanager)

- Просмотр и анализ статистических данных - очень важная вещь в мониторинге приложений. Однако не менее важным аспектом мониторинга является реакция на чрезвычайные ситуации - например, сайт может перестать работать, на диске закончится место, а оперативная память может быть заполнена на сто процентов. Prometheus предоставляет возможность отслеживать такие ситуации и своевременно отправлять своему администратору сообщения - по электронной почте, Telegram, Slack и т.д.
- Для отправки сообщений следует установить соответствующую программу под названием Alertmanager. Она доступна на DockerHub, поэтому установим ее из готового образа **prom/alertmanager**. Не забудем подключить Alertmanager к одной сети с Prometheus (**prometheus-network**) и установить явный IP адрес контейнера (например, **172.20.0.14**):

```
docker run -d --name=alertmanager-prometheus  
--net=prometheus-network --ip=172.20.0.14 prom/alertmanager
```

Настройка Alertmanager

- Сначала нам надо настроить Alertmanager таким образом, чтобы он смог отправлять сообщения по эл. почте (выберем этот вариант оповещения). Для этого откроем контейнер, в котором находится наш менеджер оповещений и перейдем в директорию **/etc/alertmanager**.
- Очистим файл **alertmanager.yml** и начнем заполнять его нашими значениями. Внутри свойства верхнего уровня **route** мы устанавливаем то, как будет работать наша система оповещений. Свойство **receiver** определяет каким способом будет отправлено сообщение (список таких способов - **receivers**). Свойство **group_wait** устанавливает время, которое система будет ждать, чтобы послания с одинаковыми метками были объединены в одну группу, когда эта группа только появляется (чтобы все эти послания были отправлены одним сообщением). Свойство **group_interval** устанавливает время, которое должно пройти перед тем, как сообщения группы будут анализироваться после последней отправки. Еще одно свойство, **repeat_interval**, устанавливает то время, которое должно пройти перед тем, как сообщение может быть отправлено вновь.
- Свойство верхнего уровня **receivers** посвящено способам отправки сообщений - мы видим, что был создан один способ **email** (название может быть любым), который шлет сообщения по эл. почте (внутреннее свойство **email_configs**).

Настройка Alertmanager (`alertmanager.yml`)

```
route:
  receiver: 'email'
  group_wait: 15s
  group_interval: 15s
  repeat_interval: 1m

receivers:
  - name: 'email'
    email_configs:
      - smarthost: 'YOUR_SMTP_URL'
        auth_username: 'YOUR_SMTP_USERNAME'
        auth_password: 'YOUR_SMTP_PASSWORD'
        from: 'sender@gmail.com'
        to: 'receiver@gmail.com'
        headers:
          subject: 'Prometheus Email Alert'
```

Создание конфигурационного файла для правил

- Теперь мы должны вернуться в контейнер, где находится сам Prometheus и определить те случаи, когда мы должны отправлять сообщения. Для этого в директории **/etc/prometheus** мы должны создать специальный файл (например, **alert.yml**), где будут описаны соответствующие условия.
- Условия принято группировать - поэтому вначале идет свойство верхнего уровня **groups**, внутри которого должен быть массив групп. В каждой группе нам надо указать название (свойство **name**) и массив правил, согласно которым отправляются сообщения (свойство-массив **rules**).
- В каждом правиле должно быть наименование (свойство **alert**), выражение, написанное на языке PromQL (свойство **expr**) - если выражение будет истинным, то данное правило начнет обрабатывать. Затем идет свойство **for** - здесь надо указать то время, которое выражение из предыдущего свойства должно быть истинным. После этого идет свойство-массив **labels** - там можно указать любые метки, которые используются для группировки посланий в одно сообщение. В самом конце расположено свойство-объект **annotations**. Внутри есть два дополнительных свойства - **summary** и **description**. Их значения и попадут в финальное сообщение.

Пример конфигурационного файла для правил (alert.yml)

```
groups:
- name: "Critical Errors"
  rules:
  - alert: "Django Site is Down"
    expr: up{job="django"} == 0
    for: 15s
    labels:
      severity: critical
    annotations:
      summary: "Instance [{{ $labels.instance }}] down"
      description: "[{{ $labels.instance }}] of job {{ $labels.job }} has been down for more than 15 seconds."
```

Подключение Alertmanager и конфигурационного файла правил к Prometheus

- Теперь нам осталось собрать все вместе. Вначале подключаем Alertmanager к самому Prometheus. Для этого внутри контейнера, где работает Prometheus, надо перейти в файл **/etc/prometheus/prometheus.yml** и внутри следующего свойства **alerting -> alertmanagers -> static_configs -> targets** добавим ссылку на IP адрес и порт того контейнера, где работает Alertmanager.

```
- targets: ["172.20.0.14:9093"]
```

- Далее переходим к свойству **rule_files** и добавляем в него ссылку на файл с правилами инициализации сообщений (у нас он называется **alert.yml**):

```
rule_files:  
  - "alert.yml"
```

- После всех этих действий надо перезагрузить конфигурации Prometheus:

```
killall -HUP prometheus
```

- Теперь мы можем выключить в Docker наше Django приложение - через некоторое время нам придет сообщение по указанной нами электронной почте.

Подключение Alertmanager к Prometheus (пример)

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets: ["172.20.0.14:9093"]

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  - "alert.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]
  - job_name: "linux"
    static_configs:
      - targets: ["172.20.0.12:9100"]
  - job_name: "django"
    static_configs:
      - targets: ["172.20.0.13:80"]
```

Подключение Grafana

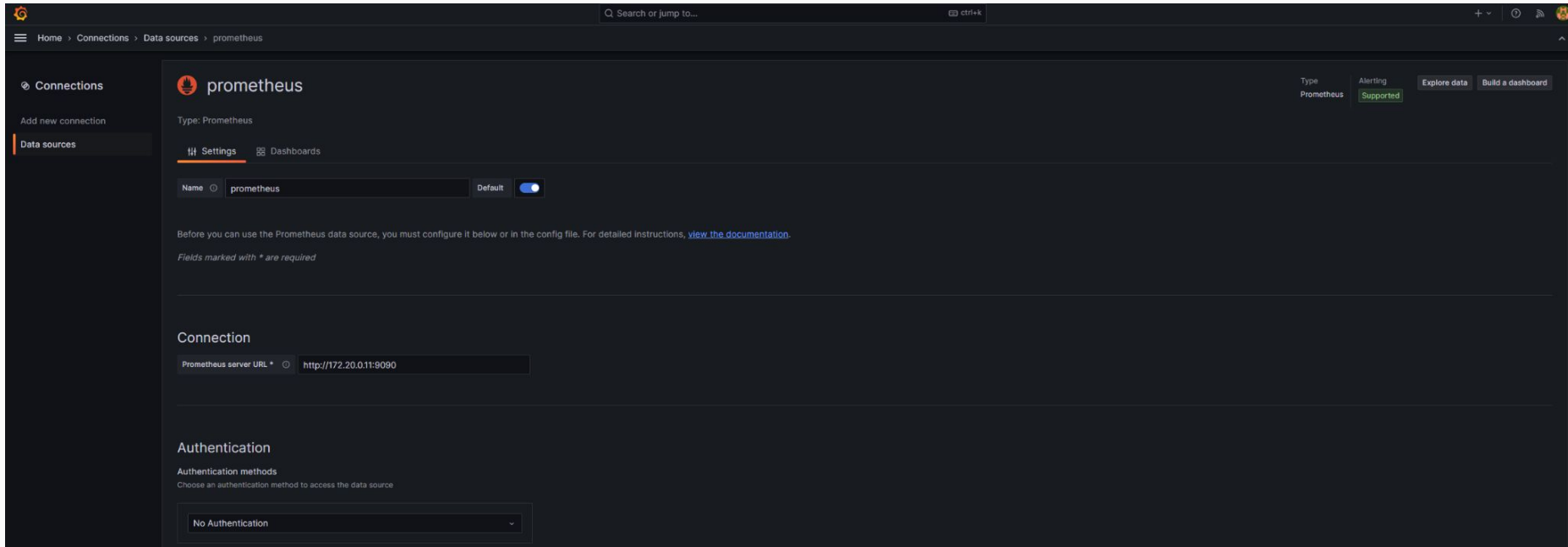
- Prometheus является отличной системой мониторинга, однако данная программа обладает крайне ограниченными средствами визуализации - у нас есть возможность отобразить данные только в виде таблиц и простейших графиков. Безусловно, анализ информации можно производить и в таком виде, однако стоит признать, что хочется немного больше комфорта при отрисовке графиков и диаграмм.
- Одной из самых лучших систем для визуализации данных на начало 2024 года является Grafana. Она умеет работать с данными таких программ и сайтов как MySQL, PostgreSQL, Jira, Github, Gitlab и, конечно же, Prometheus. Grafana доступна на DockerHub, поэтому воспользуемся готовым образом grafana/grafana, создадим новый контейнер, подключим его к нашей сети prometheus-network с IP 172.20.0.15 и пробросим внутрь контейнера 3000 порт:

```
docker run -d --name=grafana --net=prometheus-network  
--ip=172.20.0.15 -p 3000:3000 grafana/grafana
```

Настройка Grafana

- Для первоначальной настройки Grafana мы должны открыть браузер и перейти по адресу <http://localhost:3000>. Откроется окошко, в котором у нас потребуют ввести логин и пароль - в обоих случаях вводим слово **admin**. На следующей странице у нас запросят новый пароль и его подтверждения - для простоты воспользуемся словом **secret**.
- Теперь мы должны подключить к Grafana наш Prometheus. Для этого открываем главное меню (кнопка в верхнем левом углу) и выбираем пункт **Data Sources**. В новом окне нажимает большую синюю кнопку **Add data source** и выбираем опцию **Prometheus**.
- На новой странице мы увидим много настроек, но нам нужна только одна - **Prometheus server URL** (находится внутри блока **Connection**). Внутри этой настройки необходимо вписать IP адрес и порт нашего Prometheus сервера - <http://172.20.0.11:9090>. После этого нам надо пролистать страницу до самого низа и нажать на синюю кнопку **Save & test**. Если все сделано правильно, появится надпись **Successfully queried the Prometheus API**.

Настройка Grafana (пример подключение к Prometheus)



The screenshot displays the Grafana web interface for configuring a Prometheus data source. The top navigation bar includes the Grafana logo, a search bar, and user controls. The left sidebar shows the 'Connections' menu with 'Data sources' selected. The main content area is titled 'prometheus' and shows the 'Settings' tab. The configuration includes a name field set to 'prometheus', a 'Default' toggle switch, and a 'Prometheus server URL' field set to 'http://172.20.0.11:9090'. The 'Authentication' section shows 'No Authentication' selected from a dropdown menu.

Home › Connections › Data sources › prometheus

Search or jump to... ctrl+k

Connections

Add new connection

Data sources

prometheus

Type: Prometheus

Alerting Supported Explore data Build a dashboard

Settings Dashboards

Name ⓘ prometheus Default ☒

Before you can use the Prometheus data source, you must configure it below or in the config file. For detailed instructions, [view the documentation](#).

Fields marked with * are required

Connection

Prometheus server URL * ⓘ http://172.20.0.11:9090

Authentication

Authentication methods

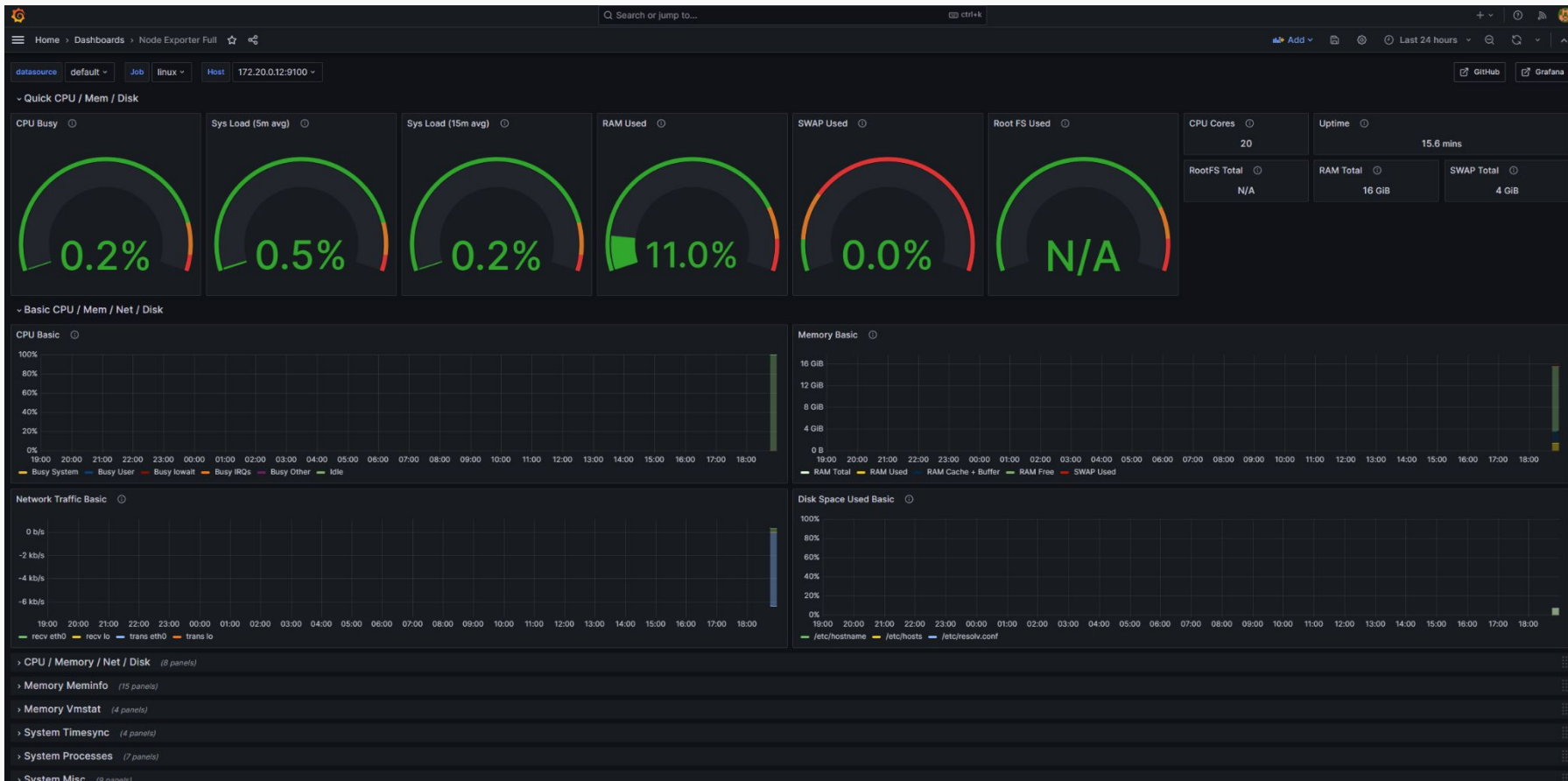
Choose an authentication method to access the data source

No Authentication

Создание приборных панелей (dashboard) для Grafana

- Создадим отдельные приборные панели (dashboards), которые будут отображать состояние приложений, отслеживаемых Prometheus - операционную систему Linux (**node exporter**) и сайт на Django (**django prometheus**). Самый правильный вариант - не придумывать велосипед, а воспользоваться готовым решением.
- Переходим в главное меню (кнопка вверху слева) и выбираем пункт **Data Sources**. В новом окне выбираем пункт Prometheus и нажимаем кнопку **Build a dashboard**, а затем кнопку **Import dashboard**. После этого переходим на сайт <https://grafana.com/grafana/dashboards/> и находим вариант **Node Exporter Full** (визуализация для Linux). Копируем ID этого варианта (**Copy ID to clipboard**) и вставляем его внутри поля внутри Grafana и нажимаем кнопку **Load** рядом с этим полем. Далее в поле Prometheus выбираем наше соединение с Prometheus и нажимаем на кнопку **Import**.
- Для добавления мониторинга Django мы должны повторить тот же самый путь, но на сайте <https://grafana.com/grafana/dashboards/> следует найти вариант **a Django Prometheus** и воспользоваться уже его идентификатором (ID).

Приборная доска для мониторинга операционных систем на ядре Linux



Приборная доска для мониторинга Django сайта

