

- [Основы](#)
  - [Архитектура Прометея.](#)
    - [Немного из зарубежных источников про архитектуру Прометея](#)
      - [What is Prometheus?](#)
      - [Prometheus Architecture](#)
      - [Some Terminologies](#)
      - [More about Metrics](#)
      - [How does it work?](#)
      - [How does it get the data from Targets?](#)
      - [Monitoring Personal Services](#)
      - [How is it different?](#)
      - [How to use it?](#)
      - [Default Configuration File](#)
      - [How does Alerting work?](#)
      - [Where is the metrics data stored?](#)
      - [How to get the data?](#)
      - [Running it Locally.](#)
- [Exposition](#)
  - [Общие сведения о процессе экспозиции](#)
  - [Node Exporter](#)
  - [Типы метрик](#)
    - [Counter](#)
    - [Gauge](#)
    - [Histogram](#)
    - [Summary](#)
  - [Установка Node Exporter](#)
    - [Ключи запуска Node Exporter](#)
    - [Добавление метрик из файла](#)
  - [Blackbox Exporter](#)
  - [Установка Blackbox Exporter](#)
  - [Настройка Blackbox Exporter](#)
    - [Общая структура конфигурационного файла.](#)
    - [ICMP протокол](#)
    - [DNS протокол](#)
    - [TCP протокол](#)
    - [HTTP протокол](#)
    - [Настройка со стороны Prometheus](#)
  - [Custom Exporter](#)
    - [Пример exporter на языке go](#)
  - [Application Library](#)
- [Prometheus](#)
  - [Основа](#)
    - [Установка Prometheus](#)
    - [Основной конфигурационный файл Prometheus](#)

- [Ключи запуска Prometheus](#)
- [Service Discovery](#)
  - [Настройка scraping с использованием static config](#)
  - [Настройка service discovery с использованием файлов](#)
  - [Настройка service discovery с использованием api openstack](#)
- [Labels](#)
  - [Модификаторы labels](#)
- [Push Gateway](#)
  - [Установка Pushgateway](#)
  - [Ключи запуска PushGateway](#)
  - [PushGateway API](#)
    - [Структура URL](#)
    - [Метод PUT](#)
    - [Метод POST](#)
    - [Метод DELETE](#)
  - [Отправка данных в PushGateway](#)
  - [UI PushGateway](#)
  - [Настройка Prometheus](#)
- [PromQL](#)
  - [Хранение данных](#)
  - [Типы данных](#)
  - [Выражения](#)
    - [Типы данных в PromQL](#)
    - [Операторы PromQL](#)
      - [Математические операторы:](#)
      - [Операторы сравнения:](#)
      - [Логические операторы:](#)
    - [Сопоставления векторов](#)
    - [Операторы агрегации:](#)
    - [Приоритет бинарных операторов](#)
    - [Математические функции](#)
    - [Функции времени и даты](#)
    - [Метки](#)
    - [Пропущенные серии](#)
    - [Сортировки](#)
    - [Агрегация во времени](#)
    - [Функции для Counter](#)
    - [Функции для Histograms](#)
    - [Функции для Gauges](#)
  - [Record Rules](#)
    - [Настройка Rules](#)
    - [Именование Rules](#)

- [Антипаттерны для использования Rules](#)
  - [Настройка Rules](#)
- [Alerting](#)
  - [Alerting Rules](#)
    - [Настройка Alert Rules](#)
    - [Добавление правил уведомлений](#)
  - [Alertmanager](#)
    - [Установка Alertmanager](#)
    - [Ключи запуска Alertmanager](#)
    - [Настройка Alertmanager](#)
      - [Общие сведения](#)
      - [Блок global](#)
      - [Блок route](#)
      - [Блок receivers](#)
      - [Блок inhibit rules](#)
      - [Блок templates](#)
      - [Настройка alertmanager - практика](#)
      - [Настройка Alertmanager HA](#)
- [Визуализация данных](#)
  - [Grafana](#)
    - [Установка Grafana](#)
    - [Настройка источника данных в Grafana](#)
    - [Настройка dashboard в Grafana](#)
    - [Импорт готовых dashboard для Grafana](#)
- [Advanced usage of Prometheus](#)
  - [High Availability.](#)
    - [Настройка Prometheus в режиме HA](#)
    - [Настройка HA Proxy](#)
  - [Federation](#)
    - [Настройка Prometheus Federation](#)
  - [Remote read/write](#)
    - [Настройка Remote read/write](#)
  - [Thanos](#)
  - [HTTP API](#)
    - [Общие сведения для запросов API](#)
    - [Запросы мгновенного вектора](#)
    - [Запрос вектора за период времени](#)
    - [Поиск временных рядов на основании labels](#)
    - [Получение списка меток](#)
    - [Запрос значений для label](#)
    - [API для получения конфигурации](#)
      - [Конфигурация сервера](#)
      - [Список ключей запуска](#)
      - [Список targets](#)

- [Rules и Alerts API](#)
  - [Alertmanagers](#)
  - [Alerts](#)
  - [Rules](#)
- [Prometheus в Kubernetes](#)
  - [Установка](#)
    - [Установка Prometheus в kubernetes](#)

## Основы

---

Для чего нужен курс?

- Систематизация знаний.
- Нюансы настройки Prometheus.
- Лучшие паттерны использования.

Прометей - OpenSource. Изначально разрабатывался в SoundCloud.

Сделан на основе Boardman от Google. Разработан примерно в 2011-2013 году.

Почему Прометей стал стандартом на сегодня?

- Изначально ориентирован на работу в динамических окружениях - Service Discovery, Labels
- HA/Scalability - отказоустойчивость и масштабируемость
- включает Time Series Database, что вывело на качественно новый уровень скорость обработки данных
- собственный язык запросов к БД PromQL, который разрабатывался именно для решения задач мониторинга

В Zabbix в последних версиях тоже добавили поддержку Prometheus а также тоже перешли на Time Series DB.

## Архитектура Прометея.

---

Архитектура состоит из 3 частей:

- Scraping
- Rules & Alerts
- Service Discovery.

**Источники данных:**

- экспортеры - либо готовые, либо самописные.
- клиентские библиотеки, встраиваемые в приложение.

**Скрейпинг** - механизм сбора метрик из источников данных в базу Прометея.

Источники данных можно задавать статически, но рекомендуется динамически определять таргеты для скрейпинга через Service Discovery.

**Rules & Alerts:**

Правила позволяют сохранять кастомные PromQL-запросы в виде отдельной метрики (примерно как хранимые процедуры в SQL).

Алерты - проверяются условия срабатывания алertsов и срабатывает вебхук. Далее через механизм Alertmanager производится роутинг и доставка алertsов в точку назначения - почта, мессенджер, СМС и т.д.

Для визуализации используется чаще всего Grafana.

# Немного из зарубежных источников про архитектуру Prometheus

1

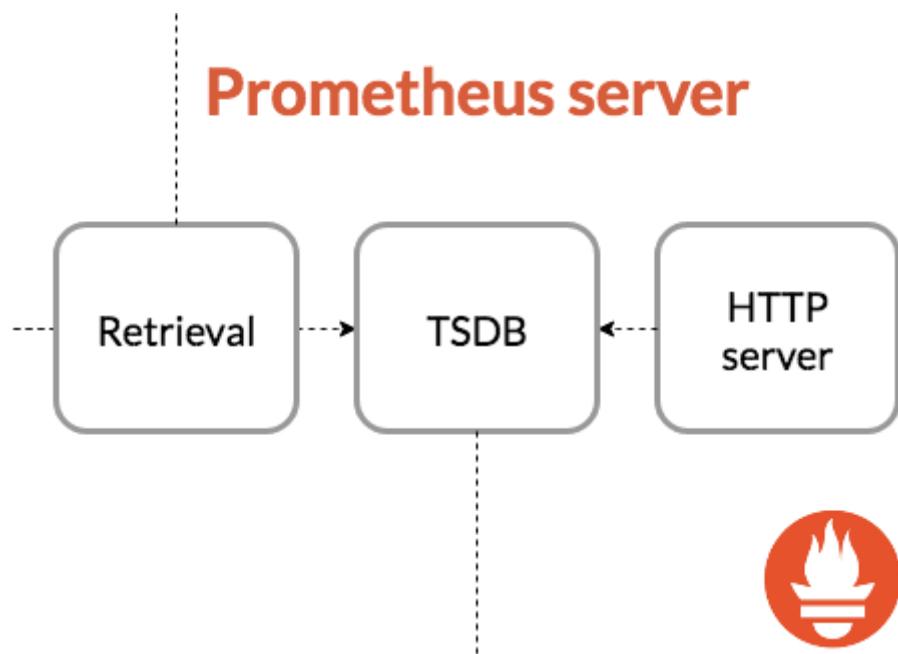
## What is Prometheus?

Prometheus is an open-source **monitoring & alerting tool**. It was originally built by SoundCloud and now it is 100% open-source as a Cloud Native Computing Foundation graduated project. It has become highly popular in monitoring container & microservice environments.

## Prometheus Architecture

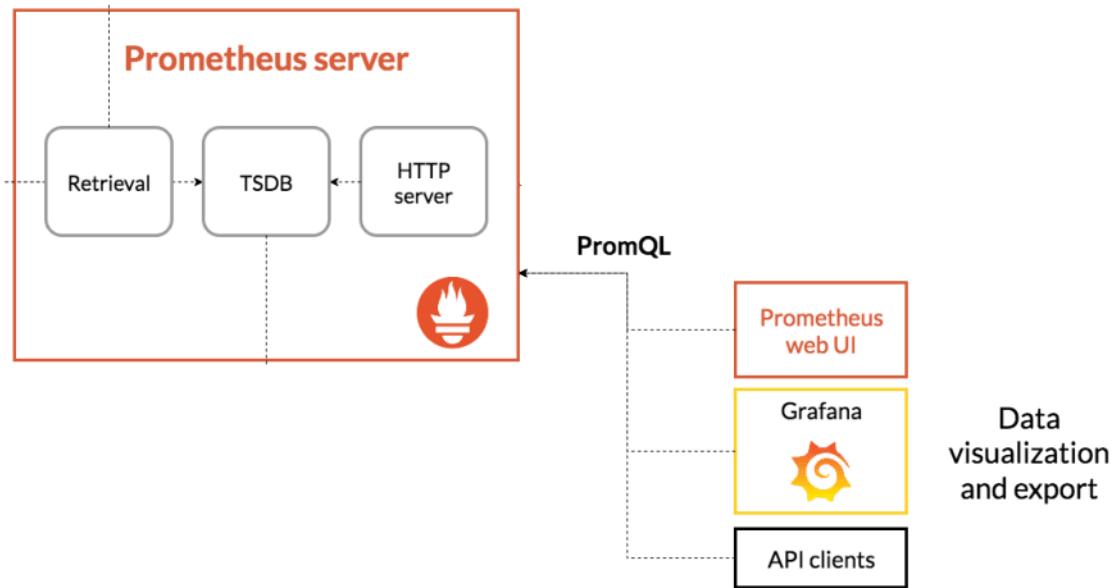
### Some Terminologies

- **Target** - It is what Prometheus monitors. It can be your applications, servers, etc.
- **Metric** - For our targets, we would like to monitor particular things. Like for example, if we have a server (target) we would want to monitor the number of errors on the HTTP endpoints exposed (metric).



Here we see the main component of Prometheus, i.e, the server. It consists of three parts:

1. **Time Series Database (TSDB)** - Stores the metrics data. It also ingest it (append only), compacts and allows querying efficiently.
2. **Scrape Engine** - Pulls the metrics (description above) from our target resources and sends them to the TSDB. (Prometheus pulls are called scrapes).
3. **Server** - Used to make queries for the data stored in TSDB. This is also used to display the metrics in a dashboard using **Grafana/Prometheus UI**.



## More about Metrics

The metrics are defined with `TYPE` & `HELP` attributes to increase readability.

- `HELP` - It provides us with the description about the metric.

`TYPE`

Even tho Prometheus offers 4 core metric types to keep things simple, it allows us to create tags within those metric types for more specific use cases. The 4 core metric types are:

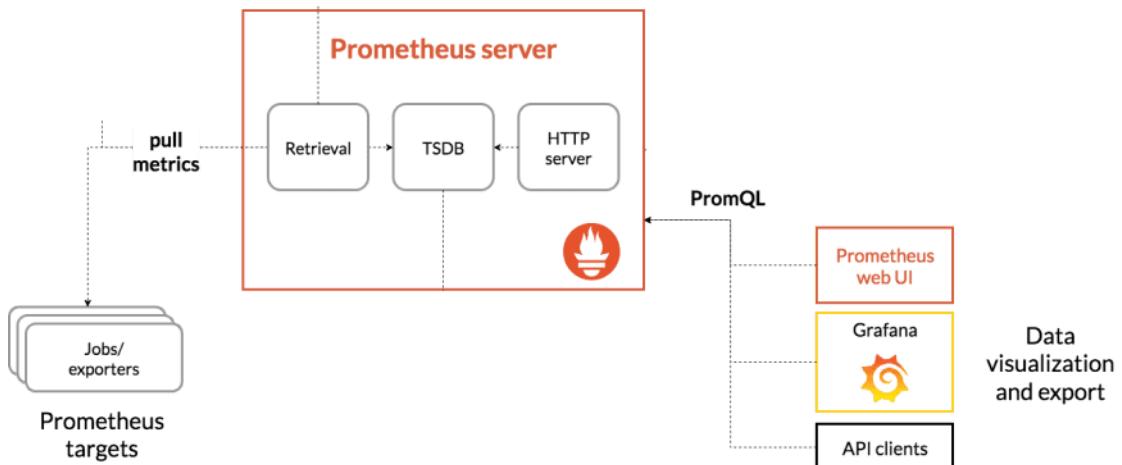
- **Counter** - As the name suggests, it is used to maintain a count of the metrics. This can be, let's say, number of requests, errors, etc. Note: Do not use this type if the value of your metric can decrease.
- **Gauge** - It is best suited for metrics that can go up & down, like CPU usage.
- **Histogram** - A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.
- **Summary** - Similar to a histogram, a summary samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

## How does it work?

### How does it get the data from Targets?

The Data Retrieval Worker pulls the data from the HTTP endpoints of the targets on path `/metrics`. Here we notice 2 things:

1. The endpoints should expose the path `/metrics`.
2. The data provided by the endpoint should be in the correct format that Prometheus understands.



**Q.** How do we make sure that the target services expose /metric & that data is in correct format?

**A.** Some of them expose the endpoint by default. Ones that do not, need a component to do so. This component is known as an **Exporter**. An Exporter does the following:

1. Fetch data from the target
2. Convert data into a format that Prometheus understands
3. Expose the /metrics endpoint (This can now be retrieved by the Data Retrieval Worker) For different types of services, like APIs, Databases, Storage, HTTP, etc, Prometheus has a [list of Exporters](#) you can use.

## Monitoring Personal Services

Let's say you want to monitor an application you have written in Java, you can use [Client Libraries](#) for that. It lets you expose application metrics via an HTTP endpoint `/metrics` on your application's instance which can then be used to send data to the Metrics Server. In the official documentation, a list of various libraries has been provided, with information on how to create your own.

## How is it different?

As mentioned above, Prometheus uses a **pull mechanism** to get data from targets. But mostly, other monitoring systems use a **push mechanism** (we'll see what that is in a bit). How is this different and what makes Prometheus so special?

**Q.** What do you mean by push mechanism?

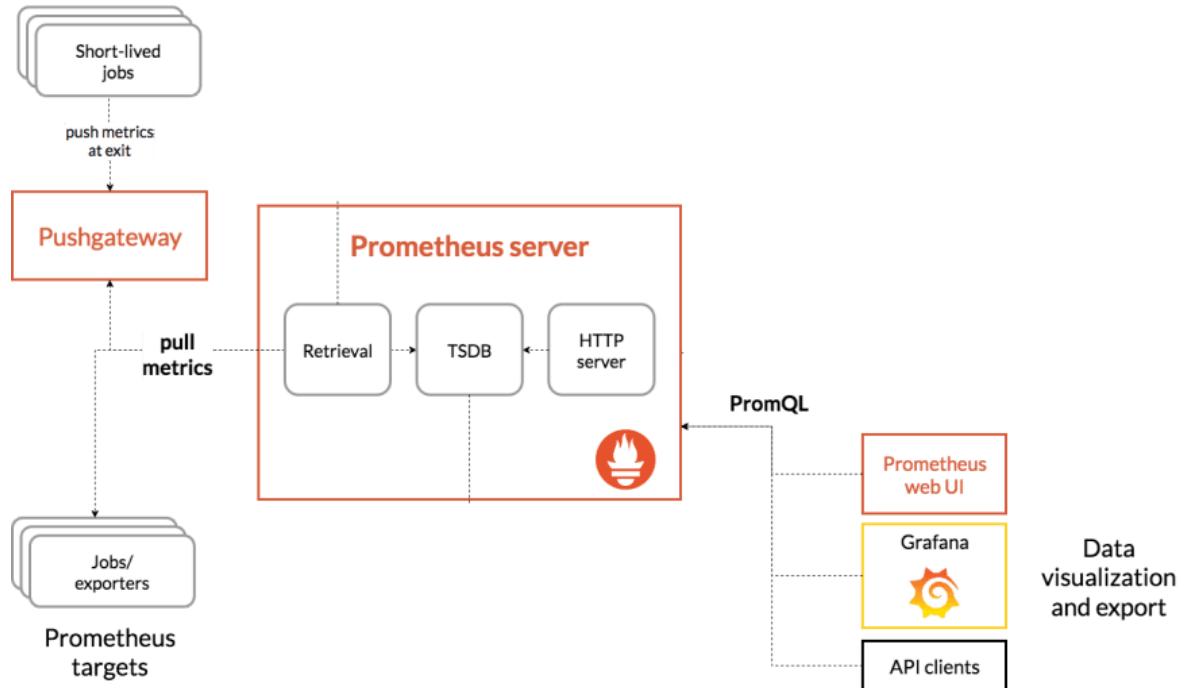
**A.** Instead of the server of the monitoring tool making requests to get the data, the servers of the application push the data to a database instead.

**Q.** Why is Prometheus better?

**A.** You can just get the data from the endpoint of the target, by multiple Prometheus instances. Also note that this way Prometheus can also monitor whether an application is responsive or not, rather than waiting for the target to push data.

(Checkout the [official comparison](#) documentation)

**NOTE:** But what happens if the targets don't give us enough time to make a pull request? For this, Prometheus uses the **Pushgateway**. Using this, these services can now push their data to the Data Retrieval Worker instead of it pulling data like it usually does. Using this, you get the best out of both the ways!

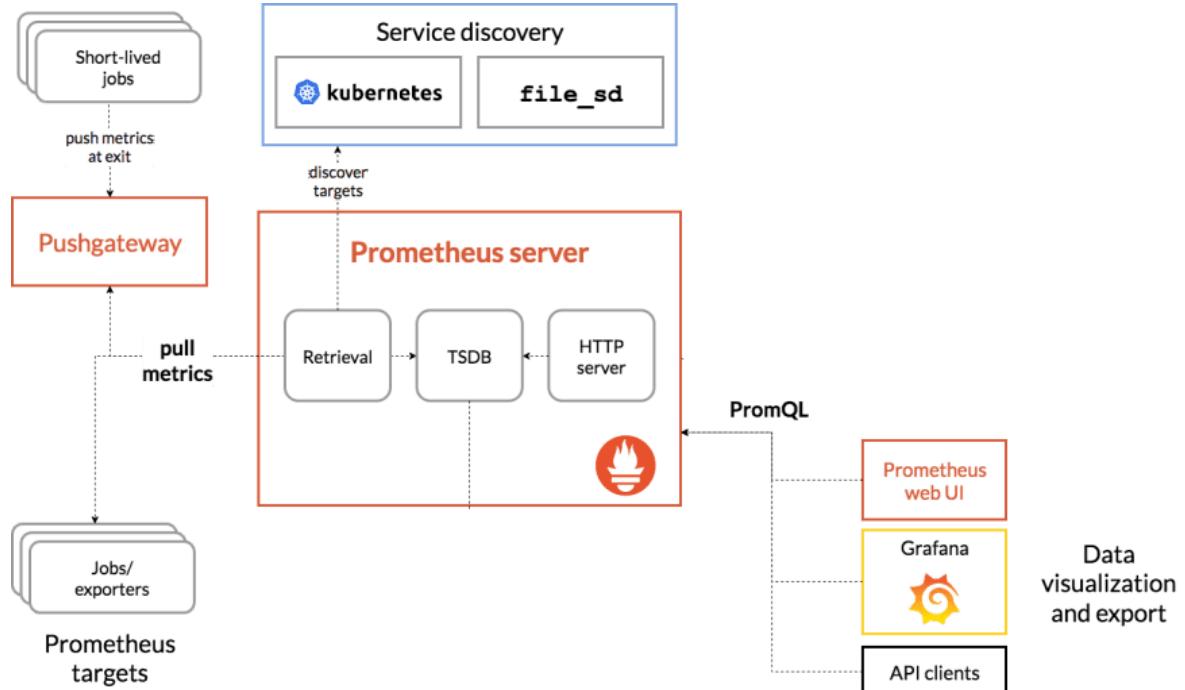


## How to use it?

Now that we know how Prometheus works, let's take a look into how we actually use it. So we mentioned about targets, metrics and all sorts of things. Where do we define those? Answer, in a **config (yaml)** file.

**Q.** When you define what targets you want to collect data from in the file, how does Prometheus find these targets

**A.** Using the Service Discovery. It also discovers services automatically based on the application running.



## Default Configuration File

(Check the [official documentation](#) for configuration)

```

global:
  scrape_interval:      15s
  evaluation_interval: 15s

rule_files:
  # - "first.rules"
  # - "second.rules"

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']

```

- `global` - `scrape_interval` defines how often Prometheus is going to collect data from the targets mentioned in the file. This can of course be overridden.

`rule_files`

This allows us to set rules for metrics & alerts. These files can be reloaded at runtime by sending

`SIGHUP`

to the Prometheus process. The

`evaluation_interval`

defines how often these rules are evaluated. Prometheus supports 2 types of such rules:

- **Recording Rules** - If you are performing some frequent operations, they can be precomputed and saved in as a new set of time series. This makes the monitoring system a bit faster.
- **Alerting Rules** - This lets you define conditions to send alerts to external services, for example, when a particular condition is triggered.
- `scrape_configs` - Here we define the services/targets that we need Prometheus to monitor. In this example file, the `job_name` is `prometheus`. Meaning that it is monitoring the target as the Prometheus server itself. In short, it will get data from the `/metrics` endpoint exposed by the Prometheus server. Here, the target by default is `localhost:9090` which is where Prometheus will expect the metrics to be, at `/metrics`.

## How does Alerting work?

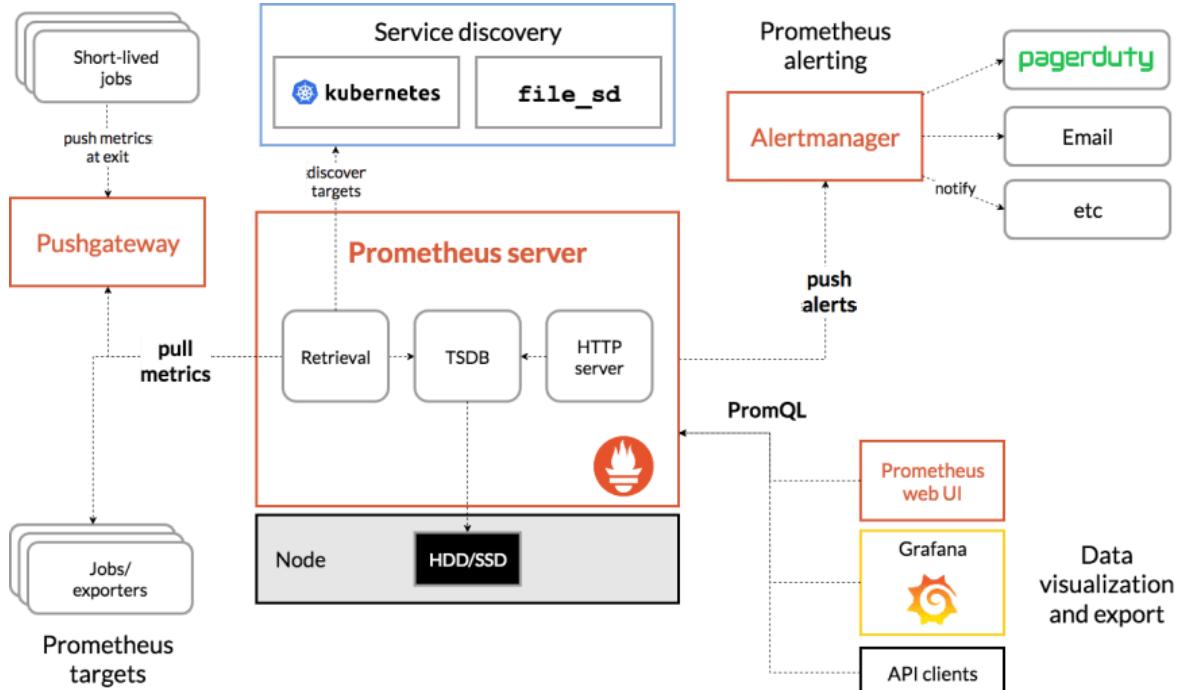
Prometheus has an **Alermanager** that can be used to send alerts to you via Emails, mailing lists, etc. As mentioned above, Prometheus server uses the **Alerting Rules** to send alerts.

## Where is the metrics data stored?

Prometheus stores it on disk, this can be a local database or remote. The data is stored in a time-series format so that one cannot write data directly.

## How to get the data?

Prometheus lets use get the metrics data using the **PromQLQuery Language**. You can use a Web UI to request data from Prometheus server via PromQL.

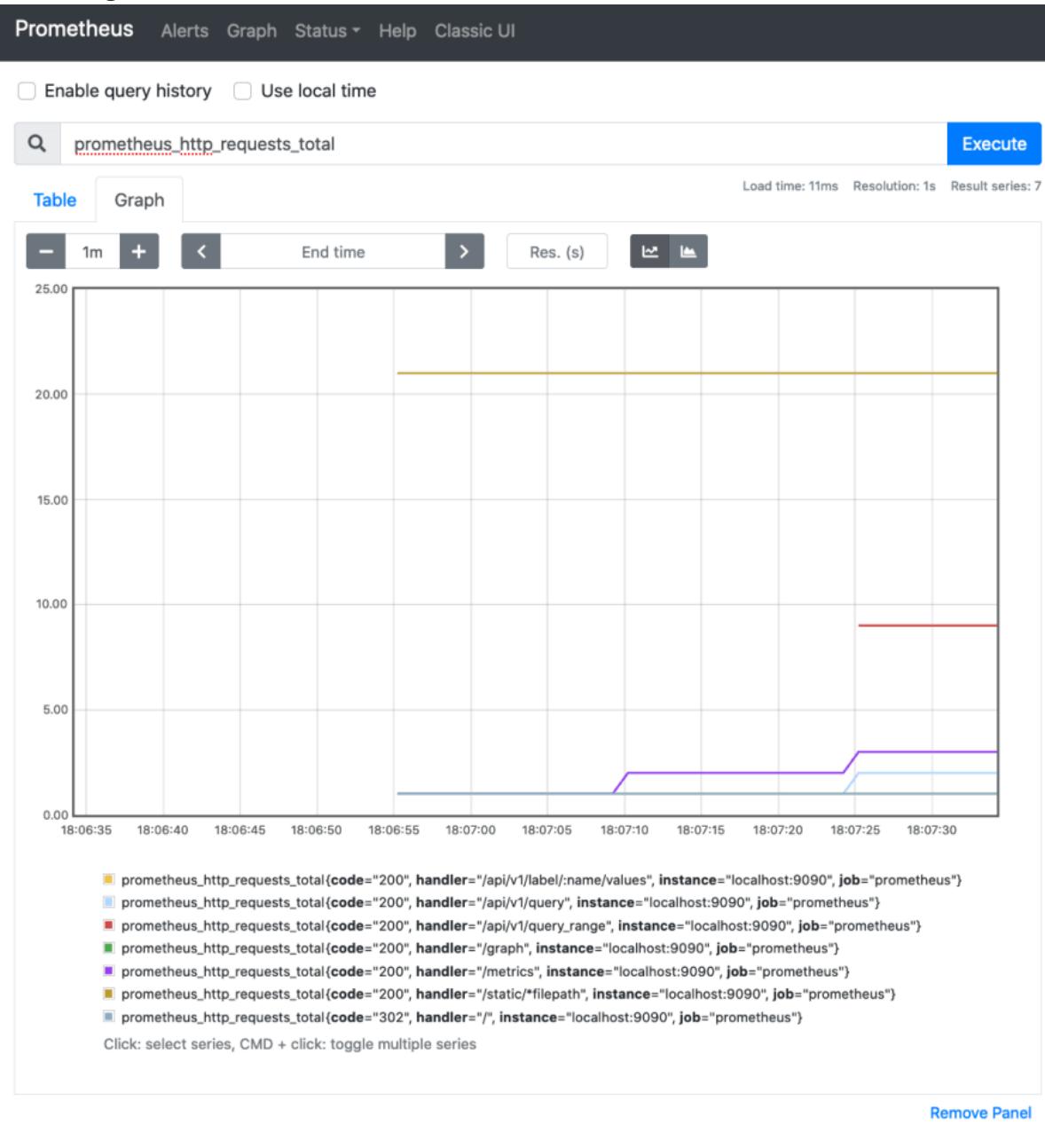


## Running it Locally

Let's take the example of the configuration file (`config.yml`) above that monitors the Prometheus server running on our machine.  
(Checkout the [README.md](#) file for more information)

```
$ mkdir -p $GOPATH/src/github.com/prometheus
$ cd $GOPATH/src/github.com/prometheus
$ git clone https://github.com/prometheus/prometheus.git
$ cd prometheus
$ make build
$ ./prometheus --config.file=your_config.yml$ mkdir -p
$GOPATH/src/github.com/prometheus
$ cd $GOPATH/src/github.com/prometheus
$ git clone https://github.com/prometheus/prometheus.git
$ cd prometheus
$ make build
$ ./prometheus --config.file=config.yml
```

Running it on `localhost:9090`, you'll get the following Prometheus UI Dashboard that you can now configure:



# Exposition

## Общие сведения о процессе экспозиции

Процесс предоставления данных для Prometheus называется **exposition**. Prometheus собирает данные по HTTP протоколу, в формате plain text с кодировкой UTF-8.

Начиная с версии 0.4.0 это единственный способ, а до этого была еще возможность предоставлять данные в виде **Protobuf**.

Для комментариев используется #, с одним исключением: после # не следует HELP или TYPE – это зарезервированные токены. В случае HELP ожидается, как минимум, название метрики. После названия может содержать любое количество произвольных символов с описанием метрики. TYPE ожидает 2 токена: название метрики и тип счетчика – и должен быть объявлен до первого использования метрики.

Метрики, в общем, имеют следующий формат:

```
metric_name{label_name="label_value",label_name="label_value"} value timestamp
```

Источники данных для Prometheus:

1. Для экспозиции уже существующих метрик из разнообразных источников, таких как nginx, Linux или Postgres, используются exporters. **Exporter** – это стороннее приложение, которое собирает метрики и отдает их в понятном для prometheus виде. На данный момент часть exporters поддерживается командой Prometheus, часть поддерживается open source сообществом. Полный список exporters представлен в [документации](#) и постоянно пополняется.
2. Если же Вы не нашли подходящего именно Вам exporter, то всегда можно написать его самостоятельно. Для этого есть готовые библиотеки, которые существенно облегчают процесс написания exporter. Библиотеки есть для большинства современных языков программирования.
3. Также можно встроить процесс экспозиции прямо в свое приложение. Для этого есть готовые библиотеки, которые существенно облегчают этот процесс. Эти библиотеки также имеются для многих языков программирования.

## Node Exporter

Собирает базовые системные метрики по умолчанию - CPU, жесткий диск, память, сеть.

Может производить экспозицию произвольных метрик из файла.

Умеет также предоставлять информацию о состоянии systemd сервисов.

Работает только под Linux. Для Windows нужен WMI-экспортер.

Не имеет собственного конфиг-файла, вся настройка с помощью ключей. Лучшая практика - не запускать из командной строки напрямую, а сделать systemd - сервис.

Порт по умолчанию - 9100. При переходе на страницу будет единственная ссылка, по которой будут показаны все доступные метрики.

Перед каждой метрикой - 2 поля: #HELP и #TYPE.

**#HELP** - описание метрики

**#TYPE** - тип метрики

В конце списка будут добавленные метрики из файлов.

## Типы метрик

Всего 4 типа метрик.

### Counter

A *counter* is a cumulative metric that represents a single [monotonically increasing counter](#) whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.

Do not use a counter to expose a value that can decrease. For example, do not use a counter for the number of currently running processes; instead use a gauge.

Client library usage documentation for counters:

- [Go](#)
- [Java](#)
- [Python](#)
- [Ruby](#)

## Gauge

A *gauge* is a metric that represents a single numerical value that can arbitrarily go up and down.

Gauges are typically used for measured values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of concurrent requests.

Client library usage documentation for gauges:

- [Go](#)
- [Java](#)
- [Python](#)
- [Ruby](#)

## Histogram

A *histogram* samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

A histogram with a base metric name of `<basename>` exposes multiple time series during a scrape:

- cumulative counters for the observation buckets, exposed as `<basename>_bucket{1e="<upper inclusive bound>"}`
- the **total sum** of all observed values, exposed as `<basename>_sum`
- the **count** of events that have been observed, exposed as `<basename>_count` (identical to `<basename>_bucket{1e="+Inf"}` above)

Use the [histogram\\_quantile\(\) function](#) to calculate quantiles from histograms or even aggregations of histograms. A histogram is also suitable to calculate an [Apdex score](#). When operating on buckets, remember that the histogram is [cumulative](#). See [histograms and summaries](#) for details of histogram usage and differences to [summaries](#).

Client library usage documentation for histograms:

- [Go](#)
- [Java](#)
- [Python](#)
- [Ruby](#)

## Summary

Similar to a *histogram*, a *summary* samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

A summary with a base metric name of `<basename>` exposes multiple time series during a scrape:

- streaming  **$\phi$ -quantiles** ( $0 \leq \phi \leq 1$ ) of observed events, exposed as `<basename>{quantile="<\phi>"}`
- the **total sum** of all observed values, exposed as `<basename>_sum`

- the **count** of events that have been observed, exposed as `<basename>_count`

See [histograms and summaries](#) for detailed explanations of φ-quantiles, summary usage, and differences to [histograms](#).

Client library usage documentation for summaries:

- [Go](#)
- [Java](#)
- [Python](#)
- [Ruby](#)

## Установка Node Exporter

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и команды выполняются с root привилегиями.

1. Скачиваем архив с Node Exporter и распаковываем его.

Для установки используется версия: 0.18.1, последняя стабильная версия на момент подготовки курса.

```
# cd /tmp
# wget
https://github.com/prometheus/node_exporter/releases/download/v0.18.1/node_exporter-0.18.1.linux-amd64.tar.gz
# tar xvfz node_exporter-0.18.1.linux-amd64.tar.gz
# cd node_exporter-0.18.1.linux-amd64
# mv node_exporter /usr/local/bin/
```

Список версий можно посмотреть на странице [github](#).

2. Создадим файл со списком ключей для запуска Node Exporter.

Node Exporter не имеет конфигурационного файла и настраивается дополнительными ключами. Для того, чтобы каждый раз не перечитывать настройки systemd, список ключей вынесен в файл: /etc/sysconfig/node\_exporter.

```
cat <<EOF > /etc/sysconfig/node_exporter
OPTIONS=""
EOF
```

3. Создаем systemd сервис.

```
cat <<EOF > /usr/lib/systemd/system/node_exporter.service
[Unit]
Description=Prometheus Node exporter for machine metrics
Documentation=https://github.com/prometheus/node_exporter

[Service]
Restart=always
User=root
EnvironmentFile=/etc/sysconfig/node_exporter
ExecStart=/usr/local/bin/node_exporter \$OPTIONS
ExecReload=/bin/kill -HUP \$MAINPID
TimeoutStopSec=20s
SendSIGKILL=no
```

```
[Install]
wantedBy=multi-user.target
EOF
```

С помощью директивы: EnvironmentFile все переменные из файла, указанного в этой директиве, добавляются в env, это позволяет нам передавать список ключей для Node Exporter через переменную `$OPTIONS` без изменения службы.

NB! Обратите внимание, Node Exporter требует root привилегий.

4. Обновляем список служб systemd, запускаем Node exporter и добавляем в автозапуск.

```
# systemctl daemon-reload
# systemctl start node_exporter
# systemctl enable node_exporter
```

5. Проверяем работу.

По умолчанию, Node Exporter слушает порт **9100**.

```
# curl -I http://localhost:9100
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK
Date: Fri, 08 Nov 2019 14:01:07 GMT
Content-Length: 150
Content-Type: text/html; charset=utf-8
```

## Ключи запуска Node Exporter

Далее приведен список наиболее востребованных ключей Node Exporter

```
--log.level
```

Default: info

Данный ключ устанавливает уровень логирования. Возможные уровни логирования: debug, info, warn, error.

```
--log.format
```

Default: logfmt

Данный ключ устанавливает формат логов. Доступные форматы: logfmt и json.

```
--web.listen-address
```

Default: ":9100"

Данный ключ устанавливает адрес и порт, по которому будет доступен Node Exporter.

```
--web.telemetry-path
```

Default: "/metrics"

Данный ключ устанавливает адрес, по которому доступны результаты экспозиции.

```
--web.disable-exporter-metrics
```

Default: -

Данный ключ устанавливает список метрик, которые будут исключены из экспозиции.

Например, для исключения всех метрик, имя которых начинается с go, значение ключа будет: go\*. Допускается использовать перечисление нескольких метрик, с запятой в качестве разделителя.

Полный список ключей можно просмотреть с помощью команды help.

```
# node_exporter --help
```

## Добавление метрик из файла

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

1. Создадим файл, который будет содержать метрики.

```
# mkdir /tmp/node_exporter/
cat <<EOF > /tmp/node_exporter/node_exporter_custom_metric.prom
#TYPE slurm_demo_metric counter
slurm_demo_metric 100
EOF
```

2. Настраиваем Node Exporter для чтения метрик из файла.

С этой целью для ключа `collector.textfile.directory` укажем в качестве значения путь каталога, содержащий файлы с метриками:

```
cat <<EOF > /etc/sysconfig/node_exporter
OPTIONS="--collector.textfile.directory=/tmp/node_exporter"
EOF
```

В указанном каталоге Node Exporter читает все файлы по маске \*.prom

3. Перезапустим службу Node Exporter:

```
# systemctl restart node_exporter
```

4. Проверим, что в экспозиции появилась новая метрика:

```
# curl -si http://localhost:9100/metrics | grep slurm
```

Ответ должен быть таким:

```
# HELP slurm_demo_metric Metric read from  
/tmp/node_exporter/node_exporter_custom_metric.prom  
# TYPE slurm_demo_metric counter  
slurm_demo_metric 100
```

## Blackbox Exporter

Этот экспортер - тестирование черного ящика, с угла зрения пользователя, поэтому его надо запускать отдельно.

Настройка разделена на 2 части: что мониторить - предоставляет Prometheus в качестве параметров OWL и остальная настройка - на стороне самого экспортёра.

Возможные протоколы проверок от экспортёра:

**ICMP** - скорость DNS запроса, время пинга, версия протоколов и результат проверки

**TCP** - скорость DNS запроса, установка подключения, результат, плюс по регекспу проверить нужный текст ответа от сервера

**DNS** - исключительно для проверки DNS-сервера

**HTTP** - скорость стадий запроса, наличие или отсутствие паттернов в ответе

Blackbox Exporter доступен по порту 9115, в отличие от Node Exporter - у него больше опций не веб-странице.

Раздел **Configuration** - текущая конфигурация.

Раздел **Metrics** - метрики, которые относятся к работе самого экспортёра.

Раздел **Debug** - тестовые проверки для демонстрации возможности экспортёра.

Раздел **Probe** - тестовая проверка. Отличается тем, что в ней не выводится дебаг-информация.

## Установка Blackbox Exporter

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

1. Скачиваем архив с Blackbox Exporter и распаковываем его.

Для установки используется версия: 0.18, последняя стабильная версия на момент подготовки курса.

```
cd /tmp  
wget  
https://github.com/prometheus/blackbox_exporter/releases/download/v0.18.0/blackbox_exporter-0.18.0.linux-amd64.tar.gz tar xvfz blackbox_exporter-0.18.0.linux-amd64.tar.gz  
cd blackbox_exporter-0.18.0.linux-amd64  
mv blackbox_exporter /usr/local/bin/
```

Список версий можно посмотреть на странице [github](#).

2. Создаем каталог для конфигурационных файлов и копируем конфигурационные файл по умолчанию.

```
mkdir /etc/blackbox_exporter  
cp blackbox.yml /etc/blackbox_exporter/
```

3. Создадим файл со списком ключей для запуска Blackbox Exporter.

Для того, чтобы каждый раз не перечитывать настройки systemd, список ключей вынесен в файл: `/etc/sysconfig/blackbox_exporter`

```
cat <<EOF > /etc/sysconfig/blackbox_exporter  
OPTIONS="--config.file=/etc/blackbox_exporter/blackbox.yml"  
EOF
```

!NB Поскольку Blackbox Exporter имеет конфигурационный файл и по умолчанию ищет его в том же каталоге, где и бинарный файл, то через ключ `--config.file` мы задаем путь до конфигурационного файла.

4. Создаем systemd сервис:

```
cat <<EOF > /usr/lib/systemd/system/blackbox_exporter.service  
[Unit]  
Description=Prometheus exporter for machine metrics  
Documentation=https://github.com/prometheus/blackbox_exporter  
  
[Service]  
Restart=always  
User=root  
EnvironmentFile=/etc/sysconfig/blackbox_exporter  
ExecStart=/usr/local/bin/blackbox_exporter \$OPTIONS  
ExecReload=/bin/kill -HUP \$MAINPID  
TimeoutStopSec=20s  
SendSIGKILL=no  
  
[Install]  
WantedBy=multi-user.target  
EOF
```

С помощью директивы: `EnvironmentFile` все переменные из файла, указанного в этой директиве, добавляются в env. Это позволяет передавать список ключей для Node Exporter через переменную `$OPTIONS` без изменения службы.

!NB Обратите внимание, Blackbox Exporter **требует root привилегий** для выполнения некоторых проверок.

5. Обновляем список служб systemd и запускаем Blackbox Exporter.

```
systemctl daemon-reload  
systemctl start blackbox_exporter  
systemctl enable blackbox_exporter
```

6. Проверяем работу.

По **умолчанию**, Blackbox Exporter слушает порт **9115**

```
curl -I http://localhost:9115
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Tue, 05 Nov 2019 07:09:10 GMT
Content-Length: 544
```

NB! Особенности использования Blackbox Exporter:

1. По умолчанию, все проверки выполняются по IPv6. Рекомендуется изменить значение на IPv4. Для этого надо использовать директиву: `preferred_ip_protocol: ip4` в настройках каждого модуля.
2. По ссылке /metrics Blackbox Exporter отдает данные о своей работе, такие как: потребление сри, потребление ram и т.д. Результаты проверки доступны по url /probe.
3. Для получения подробностей по проверке в параметры запроса нужно добавить:  
`debug=true`

## Настройка Blackbox Exporter

### Общая структура конфигурационного файла.

1. Конфигурационный файл имеет формат yaml.
2. По умолчанию, Blackbox Exporter ищет конфигурационный файл с именем `blackbox.yml`, в том же каталоге, что и бинарный файл.
3. Общая структура конфигурационного файла:

```
module:
  icmp_slurm:
    prober: icmp
    icmp:
      preferred_ip_protocol: ip4
```

`icmp_slurm` – имя, по которому будет запускаться проверка. Это позволяет осуществить несколько проверок одного типа, с разными настройками.

Например, с использованием ipv4 и ipv6 протоколов:

```
module:
  icmp_v4_slurm:
    prober: icmp
    icmp:
      preferred_ip_protocol: ip4
  icmp_v6_slurm:
    prober: icmp
    icmp:
      preferred_ip_protocol: ip6
```

`prober`: задает, какой протокол будет использован для проверки. Возможные значения: `icmp` | `dns` | `tcp` | `http`.

Далее идут настройки, специфичные для каждого протокола.

## ICMP протокол

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Данная проверка предназначена для проверки доступности хостов по протоколу ICMP.

- Добавляем в конфигурационный файл проверку с использованием протокола ICMP и именем icmp\_slurm

```
cat <<EOF >> /etc/blackbox_exporter/blackbox.yml
icmp_slurm:
  prober: icmp
  timeout: 2s
  icmp:
    preferred_ip_protocol: ip4

EOF
```

NB! Для преобразования имен в ip используется dns, без учета файла hosts.

- Чтобы применить изменения, перезапускаем Blackbox Exporter:

```
systemctl restart blackbox_exporter
```

- Проверяем работу:

```
curl -is "http://localhost:9115/probe?module=icmp_slurm&target=slurm.io" | grep
probe_success
```

В запросе, в качестве параметра **module**, передается имя проверки, а в качестве параметра **target** – какой ресурс проверять. С помощью grep фильтруем результат, чтобы получить только результат проверки.

Результат должен быть таким:

```
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

- Полный список параметров для проверки по ICMP протоколу:

```
timeout
```

Default: scrape\_timeout

Время, после которого проверка будет считаться неудачной. NB! Если значение не задано, используется scrape\_timeout, который передал Prometheus.

```
preferred_ip_protocol
```

Default: ip6

Какой протокол используется для проверки. Допустимые значения: ip4 | ip6.

```
source_ip_address
```

Default: -

Если на сервере несколько IP адресов, можно указать, с какого ip будет проводиться проверка.

```
dont_fragment
```

Default: false

Разрешен ли бит фрагментации пакетов. !NB Работает только с linux и IPv4.

```
payload_size
```

Default: -

Размер пакета, который отправляется при выполнении проверки.

## DNS протокол

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Данная проверка предназначена для тестирования DNS серверов и наличия на них определенных записей.

1. Добавляем в конфигурационный файл проверку dns с именем dns\_slurm:

```
cat <<EOF >> /etc/blackbox_exporter/blackbox.yml
dns_slurm:
  prober: dns
  timeout: 2s
  dns:
    query_name: slurm.io
    preferred_ip_protocol: ip4
EOF
```

В данной конфигурации будет проверяться, может ли DNS сервер разрешить имя slurm.io.

2. Чтобы применить изменения, перезапускаем Blackbox Exporter:

```
systemctl restart blackbox_exporter
```

3. Проверяем работу:

```
curl -is "http://localhost:9115/probe?module=dns_slurm&target=8.8.8.8" | grep
probe_success
```

В запросе, в качестве параметра module, передаётся имя проверки, а в качестве параметра target – на какой DNS сервер будет отправлен запрос. С помощью grep фильтруем результат, чтобы получить только результат проверки.

Результат должен быть таким:

```
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

4. Полный список параметров для проверки по dns протоколу:

timeout

Default: scrape\_timeout

Время, после которого проверка будет считаться неудачной. NB! Если значение не задано, используется scrape\_timeout, который передал Prometheus.

preferred\_ip\_protocol

Default: ip6

Какой протокол используется для проверки. Допустимые значения: ip4 | ip6.

source\_ip\_address

Default: -

Если на сервере несколько IP адресов, можно указать, с какого ip будет проводиться проверка.

transport\_protocol

Default: udp

Протокол, по которому будет производиться проверка. Возможные значения: udp, tcp.

query\_name:

Default: -

Запрос, который будет отправлен на DNS сервер.

query\_type

Default: "ANY"

Тип записи, который будет запрашиваться. По умолчанию, запрашиваются все типы записей.

## TCP протокол

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Данная проверка предназначена для тестирования сервисов с использованием TCP протокола.

1. Добавляем в конфигурационный файл проверку по TCP протоколу с именем tcp\_slurm:

```
cat <<EOF >> /etc/blackbox_exporter/blackbox.yml
tcp_slurm:
  prober: tcp
  timeout: 2s
  tcp:
    query_response:
      - expect: "SSH-2.0-"
    preferred_ip_protocol: ip4
EOF
```

В данной конфигурации будет проверяться, присутствует ли в ответе строка: SSH-2.0-

2. Чтобы применить изменения, перезапускаем Blackbox Exporter:

```
systemctl restart blackbox_exporter
```

3. Проверяем работу:

```
curl -is "http://localhost:9115/probe?module=tcp_slurm&target=127.0.0.1:22" | grep probe_success
```

В запросе, в качестве параметра module, передаётся имя проверки, а в качестве параметра target – IP адрес хоста и порт, для которых будет выполнена проверка. С помощью grep фильтруем результат, чтобы получить только результат проверки.

Результат должен быть таким:

```
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

4. Полный список параметров для проверки по TCP протоколу:

```
timeout
```

Default: scrape\_timeout

Время, после которого проверка будет считаться неудачной. NB! Если значение не задано, используется scrape\_timeout, который передал Prometheus.

```
preferred_ip_protocol
```

Default: ip6

Какой протокол используется для проверки. Допустимые значения: ip4 | ip6.

```
query_response
```

- expect - проверка на наличие строки в ответе.
- send - позволяет задать, какой запрос будет отправлен на сервер.
- starttls - задает, будет ли использоваться tls при подключении, по умолчанию – false.

```
source_ip_address
```

Default: -

Если на сервере несколько IP адресов, можно указать, с какого ip будет проводиться проверка.

```
tls
```

Default: false

Использовать ли tls после подключения.

```
tls_config
```

Настройки для tls. Возможны следующие **настройки для tls**:

- **insecure\_skip\_verify** – проверять ли валидность сертификата. Значение по умолчанию – false.
- **ca\_file** – путь к файлу с корневыми сертификатами.
- **cert\_file** – путь к файлу с клиентским сертификатом.
- **key\_file** – путь к файлу с клиентским ключом.
- **server\_name** – строка для проверки имени сервера.

## HTTP протокол

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Данная проверка предназначена для проверки сервисов с использованием HTTP протокола.

1. Добавляем в конфигурационный файл проверку по HTTP протоколу с именем проверки http\_slurm:

```
cat <<EOF >> /etc/blackbox_exporter/blackbox.yml
http_slurm:
  prober: http
  timeout: 2s
  http:
    preferred_ip_protocol: ip4
    valid_http_versions: ["HTTP/1.1", "HTTP/2"]
    valid_status_codes: [200]
    fail_if_not_ssl: true
    method: GET
    fail_if_body_not_matches_regexp:
      - "Prometheus"
EOF
```

В данной конфигурации для проверки используется протокол ip v4, проверяется версия HTTP, код ответа, метод GET. Также проверка закончится с ошибкой, если не используется https или если в ответе отсутствует слово: Prometheus.

2. Чтобы применить изменения, перезапускаем Blackbox Exporter:

```
systemctl restart blackbox_exporter
```

3. Проверка.

```
curl -is "http://localhost:9115/probe?  
module=http_slurm&target=http://prometheus.io" | grep probe_success
```

В запросе, в качестве параметра module, передаётся имя проверки, а в качестве параметра target – адрес веб сайта, который проверяется. С помощью grep фильтруем результат, чтобы получить только результат проверки.

NB! Адрес сайта может передаваться вместе со схемой. Если схема не указана, то будет использоваться http.

Результат должен быть таким:

```
# HELP probe_success Displays whether or not the probe was a success  
# TYPE probe_success gauge  
probe_success 1
```

4. Полный список параметров для проверки по HTTP протоколу:

`timeout`

Default: `scrape_timeout`

Время, после которого проверка будет считаться неудачной. NB! Если значение не задано, используется `scrape_timeout`, который передал Prometheus.

`preferred_ip_protocol`

Default: `ip6`

Какой протокол используется для проверки. Допустимые значения: `ip4` | `ip6`.

`source_ip_address`

Default: `-`

Если на сервере несколько IP адресов, можно указать, с какого ip будет проводиться проверка.

`valid_status_codes`

default: `200`

Проверка считается неудачной, если код не соответствует заданному.

`valid_http_versions`

Проверка считается неудачной, если версия HTTP не соответствует строке.

`method`

Default: `GET`

Тип запроса. Возможные значения: `GET` | `POST`.

## headers

Список заголовков, которые передаются во время проверки.

## no\_follow\_redirects

Default: false

Следовать ли редиректам при проверке.

## fail\_if\_ssl

Default: false

Проверка считается неуспешной, если соединение установлено по https.

## fail\_if\_not\_ssl

Default: false

Проверка считается неуспешной, если соединение установлено без использования https.

## fail\_if\_body\_matches\_regexp

Проверка считается неуспешной, если в body присутствует строка удовлетворяющая регулярному выражению.

## fail\_if\_body\_not\_matches\_regexp

Проверка считается неуспешной, если в body отсутствует строка удовлетворяющая регулярному выражению.

## fail\_if\_header\_matches

Проверка считается неуспешной, если в ответе присутствует заголовок, значение которого удовлетворяет регулярному выражению.

- header – имя заголовка, который проверяется.
- regexp – регулярное выражение, которое проверяется в значении заголовка.
- allow\_missing – разрешить отсутствие заголовка. По умолчанию: false.

## fail\_if\_header\_not\_matches

Проверка считается неуспешной, если в ответе отсутствует заголовок, значение которого удовлетворяет регулярному выражению.

- header – имя заголовка, который проверяется.
- regexp – регулярное выражение, которое проверяется в значении заголовка.
- allow\_missing – разрешить отсутствие заголовка. По умолчанию: false.

## basic\_auth

- `username` – имя пользователя, которое используется для авторизации на проверяемом сайте.
- `password` – пароль, который используется для авторизации на проверяемом сайте.

```
bearer_token
```

Токен для bearer авторизации.

```
bearer_token_file
```

Файл, который содержит токен для bearer авторизации.

```
proxy_url
```

Адрес proxy сервера, если проверку необходимо выполнить через proxy сервер.

```
body
```

Body, которое передается вместе с запросом.

```
tls_config
```

- `insecure_skip_verify` – проверять ли валидность сертификата. Значение по умолчанию: `false`
- `ca_file` – путь к файлу с корневыми сертификатами.
- `cert_file` – путь к файлу с клиентским сертификатом.
- `key_file` – путь к файлу с клиентским ключом.
- `server_name` – строка для проверки имени сервера.

## Настройка со стороны Prometheus

Prometheus пока еще не установлен. В данном шаге приведена настройка Prometheus для работы с Blackbox Exporter. К ней необходимо будет вернуться после установки Prometheus, описанной в одной из следующих глав.

Пример настроек Prometheus для произведения скрайпинга с Blackbox Exporter:

```
scrape_configs:  
  - job_name: blackbox  
    metrics_path: /probe  
    params:  
      module: [http_slurm]  
    static_configs:  
      - targets:  
        - "http://www.prometheus.io"  
    relabel_configs:  
      - source_labels: [__address__]  
        target_label: __param_target  
      - source_labels: [__param_target]  
        target_label: instance  
      - target_label: __address__  
        replacement: 127.0.0.1:9115
```

Настройка скрайпинга для Blackbox Exporter сильно отличается от настройки для большинства exporters. Далее, разберем конфиг по порядку.

```
params:  
  module: [http_slurm]
```

Для работы Blackbox, в качестве параметров запроса, надо передавать, какой модуль должен использоваться для проверки. Эта часть конфигурационного файла добавляет к запросу параметр `module=http_slurm`. Здесь можно было бы передать и второй параметр, `target`. Но в этом случае для каждого URL придется делать отдельную конфигурацию. Поэтому рекомендуется делать это через переопределение `labels`.

```
static_configs:  
  - targets:  
    - "http://www.prometheus.io"
```

В отличие от большинства exporters, в секции задается не адрес и порт exporter, а адрес проверяемого ресурса.

А дальше идет `relabel_configs`, в котором происходит вся магия. Разберем по шагам, что происходит в `relabel_configs`.

```
relabel_configs:  
  - source_labels: [__address__]  
    target_label: __param_target
```

В `source label __address__` содержится адрес exporter – тот, который задается в `targets`. Данная часть конфига копирует значение из `__address__` (по сути, из `targets`) в `__param_target`. Это преобразование добавляет второй параметр, `target`, к запросу.

```
relabel_configs:  
  - source_labels: [__param_target]  
    target_label: instance
```

В этой части мы значение из `label __param_target` записываем в `instance`. Это преобразование нужно, чтобы проверка каждого ресурса в `label instance` в качестве значения имела адрес этого ресурса.

```
relabel_configs:  
  - target_label: __address__  
    replacement: 127.0.0.1:9115
```

Так как в `targets` у нас указаны адреса проверяемых ресурсов, то нужно сказать, как подключаться к exporter. Для этого в `label: __address__` записываем адрес Blackbox.

## Custom Exporter

Кастомный экспортер нужен тогда, когда не хватает возможностей готовых. Для кастомных экспортёров существуют библиотеки на большинстве популярных ЯП.

Метрики производительности приложения логично получать из самого приложения (инструментирование кода).

## Общие рекомендации по написанию exporter.

Есть ряд рекомендаций, которые пригодятся при написании exporter. Они не обязательны к исполнению, но их стоит придерживаться, так как они составлены на основе опыта по решению реальных проблем.

### 1. Выбор имени для метрик:

- Имена метрик должны быть в нижнем регистре.
- Для именования допускаются только символы: [a-zA-Z0-9:\_].
- `_sum`, `_count`, `_bucket` и `_total` суффиксы, которые используются для **Summaries**, **Histograms** и **Counters**. В других случаях использовать их не стоит.
- `process_` и `scrape_` являются зарезервированными префиксами.
- Из названия метрики должно быть понятно, что эта метрика собирает.
- В названии метрики должны содержаться единицы измерения.

Пример плохого именования:

```
http_request_duration
```

Из названия не ясны ни единицы, в которых производится измерение, ни к какому приложению метрика относится.

Пример хорошего именования этой же метрики:

```
nginx_http_request_duration_seconds_bucket
```

- Если есть число удачных запросов, число неудачных запросов и общее количество запросов, то лучше сделать 2 метрики: число удачных запросов и общее число запросов. В этом случае число неудачных запросов легко обработать.

### 2. Выбор labels:

- Ряд слов, использовать которые в имени метки не запрещено, но и не рекомендовано: `region`, `zone`, `cluster`, `az`, `datacenter`, `customer`, `dc`, `owner`, `stage`, `service`, `environment`, `job`, `instance`.
- Метку `le` стоит использовать только с Histograms, а метку `quantile` стоит использовать только с Summaries.
- Метки необходимо выбирать таким образом, чтобы сумма значений метрики имела значение. Например, read/write показатели лучше хранить как отдельные метрики, а не разделять с использованием labels.

### 3. Рекомендации по написанию exporters:

- Получение данных должно производиться, только когда Prometheus производит scraping.
- Не надо выставлять временные метки для метрик, это задача Prometheus.
- Выбирать порт для exporter стоит с учетом уже занятых портов. Список уже использованных портов размещен на [github](#).

## Пример exporter на языке go

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Ниже приведен пример кода для простейшей реализации exporter. Exporter написан на языке go. Для реализации exporter используются готовые библиотеки prometheus. Exporter возвращает стандартные метрики, связанные с потреблением ресурсов самим exporter; данные метрики являются частью функционала библиотеки. Также присутствует собственная метка с типом данных Counter. Данная метрика увеличивает свое значение каждые 2 секунды.

1. Код приложения необходимо скопировать на сервер в файл slurm\_exporter.go:

```
package main

import (
    "net/http"
    "time"
    log "github.com/Sirupsen/logrus"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promhttp"
    "github.com/prometheus/client_golang/promauto"
)

func recordMetrics() {
    go func() {
        for {
            opsProcessed.Inc()
            time.Sleep(2 * time.Second)
        }
    }()
}

var (
    opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
        Name: "slurm_example_exporter_processed_ops_total",
        Help: "The total number of processed events",
    })
)

func main() {
    recordMetrics()

    http.Handle("/metrics", promhttp.Handler())
    log.Info("Beginning to serve on port :2112")
    log.Fatal(http.ListenAndServe(":2112", nil))
}
```

2. Устанавливаем go на сервер:

```
yum install -y go
```

3. Устанавливаем необходимые зависимости:

```
go get -d -u github.com/prometheus/client_golang/prometheus
go get -d -u github.com/Sirupsen/logrus
```

Первая зависимость – это библиотека prometheus. Вторая библиотека – для реализации логирования.

4. Запускаем exporter:

```
go run slurm_exporter.go
```

5. Проверка работоспособности.

Откройте еще одно подключение к серверу monitoring и выполните в новой консоли:

```
curl http://localhost:2112/metrics
```

В результате вы должны получить полный список метрик, который предоставляет exporter.

## Application Library

Это самый эффективный способ мониторинга:

- проще и лучше всего сделать бизнес-метрики
- самый надежный источник данных для приложения - само приложение
- экономия потребляемых ресурсов

Надо не увлечься и не превращать систему мониторинга в систему трассировки - задача мониторинга узнать о том, что есть проблема, а не решать саму проблему! То есть не пытаться замониторить каждый возможный запрос, а вместо этого дать алерт, работайте в целом система или нет.

Далее уже - делать трейсинг запросов по необходимости уже в системе трассировки.

## Prometheus

### Основа

Является опенсорсом. Распространяется в виде бинарника либо исходников.

Обычно ставится в виде systemd-сервиса.

Большая часть настроек - в виде текстовых конфигов.

Имеет скучный веб-интерфейс, в основном используется для настройки и отладки, для визуализации используется Grafana.

UI находится по умолчанию по порту 9090.

Запросы в текстовом окне, по нажатию кнопки Execute будут выведены все вектора, удовлетворяющие условию поиска.

Раздел **Graph** по полученным данным вывести интерактивные графы.

Раздел **Alerts** - показывает все существующие алERTы и файл, из которого были считаны правила.

Раздел **Status** - информация о runtime, значения ключей, с которыми был запущен сервер

Раздел **Configuration** - срез конфига

Раздел **Rules** - все выполняющиеся правила и время на их выполнение

Раздел **Targets** - все источники данных

Раздел **Service Discovery** - все сервера, которые удалось найти с помощью Service Discovery, а также их labels

## Установка Prometheus

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и команды выполняются с root привилегиями.

1. Создаем пользователя Prometheus:

```
useradd prometheus --comment "Prometheus server user" --shell /bin/false
```

Для запуска Prometheus **не требуется root привилегий** и, с точки зрения безопасности, лучше его запускать от непривилегированного пользователя.

2. Скачиваем архив с **Prometheus** и распаковываем его.

Для установки используется версия: 2.13.1, последняя стабильная версия на момент подготовки курса.

```
cd /tmp
wget https://github.com/prometheus/prometheus/releases/download/v2.13.1/prometheus-2.13.1.linux-amd64.tar.gz
tar xvfz prometheus-2.13.1.linux-amd64.tar.gz
cd prometheus-2.13.1.linux-amd64
mv prometheus /usr/local/bin/
```

Список версий можно посмотреть странице [github](#).

3. Создаем каталог для конфигурационных файлов, копируем конфигурационные файлы по умолчанию и выставляем на них права:

```
mkdir /etc/prometheus
cp prometheus.yaml /etc/prometheus/
chown -R prometheus:prometheus /etc/prometheus
```

4. Создаем каталог для Prometheus DB и выставляем на него права:

```
mkdir /var/lib/prometheus
chown prometheus:prometheus /var/lib/prometheus
```

5. Создадим файл со списком ключей для запуска Prometheus.

Для того, чтобы каждый раз не перечитывать настройки systemd, список ключей вынесен в файл: `/etc/sysconfig/prometheus`

```
cat <<EOF > /etc/sysconfig/prometheus
OPTIONS="--config.file=/etc/prometheus/prometheus.yaml \
--storage.tsdb.path=/var/lib/prometheus/"
EOF
```

**NB!** Поскольку Prometheus имеет конфигурационный файл и, по умолчанию, ищет его в том же каталоге, где и бинарный файл, то через ключ `--config.file` задаем путь до конфигурационного файла.

Также с помощью ключа: `--storage.tsdb.path` задаем путь до каталога, где будет располагаться Prometheus DB.

6. Создаем systemd сервис:

```
cat <<EOF > /usr/lib/systemd/system/prometheus.service
[Unit]
Description=Prometheus Server
Documentation=https://github.com/prometheus/prometheus

[Service]
Restart=always
User=prometheus
Group=prometheus
EnvironmentFile=/etc/sysconfig/prometheus
ExecStart=/usr/local/bin/prometheus \$OPTIONS
ExecReload=/bin/kill -HUP \$MAINPID
TimeoutStopSec=20s
SendSIGKILL=no

[Install]
WantedBy=multi-user.target
EOF
```

С помощью директивы: `EnvironmentFile` все переменные из файла, указанного в этой директиве, добавляются в env, это позволяет передавать список ключей для Node Exporter через переменную `$OPTIONS` без изменения службы.

7. Обновляем список служб systemd и запускаем Prometheus:

```
systemctl daemon-reload
systemctl start prometheus
systemctl enable prometheus
```

8. Проверяем работу.

По умолчанию, Prometheus слушает порт **9090**.

```
curl -XGET -IL http://localhost:9090/graph
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK
Date: Thu, 14 Nov 2019 07:05:12 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
```

9. Проверка конфигурационного файла.

NB! В состав Prometheus входит утилита **promtool** для тестирования конфигурационных файлов. К слову, проверять конфигурационный файл после каждого изменения – хорошая привычка.

Данная утилита входит в состав Prometheus. Для установки скопируем ее аналогично Prometheus:

```
cp promtool /usr/local/bin/
```

Для проверки конфигурационного файла запускаем команду:

```
promtool check config /etc/prometheus/prometheus.yaml
```

## Основной конфигурационный файл Prometheus

- Путь к конфигурационному файлу задается через ключ: `--config.file` при запуске `prometheus`.
- Формат конфигурационного файла: **yaml**.
- В конфигурационном файле **не допускаются символы табуляции, только пробелы**.
- Общий вид конфигурационного файла:

```
global:  
  # Как часто собирать данные, значение по умолчанию.  
  [ scrape_interval: <duration> | default = 1m ]  
  
  # Значение тайм-аут для процесса сбора данных, значение по умолчанию.  
  [ scrape_timeout: <duration> | default = 10s ]  
  
  # Как часто перечитывается список правил.  
  [ evaluation_interval: <duration> | default = 1m ]  
  
  # Метки, добавляем по умолчанию.  
  external_labels:  
    [ <labelname>: <labelvalue> ... ]  
  
  # файл со списком правил  
rule_files:  
  [ - <filepath_glob> ... ]  
  
  # Конфигурация для сбора данных.  
scrape_configs:  
  [ - <scrape_config> ... ]  
  
  #Настройка для взаимодействия с Alert Manager.  
alerting:  
  alert_relabel_configs:  
    [ - <relabel_config> ... ]  
  alertmanagers:  
    [ - <alertmanager_config> ... ]  
  
  # Настройки связанные, с функционалом remote write.  
remote_write:  
  [ - <remote_write> ... ]  
  
  # Настройки связанные, с функционалом remote read.  
remote_read:  
  [ - <remote_read> ... ]
```

**NB!** Наиболее востребованные настройки, их определено стоит запомнить:

`scrape_interval` - как часто `prometheus` будет опрашивать источники данных  
`scrape_configs` - список источников данных

Разделы:

- `rule_files`
- `scrape_configs`
- `alerting`
- `remote_write`
- `remote_read`

будут рассмотрены в соответствующих разделах нашего курса.

## Ключи запуска Prometheus

Далее приведен список наиболее востребованных ключей `Prometheus`:

`--config.file`

Default: "prometheus.yml"

Путь к конфигурационному файлу `Prometheus`. Если путь не полный, то поиск производится по каталогу, из которого запущен `Prometheus`.

`--web.listen-address`

Default: "0.0.0.0:9090"

Адрес и порт, по которому доступны UI и метрики `Prometheus`.

`--web.read-timeout`

Default: 5m

Максимальное время ожидания ответа от сервера и закрытия idle подключений.

`--storage.tsdb.path`

Default: "data/"

Путь к каталогу для сохранения метрик.

`--storage.tsdb.retention.time`

Default: -

Как долго хранить данные метрик. Время хранения метрик, по умолчанию: 15 дней.

`--storage.tsdb.retention.size`

Default: 0B

[Экспериментальная] Максимальный размер, отведенный под хранение метрик. Допускаются следующие единицы: KB, MB, GB, TB, PB. Первыми удаляются наиболее старые данные.

```
--storage.remote.flush-deadline
```

Default: 1m

Как долго ожидать окончания процесса сохранения данных при restart и reload.

```
--rules.alert.for-outage-tolerance
```

Default: 1h

Максимальное задержки для "for".

```
--rules.alert.for-grace-period
```

Default: 10m

Минимальное время восстановления состояния.

```
--rules.alert.resend-delay
```

Default: 1m

Минимальное время ожидания до отправки сообщения в Alertmanager.

```
--alertmanager.notification-queue-capacity
```

Default: 10000

Максимальное число сообщений, ожидающих отправки в Alertmanager.

```
--alertmanager.timeout
```

Default: 10s

Timeout для отправки сообщений в Alertmanager.

```
--query.timeout
```

Default: 2m

Максимальное время для выполнения запроса, после этого запрос будет сброшен.

```
--query.max-concurrency
```

Default: 20

Максимальное число параллельно выполняющихся запросов.

```
--log.level
```

Default: info

Данный ключ устанавливает уровень логирования. Возможные уровни логирования: `debug`, `info`, `warn`, `error`.

```
--log.format
```

Default: logfmt

Данный ключ устанавливает формат логов. Доступные форматы: logfmt и json.

Полный список ключей можно просмотреть с помощью команды help:

```
prometheus --help
```

## Service Discovery

Есть 2 принципиально разных способа добавления источников данных:

- 1) Статическая конфигурация - задается список эндпоинтов, с которых надо производить scraping
- 2) Service Discovery - настраивается, откуда Prometheus может получить данные о новых хостах.

Сервис дисковери - это то, за что любят Prometheus.

2 источника данных для Service Discovery:

1. Локальный JSON-файл - похож на статический конфиг, но динамически перечитывается.
2. данные получаются из API провайдера, список провайдеров довольно большой - K8S, Consul, Openstack, AWS EC2, и т.д.

В зависимости от провайдера, для каждого хоста во втором случае еще получаются метаданные, специфичные для этого провайдера. Эти метаданные могут быть использованы в качестве лейблов, для настройки алертинга или тому подобного.

## Настройка scraping с использованием static\_config

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

1. Добавляем конфигурацию для произведения scraping с node\_exporter.

**Обратите внимание:** в targets надо указать ip адреса server1 и server2, которые указаны в ЛК и в письме. Устанавливать на них Node Exporter не требуется, он был установлен при подготовке стенда.

```
cat <<EOF >> /etc/prometheus/prometheus.yml
  - job_name: 'static_config'
    static_configs:
      - targets:
          - localhost:9100
          - <адрес сервера server1>:9100
          - <адрес сервера server2>:9100
        labels:
          sd: static
EOF
```

**NB!** Обратите внимание, что хост задается с указанием порта, по которому необходимо проводить scraping.

С помощью директивы labels указывается, что ко всем метрикам будет добавлена метрика sd со значением static.

2. Выполните reload для Prometheus, чтобы применить новые настройки:

```
systemctl reload prometheus.service
```

3. Проверяем работу.

Открываем в браузере:

```
http://<IP>:9090/targets
```

где IP необходимо заменить на IP Вашего основного сервера.

## Настройка service discovery с использованием файлов

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

1. Создаем каталог для хранения правил service discovery и выставляем на него права:

```
mkdir /etc/prometheus/sd  
chown prometheus:prometheus /etc/prometheus/sd
```

2. Создадим файл со списком хостов:

```
cat <<EOF > /etc/prometheus/sd/node_exporter.yml  
- targets:  
  - localhost:9100  
  - <адрес сервера server1>:9100  
  - <адрес сервера server2>:9100  
labels:  
  sd: file  
EOF
```

3. Добавляем настройку в основной конфиг Prometheus:

```
cat <<EOF >> /etc/prometheus/prometheus.yml  
- job_name: 'file_sd'  
  file_sd_configs:  
    - files:  
      - /etc/prometheus/sd/*.yaml  
    refresh_interval: 1m  
EOF
```

С помощью директивы file\_sd\_config задается, что будет использоваться discovery на основании файлов.

С помощью директивы files задается список файлов, на основании которых будет производиться service discovery. Допускается использование регулярных выражений.

С помощью директивы refresh\_interval задается, как часто Prometheus будет перечитывать информацию из файлов.

4. Выполните reload для Prometheus, чтобы применить новые настройки:

```
systemctl reload prometheus.service
```

5. Если Вы все верно сделали, то в веб-интерфейсе должны появиться новые хосты.

## Настройка service discovery с использованием api openstack

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

1. Добавляем настройку в основной конфиг Prometheus.

Перед добавлением нужно подставить значения , , . Значения можете посмотреть в [личном кабинете](#).

```
cat <<EOF >> /etc/prometheus/prometheus.yml
- job_name: 'openstack'
  openstack_sd_configs:
    - identity_endpoint: https://api.selvpc.ru/identity/v3
      port: 9100
      domain_name: '82113'
      username: '<user name>' # s00000
      project_name: '<Project name>' # project-s00000
      password: '<Password>' # совпадает с ssh паролем
      role: 'instance'
      region: '<Region>' # ru-3
      refresh_interval: 30s

EOF
```

С помощью директивы port задается порт, по которому будет производиться scraping. Все остальные директивы относятся к подключению Openstack API.

2. Выполните reload для Prometheus, чтобы применить новые настройки:

```
systemctl reload prometheus.service
```

3. Если Вы все верно сделали, то в веб-интерфейсе должны появиться новые хосты.

## Labels

Помимо Service Discovery - есть мощный инструмент Labels.

Что такое лейблы?

Все временные ряды идентифицируются по параметрам типа "ключ-значение", которые и называются лейблами.

Ключ и значение могут быть любыми, и не являются обязательными.

Есть 2 источника для лейблов:

- лейблы, которые присваиваются автоматически в процессе экспозиции (например, `device`, `fstype`, `mountpoint` для дисковых разделов)

- которые Prometheus добавляет сам. Всегда добавляются как минимум 2 лейбла - это **Job** и **Instance**: Job - имя задачи, которая произвела скрейпинг, а Instance - это IP+порт хоста, откуда был произведен скрейпинг.

Если пользовательские лейблы имеют то же имя, что и системные - то Прометей их затрет.

## Модификаторы labels

Для модификации labels в Prometheus есть 2 варианта: использовать директивы **relabel\_configs** или **metric\_relabel\_configs**. **relabel\_configs** производит все модификации до scraping, а **metric\_relabel\_configs** производит изменения после scraping, но перед сохранением в базу.

**metric\_relabel\_configs** обычно используется в двух случаях: когда надо удалить метрику или переименовать ее.

**NB!** **metric\_relabel\_configs** не применяется к автоматически генерируемым метрикам, таким как up.

Далее приведен список действий, которые можно производить с метриками. Для **metric\_relabel\_configs** и **relabel\_configs** они одинаковы.

### 1. Labels action

С помощью action можно задавать правила scraping. Возможны значения: **keep** или **drop**. Если используется keep, будет производиться scraping метрик только с этим labels. В случае, если drop, то метрики с этим labels не будут сохраняться.

**Пример.** В этом случае будет производиться scraping только метрик, которые имеют label `_meta_openstack_tag` равный `db`:

```
scrape_configs:
  - job_name: 'openstack'
    openstack_sd_configs:
      - identity_endpoint: https://api.selvpc.ru/identity/v3
        port: 9100
        domain_name: '82113'
        username: '<user name>' # s00000
        project_name: '<Project name>' # project_s00000
        password: '<Password>'
        role: 'instance'
        region: '<Region>' # ru-3
        refresh_interval: 30s
    relabel_configs:
      - source_labels: [_meta_openstack_tag]
        regex: db
        action: keep
```

**Пример.** В этом случае будет производиться scraping метрик, у которых значение labels `_meta_openstack_tag` равно: `front`, либо `db`, либо `back` не будут сохранены.

```
scrape_configs:
  - job_name: 'openstack'
    openstack_sd_configs:
      - identity_endpoint: https://api.selvpc.ru/identity/v3
        port: 9100
        domain_name: '82113'
```

```

username: '<user name>' # s00000
project_name: '<Project name>' # project_s00000
password: '<Password>'
role: 'instance'
region: '<Region>' # ru-3
refresh_interval: 30s
relabel_configs:
- source_labels: [__meta_openstack_tag]
  regex: front|db|back
  action: drop

```

**NB!** Важный момент: **все метки, начинающиеся с \_**, в конце обработки **отбрасываются** и не попадают в target labels.

**NB!** Для сокращения правил лучше использовать regexp.

## 2. Labels replace

Например, если есть несколько команд, работающих над проектом, они могут использовать разное именование одних и тех же меток. В этом случае можно привести все к единому виду:

```

relabel_configs:
- source_labels: [team]
  regex: '(.*)ing'
  replacement: '${1}'
  target_label: team
  action: replace

```

Обратите внимание: тут мы снова используем regexp.

**Пример.** Если метрика имеет метку `team` со значением `developing`, то в результате преобразования значение `team` будет заменено на `develop`.

NB! Несмотря на то, что тут возможно использование regexp, их надо использовать с осторожностью, чтобы не заменить лишние метки.

## 3. Labels labelmap

**Labelmap** позволяет использовать имена source labels как имена для target labels. Именем для новой метки будет часть, удовлетворяющая регулярному выражению. Например, если есть source label `__meta_filepath`, то конфигурация:

```

relabel_configs:
- regex: __meta_(filepath)
  replacement: '${1}'
  action: labelmap

```

добавит target label `filepath` со значением равным значению `__meta_filepath`.

## 4. Labels list

Некоторые SD не имеют значений key:value, вместо которых у них список. Вы можете за счет регулярного выражения выделить нужное значение. В этом примере, если в списке тэгов consul присутствует один из тэгов с именем: `prod|staging|dev`, то имя этого тэга сохранится как значение label `env`.

```
relabel_configs:  
  - source_labels: [__meta_consul_tag]  
    regex: '.*,(prod|staging|dev),.*'  
    target_label: env
```

## 5. Labels labeldrop

Все метки, удовлетворяющие регулярному выражению, будут удалены.

```
relabel_configs:  
  - regex: sd  
    action: labeldrop
```

## 6. Labels labelkeep

Все метки, не удовлетворяющие регулярному выражению, будут удалены.

```
relabel_configs:  
  - regex: sd  
    action: labelkeep
```

**NB!** Этот модификатор необходимо использовать с осторожностью, так как все служебные метки(относятся к source labels) тоже удаляются. В примере выше будут удалены, в частности, такие метки, как: **address** и **scheme**, что сделает невозможным проведение scraping с exporter.

Labels, которые относятся к Target Labels - удалить нельзя!

# Push Gateway

Если источник данных недолго живет, и Прометея не успевает его заскрайпить (например Cron Job) - для этого нужен Push Gateway.

По завершению Batch Job - данные отправляются в Push Gateway.

И далее Прометея уже по расписанию забирает данные с Push Gateway. По сути это посредник между недолгоживущими источниками данных и Прометеем.

У Push Gateway есть особенности, благодаря которым его можно использовать только для определенного набора задач:

- нельзя автоматически проверять, насколько живо приложение
- это точка отказа в случае нескольких инстансов, передающих данные через Push Gateway
- данные хранятся вечно и удалены они могут быть только через его API

## Установка Pushgateway

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и команды выполняются с root привилегиями.

1. Создаем пользователя для запуска Pushgateway:

```
useradd pushgateway --comment "Pushgateway server user" --shell /bin/false
```

Для запуска PushGateway **не требуется root привилегий** и, с точки зрения безопасности, лучше его запускать от непривилегированного пользователя.

2. Скачиваем архив с PushGateway и распаковываем его.

Для установки используется версия: 1.0.0, последняя стабильная версия на момент подготовки курса.

```
cd /tmp
wget https://github.com/prometheus/pushgateway/releases/download/v1.0.0/pushgateway-1.0.0.linux-amd64.tar.gz
tar xvfz pushgateway-1.0.0.linux-amd64.tar.gz
cd pushgateway-1.0.0.linux-amd64
mv pushgateway /usr/local/bin/
```

Список версий можно посмотреть на странице [github](#).

3. Создадим файл со списком ключей для запуска PushGateway.

PushGateway **не имеет конфигурационного файла** и настраивается дополнительными ключами. Для того, чтобы каждый раз не перечитывать настройки systemd, список ключей вынесен в файл: `/etc/sysconfig/pushgateway`

```
cat <<EOF > /etc/sysconfig/pushgateway
OPTIONS="--web.listen-address=:9091 \
--web.telemetry-path=/metrics \
--persistence.file=/tmp/metric.store \
--persistence.interval=5m \
--log.level=info \
--log.format=json"
EOF
```

4. Создаем systemd сервис:

```
cat <<EOF > /usr/lib/systemd/system/pushgateway.service
[Unit]
Description=Prometheus Pushgateway
Documentation=https://github.com/prometheus/pushgateway

[Service]
User=pushgateway
Group=pushgateway
EnvironmentFile=/etc/sysconfig/pushgateway
Restart=always
ExecStart=/usr/local/bin/pushgateway \$OPTIONS
ExecReload=/bin/kill -HUP \$MAINPID
TimeoutStopSec=20s
SendSIGKILL=no

[Install]
WantedBy=multi-user.target
EOF
```

С помощью директивы: EnvironmentFile все переменные из файла, указанного в этой директиве, добавляются в env, это позволяет передавать список ключей для PushGateway через переменную \$OPTIONS без изменения службы.

5. Обновляем список служб systemd и запускаем node-exporter:

```
systemctl daemon-reload  
systemctl start pushgateway  
systemctl enable pushgateway
```

5. Проверяем работу.

По умолчанию, PushGateway слушает порт **9091**:

```
curl -XGET -IL http://localhost:9091
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK  
Date: Thu, 14 Nov 2019 07:11:24 GMT  
Content-Type: text/html; charset=utf-8  
Transfer-Encoding: chunked
```

## Ключи запуска PushGateway

Далее приведен список наиболее востребованных ключей Pushgateway:

```
--log.level
```

Default: info

Данный ключ устанавливает уровень логирования. Возможные уровни логирования: debug, info, warn, error.

```
--log.format
```

Default: logfmt

Данный ключ устанавливает формат логов. Доступные форматы: logfmt и json.

```
--web.listen-address
```

Default: ":9091"

Данный ключ устанавливает адрес и порт, по которому будет доступен PushGateway.

```
--web.telemetry-path
```

Default: "/metrics"

```
--web.enable-admin-api
```

Default: -

Разрешить endpoints для администрирования. Это может потребоваться для очистки данных pushgateway.

```
--persistence.file
```

Default: -

Файл для сохранения метрик. По умолчанию, значения метрик сохраняются только в памяти.

```
--persistence.interval
```

Default: 5m

Минимальное время, через которое данные будут сохранены в постоянный файл.

Полный список ключей можно просмотреть с помощью команды `help`:

```
pushgateway --help
```

## PushGateway API

Все взаимодействие с PushGateway производится через **API**. Он, конечно, имеет UI, но его можно использовать для просмотра и очистки данных.

Все labels, указанные в URL, используются в качестве ключа для группировки метрик.

### Структура URL

Имя задачи и labels передаются как часть URL. Все запросы начинаются с `/metrics`

```
/metrics/job/<JOB_NAME>{<LABEL_NAME>/<LABEL_VALUE>}
```

`<JOB_NAME>` - используется как значение label `job` и является обязательным.

Далее идут пары `<LABEL_NAME>/<LABEL_VALUE>`, где `<LABEL_NAME>` это имя label, а `<LABEL_VALUE>` значение этого label. Их количество не ограничено.

Например, метрика: `slurm_pushgateway_test{job="test_job",instance="localhost",edu="slurm"}` будет преобразована в:

```
/metrics/job/test_job/instance/localhost/edu/slurm/
```

NB! Если label содержит "/", пробел или русские символы, то его необходимо закодировать в **base64**. А при запросе указать `@base64`. Данная ключевое слово помогает PushGateway понять, что контент закодирован.

Например, для передачи в качестве значения для label path, `/var/tmp` запрос будет выглядеть так:

```
/metrics/job/test_job/path@base64/L3Zhc190bXA
```

### Метод PUT

PUT запрос замещает все метрики для группы. Код ответа может быть 200 – в случае успешного обновления, и 400 – в случае ошибки.

Если запрос выполняется с пустым body, то это приводит к удалению всех метрик для группы. При этом `push_times_seconds` не обновляется.

`push_times_seconds` – метка времени последнего удачного POST или PUT запроса.

`push_failure_time_seconds` – метка времени последнего неудачного POST или PUT запроса.

## Метод POST

POST запрос работает аналогично PUT, но обновляет значение только для метрик, которые присутствуют в запросе.

Если запрос выполняется с пустым body, то `push_times_seconds` обновляется, а изменение ранее переданных метрик не производится.

## Метод DELETE

DELETE запрос используется для удаления метрик из PushGateway. Удаляется вся группа метрик. Тело запроса всегда должно быть пустым.

При успешном выполнении код ответа всегда будет 202.

# Отправка данных в PushGateway

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Чтобы ознакомиться с работой PushGateway на практике, отправим в него 2 метрики. Первая метрика типа counter с label type, вторая – типа gauge.

1. Отправка данных в PushGateway (введите номер студента в команде):

```
cat <<EOF | curl --data-binary @-
http://127.0.0.1:9091/metrics/job/slurm_io_test_job/instance/monitoring.s< ваш
номер студента>.slurm.io
# TYPE slurm_io_edu_counter counter
slurm_io_edu_counter{type="counter"} 42
# TYPE slurm_io_gauge gauge
slurm_io_gauge 2398.283
EOF
```

2. Проверим результат:

```
curl -L http://localhost:9091/metrics/
```

Ответ должен быть примерно таким:

```
push_failure_time_seconds{instance="monitoring.s00000.slurm.io",job="slurm_io_te
st_job"} 0
push_time_seconds{instance="monitoring.s00000.slurm.io",job="slurm_io_test_job"}
1.5736548226639028e+09
# TYPE slurm_io_edu_counter counter
slurm_io_edu_counter{instance="monitoring.s00000.slurm.io",job="slurm_io_test_jo
b",type="counter"} 42
# TYPE slurm_io_gauge gauge
slurm_io_gauge{instance="monitoring.s00000.slurm.io",job="slurm_io_test_job"} 2398.283
```

Обратите внимание: в ответе присутствуют `push_failure_time_seconds` равный 0. Это говорит о том, что еще не было неудачных отправок данных.

Также присутствует переменная `push_time_seconds`, значение которой говорит о времени последней удачной отправки данных.

Также группировка произведена только по `labels`, которые передавались в URL. `label type` для `slurm_io_edu_counter` не учитывается.

## UI PushGateway

UI доступен по порту 9091. Интерфейс у него очень простой. В нем можно посмотреть список групп, значение для метрик для каждой группы. Время последней удачной и неудачной отправки данных. Можно также удалить группу.

## Настройка Prometheus

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняются с root привилегиями.

Настройка scraping с PushGateway не отличается от scraping с обычных exporters.

1. Добавляем настройки в основной конфигурационный файл Prometheus:

```
cat <<EOF >> /etc/prometheus/prometheus.yml
- job_name: pushgateway
  honor_labels: true
  static_configs:
  - targets:
    - localhost:9091
EOF
```

2. Применяем настройки:

```
sudo systemctl reload prometheus.service
```

3. Для проверки:

Выполните запрос в UI Prometheus:

```
slurm_io_edu_counter[1m]
```

В ответ вы должны получить вектор.

## PromQL

### Хранение данных

Для хранения данных используется специальная БД - Time-Series Database. Она предназначена для хранения данных, которые изменяются во времени. Грубо можно представить в виде набора таблиц:

Timestamp	Metric
2019-10-10 00:00:00	100
2019-10-10 00:00:15	50
2019-10-10 00:00:30	80

В отличие от реляционных БД, TSDB изначально разрабатывалась под такие структуры данных и обеспечивает наибольшую производительность.

В частности, в TSDB нельзя удалять произвольную строку, а можно только дописывать.

Линейное распределение данных, оптимизация внутренних компонентов.

Удаление устаревших данных - кусками и автоматически по мере устаревания (весь набор данных до определенного момента)

Нельзя задавать разные периоды хранения данных для различных метрик.

Данные будут удалены только **тогда, когда будет удален последний временной ряд**, содержащий эту метрику.

**PromQL** - язык запросов, позволяющий агрегировать данные временных рядов в реальном времени. Результат запроса:

- можно отобразить в виде таблицы
- можно отобразить в виде графика
- можно использовать внешними системами через HTTP API

Поддерживает:

- математические вычисления
- логические вычисления
- агрегацию
- бинарные сравнения
- функции времени и даты
- сортировки
- специфичные функции для различных типов данных

## Типы данных

---

Есть всего 4 типа данных:

- **Counter** - самый простой тип данных, счетчик. Может либо увеличиваться, либо принимать нулевое значение. Уменьшаться и уходить в минус не может. Подходит для количества обращений к серверу, либо для подсчета аптайма (инкремент кол-ва секунд)
- **Gauge** (шкала измерения) - тип данных, который может уходить и в плюс, и в минус. Можно сравнивать с термометром, отображает текущее значение метрики. Подходит, например, для отображения свободной памяти на сервере.
- **Histogram** - более сложный тип метрики. Собирает наблюдение, такие как длительность запроса и размер ответа, считает кол-во в настраиваемых диапазонах плюс считает сумму значений. Может считать среднее значение (сумма значений/кол-во значений). Подходит для отслеживания пропорций, или если нужно установить индикаторы качества (актуально для SLA или бизнес-метрик)

- **Summary** - расширенные гистограммы, показывают сумму и количество измерений, плюс квантильно-скользящий период, расчет квантиля - на стороне клиента. Например, если нужно посчитать 90ю или 95ю процентиль от времени отклика сервиса за период.

## Выражения

Результатом значения запроса PromQL является **вектор**.

Есть 2 типа векторов:

- **Instant vector** - содержит все значения метрики по запрашиваемой метке времени
- **Range vector** - возвращает все вектора за запрашиваемый период времени, это позволяет получить изменение метрики во времени

Мультипликатор **Offset** позволяет получить вектор из прошлого на то кол-во времени, которое в нем указано. Например, если нужно посмотреть CPU load 5 минут назад.

## Типы данных в PromQL

В запросах Prometheus есть 3 типа данных:

**Instant vector** – вектор содержит в себе все значения метрики по запрашиваемой метке времени.

**Range vectors** – возвращает все вектора за указанный период времени. Это позволяет увидеть изменение метрики во времени.

**Scalar** – простое числовое значение с плавающей точкой.

Для некоторых запросов есть ограничения для типа данных. Если в запросе могут быть использованы только определенные типы данных, это будет указано в описании запроса.

## Операторы PromQL

### Математические операторы:

+

Этот оператор производит сложение. Оператор сложения возможно использовать только с **instant vector и scalar**.

Пример:  $1+2$  результат будет 3.

-

Этот оператор производит вычитание. Оператор вычитания возможно использовать только с **instant vector и scalar**.

Пример:  $3-2$  результат будет 1.

\*

Этот оператор производит умножение. Оператор умножения возможно использовать только с **instant vector и scalar**.

Пример:  $2*2$  результат будет 4.

/

Этот оператор производит деление. Оператор деления возможно использовать только с **instant vector и scalar**.

Пример:  $8/4$  результат будет 2.

%

Этот оператор возвращает модуль исходных данных. Оператор % возможно использовать только с **instant vector и scalar**.

Пример:  $13 \% 5$  результат будет 3

$\wedge$

Этот оператор производит возведение в степень. Оператор возведения в степень возможно использовать только с **instant vector и scalar**.

Пример:  $2 \wedge 10$  результат будет 1024

## Операторы сравнения:

- `==` – равно
- `!=` – не равно
- `>` – больше
- `<` – меньше
- `>=` – больше или равно
- `<=` – меньше или равно

Сравнение возможно только над **scalar** и **instant vector**. Оператор применяется к каждому значению в исходном векторе, и все элементы данных, для которых не выполняется условие сравнения, исключаются из результирующего вектора.

Если указан модификатор **bool**, то в результирующем векторе данные, которые удовлетворяют условию сравнения, будут иметь значение 1, а не удовлетворяющие 0.

**NB!** Сравнение двух **scalar** возможно только с модификатором **bool**.

Чтобы лучше понять логику работы операторов сравнения, выполните последовательно следующие запросы и сравните результаты:

```
node_cpu_seconds_total  
node_cpu_seconds_total > 3  
node_cpu_seconds_total > bool 3
```

## Логические операторы:

- `and` – логическое И
- `or` – логическое ИЛИ
- `unless` – дополнение

Логические операторы возможны только между **instant vector**.

`vector1 and vector2` приводят к вектору, состоящему из элементов `vector1`, для которых в `vector2` есть элементы с точно совпадающими наборами меток. Другие элементы отброшены. Название и значения метрики переносятся из левого бокового вектора.

`vector1 or vector2` приводит к вектору, который содержит все исходные элементы (наборы меток + значения) `vector1` и дополнительно все элементы `vector2`, которые не имеют совпадающих наборов `vector1`.

`vector1 unless vector2` приводит к вектору, состоящему из элементов `vector1`, для которых нет элементов в `vector2` с точно совпадающими наборами меток. Все совпадающие элементы в обоих векторах отбрасываются.

Чтобы лучше понять логику работы операторов сравнения, выполните последовательно следующие запросы и сравните результаты:

```
node_load1 and node_cpu_seconds_total  
node_load1 or node_cpu_seconds_total  
node_load1 unless node_cpu_seconds_total
```

## Сопоставления векторов

Сопоставление векторов производит поиск одинаковых элементов в правом векторе для каждой записи левого вектора. Есть 2 типа сопоставления: один к одному и один ко многим / многие к одному.

### Сопоставление один к одному.

При сопоставлении один к одному для всех элементов из правого вектора ищется элемент, имеющий точно такой же набор меток (учитываются как имена метки, так и их значения), и над такими парами производится операция.

Существует 2 модификатора для сравнения:

- `on` – задает, какие `labels` необходимо учитывать при сопоставлении.
- `ignoring` – задает, какие `labels` должны быть исключены в процессе сопоставления.

Чтобы лучше понять данный механизм, попробуйте выполнить запрос:

```
node_cpu_seconds_total{instance="server1:9100",job="static_config"} + on(cpu,  
mode) node_cpu_seconds_total{instance="server2:9100",job="static_config"}
```

**Обратите внимание:** в запросе необходимо изменить значения `server1` и `server2` на ip адреса серверов.

В результате данного запроса вы получите суммарное потребление сри по двум серверам.

Сопоставление один ко многим / многие к одному: в этом случае одному элементу правого вектора может соответствовать несколько элементов второго вектора. Тогда необходимо явно указать, какой вектор является главным. Это можно сделать с помощью ключевых слов `group_left` и `group_right`.

## Операторы агрегации:

- `sum` – сумма
- `min` – минимальное значение
- `max` – максимальное значение
- `avg` – среднее значение
- `stddev` – стандартное отклонение
- `stdvar` – стандартная дисперсия
- `count` – количество элементов в векторе
- `count_values` – количество элементов с одинаковым значением.

Данные операторы могут использоваться с модификаторами **by** и **without**. С помощью модификатора **by** задается список `labels`, которые будут учитываться, а модификатор **without** задает список `labels`, которые учитываться не будут. Указание данного модификатора допускается как до, так и после запроса.

```
operator ([parameter,] <vector expression>) [without|by (<label list>)]
```

Например, чтобы получить суммарное потребление сри по всем инстансам в user mode, выполните следующий запрос:

```
sum(node_cpu_seconds_total{mode="user"}) by (cpu)
```

Чтобы получить суммарное потребление процессора в user mode по всем ядрам, выполните запрос:

```
sum without (cpu) (node_cpu_seconds_total{mode="user"})
```

## Приоритет бинарных операторов

Бинарные операторы имеют следующий приоритет:

1. `&`
2. `*` , `/` , `%`
3. `+` , `-`
4. `==` `!=` , `<=` , `<` , `>=` , `>`
5. `and` , `unless`
6. `or`

## Математические функции

```
abs
```

Функция возвращает абсолютные значения, то есть любые отрицательные значения заменяются положительными. `abs(vector(-10))` вернет: 10

```
ln, log2, and log10
```

Набор данных функций принимают мгновенный вектор и возвращают логарифм значений, используя разные основания.

`exp`

Функция возвращает экспоненту для моментального вектора. Эта функция является обратной для `ln`.

`sqrt`

Функция возвращает результат возведения в квадратный корень. Она эквивалентна математическому выражению  $\wedge 0.5$ .

`ceil` and `floor`

Функция округления значений вектора.

`ceil` – округляет в большую сторону: `ceil(vector(1.1))` вернет 2.

`floor` – округляет в меньшую сторону: `floor(vector(1.1))` вернет 1.

`round`

Функция возвращает результат округления. Округление производится до ближайшего целого числа.

`round (vector(1.1))` вернет: 1, `round (vector(1.6))` вернет: 2.

Если значение находится ровно посередине между двумя целыми числами, округление производится в большую сторону.

`round (vector(1.5))` вернет: 2.

Для функции `round` может быть задан дополнительный аргумент. В этом случае функция вернет ближайшее целое число кратное, заданному аргументу. `round(vector(17), 5)` вернет: 15

`clamp_max` and `clamp_min`

Функция `clamp_max` – заменяет все значения выше заданного на максимальное. `clamp_max(vector(9), 5)` вернет: 5.

Функция `clamp_min` – заменяет все значения меньше заданного на минимальное. `clamp_min(vector(3), 5)` вернет: 5.

## Функции времени и даты

`time`

Функция возвращает текущее время в Unix time формате.

`minute, hour, day_of_week, day_of_month, days_in_month, month, and year`

Данные функции возвращают:

```
minute - минуты  
hour - часы  
day_of_week - день недели  
day_of_month - день месяца  
days_in_month - количество дней в месяце  
month - месяц  
year - год
```

В качестве аргументов эти функции могут принимать вектор, значение которого – дата. Например, `year(process_start_time_seconds)` вернет год, в котором был запущен процесс.

```
timestamp
```

Это функция, в отличие от остальных функций времени, смотрит не на значение вектора, а на его временную метку и возвращает ее значение.

## Метки

```
label_replace
```

Функция позволяет добавить новую метку на основании уже имеющихся. Это удобно, когда вы агрегируете данные из разных источников, где метки имеют одинаковый смысл, а их названия различаются. Например, запрос `label_replace(up, "replace", "${1}", "job", "(.*)"")` вернет:  
`up{instance="localhost:9090",job="prometheus",replace="prometheus"}`

```
label_join
```

Позволяет объединять значения нескольких меток в одну. Например, запрос `label_join(up, "join", "-", "job", "instance")` вернет:  
`up{instance="localhost:9090",job="prometheus",join="prometheus-localhost:9090"}`

**Важно!** `label_join` и `label_replace` не удаляют имена метрик.

## Пропущенные серии

```
absent
```

Функция возвращает пустой вектор, если переданный ей вектор содержит значения. В противном случае она возвращает 1.

## Сортировки

```
sort
```

Функция позволяет отсортировать значения в возвращаемом векторе. Сортировка производится по возрастанию.

## sort\_desc

Функция позволяет отсортировать значения в возвращаемом векторе. Сортировка производится по убыванию.

## Агрегация во времени

### \_over\_time()

Следующие функции позволяют агрегировать каждую серию заданного диапазона вектора во времени и возвращать мгновенный вектор с результатами агрегации для каждой серии:

#### avg\_over\_time

Функция возвращает среднее значение всех точек в указанном интервале.

#### min\_over\_time

Функция возвращает минимальное значение всех точек в указанном интервале.

#### max\_over\_time

Функция возвращает максимальное значение всех точек в указанном интервале.

#### sum\_over\_time

Функция возвращает сумму всех значений в указанном интервале.

#### count\_over\_time

Функция возвращает количество всех значений в указанном интервале.

#### quantile\_over\_time

Функция возвращает  $\phi$ -квантиль ( $0 \leq \phi \leq 1$ ) значений в указанном интервале.

#### stddev\_over\_time

Функция возвращает стандартное отклонение совокупности значений в указанном интервале.

#### stdvar\_over\_time

Функция возвращает стандартную дисперсию совокупности значений в указанном интервале.

## Функции для Counter

### topk и bottomk

Функция вычисляет среднюю скорость увеличения временного ряда в секунду. При сбросе счетчика в 0, данные корректируются, чтобы это не влияло на конечный результат.

**NB!** topk и bottomk при использовании с by и without, в отличие от остальных операторов агрегации, возвращают полный набор метрик, а by и without используется только для группировки значений.

### increase

Функция вычисляет увеличение во временном ряду в диапазоне вектора. Формула для расчета:  
rate(x\_total [time]) \* time

### irate

Функция вычисляет мгновенную скорость увеличения временного ряда в векторе диапазона. Он похож на rate, но для анализа использует последние две выборки вектора.

### resets

Функция вычисляет число сбросов счетчика в предоставленном временном диапазоне в качестве мгновенного вектора. Любое уменьшение значения между двумя последовательными выборками интерпретируется как сброс счетчика.

## Функции для Histograms

### histogram\_quantile

Функция группирует значения по bucket, а затем вычисляет ф-квантиль ( $0 \leq \varphi \leq 1$ ). Функция rate() позволяет произвести расчет за период времени. Функция:

```
histogram_quantile(0.90, rate(prometheus_tsdb_compaction_duration_seconds_bucket[1d]))
```

рассчитывает 0,9 квантиль для prometheus\_tsdb\_compaction\_duration\_seconds\_bucket за предыдущий день.

Значения за пределами  $0 \leq \varphi \leq 1$  не имеют смысла и равняются бесконечности.

Предпочтительным способом расчета квантили является использование Summary, но некоторые exporters предоставляют данные в виде Histogram. В этом случае использование функции histogram\_quantile является обоснованным.

## Функции для Gauges

### changes

Функция позволяет подсчитать, сколько раз временной ряд изменил свое значение. Данная функция удобна, например, для подсчета количества перезапусков процесса за период времени.

### deriv

Функция позволяет узнать скорость изменения временного ряда в секунду за период времени. Эта функция похожа на `x - x offset 1h`, но на результат данного запроса могут повлиять локальные выбросы, а `deriv` для расчета значения использует функцию простой линейной регрессии, что делает ее результат точнее и устойчивее к локальным выбросам.

### `predict_linear`

Функция возвращает предсказание о значении временного ряда через `n` секунд. Предсказание вычисляется с помощью функции простой линейной регрессии.

### `delta`

Функция похожа на `increase` и возвращает изменение временного ряда за период времени, но без учета сбросов. Данная функция является чувствительной к локальным выбросам и использовать ее стоит с осторожностью.

### `idelta`

Функция возвращает разницу между последними 2-мя значениями во временном ряду.

### `holt_winters`

Функция реализует двойное экспоненциальное сглаживание Holt-Winters. Это полезно для очистки данных от локальных выбросов и оценки трендов изменения метрики. В качестве входных параметров функция принимает временной ряд, коэффициент сглаживания и коэффициент важности более старых данных по отношению к более новым.

## Record Rules

Есть возможность сохранять результаты запроса как новый временной ряд - это и называется **Record Rules**.

Если визуализируются данные, которые сложно высчитывать, то можно результаты сохранить в БД, чтобы заново их не пересчитывать, а брать уже подготовленные данные. Это дает существенное снижение нагрузки на Prometheus.

Пример такого выражения:

```
max_over_time(sum without(instance)(rate(x_total[5m]))[1h])
```

Еще полезно - когда Prometheus используется во внешних системах, например для автоматизации скейлинга, в этом случае имена для метрик выбираются самостоятельно, и если изначальные метрики поменяют свои названия, то достаточно поправить их названия в правилах, чтобы все работало.

## Настройка Rules

### Общие сведения

Rules описываются в отдельном конфигурационном файле. Формат - `yaml`. Prometheus поддерживает загрузку правил из нескольких файлов. Rules загружаются только при отсутствии ошибок во всех Rules. Для проверки на наличие синтаксических ошибок можно использовать утилиту `promtool`.

```
promtool check rules /path/to/example.rules.yml
```

**NB!** На файлы с правилами не выставляется inotify, поэтому после любого изменения правил требуется, как минимум, выполнять reload – для того, чтобы Prometheus применил изменения.

Чтобы загрузить правила в Prometheus, список файлов с Rules необходимо взять в основном конфигурационном файле Prometheus.

```
rule_files:  
  - "rules_file1.yml"  
  - "rules_file2.yml"  
  - "rules/*.yml"
```

В rules\_files задается список, также можно задавать имена файлов по маске.

### Синтаксис rules файла

Все правила объединяются в группы. Правила внутри группы запускаются последовательно, с регулярными интервалами. Разные группы выполняются в разное время, чтобы снизить нагрузку на Prometheus.

```
groups:  
  - name: example  
    interval: 10s  
    rules:  
      - record: job:process_cpu_seconds:rate5m  
        expr: sum without(instance)(rate(process_cpu_seconds_total[5m]))  
        labels:  
          slurm_edu: example_rules
```

**name** – это имя для группы правил. Оно должно быть уникальным и является обязательным.

**interval** – интервал, с которым будет производиться выполнение и сохранение правил в группе. Не является обязательным. По умолчанию равен global.evaluation\_interval.

**record** – имя, по которому результат будет доступен при извлечении данных. Рекомендации по составлению будут приведены в следующем шаге. Является обязательным.

**expr** – выражение, которое используется для вычисления. Является обязательным.

**labels** – список меток, которые будут добавлены к вектору. Не является обязательным.

## Именование Rules

Для Rules у Prometheus имеются рекомендации по именованию правил, что упрощает интерпретацию значения правила.

1. Разделителем в имени правила является ":"
2. Общий вид имени должен быть таким:

```
level:metric:operations
```

**level** – отражает уровень агрегации на основании labels. Он должен включать метку job и другие значимые labels, которые имеют отношение к метрике.

**metric** – это имя метрики. Допускается удаление *total*, но в остальных случаях это должно быть точное название исходной метрики. Для обозначения необходимо использовать *\_per*.

**operations** – представляет собой список функций и агрегаций, примененных к метрике. Если применяется несколько одинаковых функций, например *min*, указывать необходимо только одну.

Для лучшего понимания ниже приведено несколько примеров:

```
- record: instance_path:request_latency_seconds_count:rate5m
  expr: rate(request_latency_seconds_count{job="myjob"}[5m])

- record: instance_path:request_latency_seconds_sum:rate5m
  expr: rate(request_latency_seconds_sum{job="myjob"}[5m])

- record: job:request_failures_per_requests:ratio_rate5m
  expr: |2
    sum without (instance, path)
  (instance_path:request_failures:rate5m{job="myjob"})
  /
  sum without (instance, path)(instance_path:requests:rate5m{job="myjob"})
```

## Антипаттерны для использования Rules

Правила – очень мощный инструмент, но использовать их надо с осторожностью. Есть несколько антипаттернов использования правил.

1. Не стоит перемещать значения *labels* в название правил, это существенно сократит возможность обработки данных в дальнейшем.
2. Не стоит использовать *rules* для всех данных по умолчанию, это приведет к дополнительным накладным расходам. Как правило, 90% метрик не используются в повседневной работе и к ним обращаются только когда ищут источник проблем. Единственно когда, можно добавить правило - если без него запрос выполняется намного дольше.
3. Не стоит использовать *rules* для исправления имен или *labels*. Новый временной ряд записывается с новой временной меткой и приводит к потере исходной временной метрики.

## Настройка Rules

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняется с root привилегиями.

1. Сохраняем правила в файл: *rules.yml*

```
cat <<EOF > /etc/prometheus/rules.yml
groups:
- name: slurm-edu-example-node-exporter-rules
  rules:
    # The count of CPUs per node, useful for getting CPU time as a percent of
    # total.
    - record: instance:node_cpus:count
      expr: count(node_cpu_seconds_total{mode="idle"}) without (cpu, mode)

    # CPU in use by CPU.
    - record: instance_cpu:node_cpu_seconds_not_idle:rate5m
```

```
expr: sum(rate(node_cpu_seconds_total{mode!="idle"}[5m])) without (mode)

# CPU in use by mode.
- record: instance_mode:node_cpu_seconds:rate5m
  expr: sum(rate(node_cpu_seconds_total[5m])) without (cpu)

# CPU in use ratio.
- record: instance:node_cpu_utilization:ratio
  expr: sum(instance_mode:node_cpu_seconds:rate5m{mode!="idle"}) without (mode)
/ instance:node_cpus:count
EOF
```

## 2. Проверяем правила.

Для проверки правил также используется утилита `promtool`, отличие только в ключах запуска.

```
promtool check rules /etc/prometheus/rules.yml
```

## 3. Добавляем данные о Rules в основной конфигурационный файл Prometheus.

Открываем на редактирование файл: `/etc/prometheus/prometheus.yml` и после `rule_files` добавляем `" - rules.yml"`. В результате у вас должно получиться:

```
rule_files:
  - rules.yml
```

## 4. Проверяем корректность внесенных изменений:

```
promtool check config /etc/prometheus/prometheus.yml
```

## 5. Выполните `reload` для Prometheus, чтобы применить новые настройки:

```
systemctl reload prometheus.service
```

**NB!** Для применения изменений в `prometheus.yml` достаточно выполнить `reload` для службы. Этот способ предпочтительней, так как в случае ошибки в конфигурационном файле просто не применяются изменения. Если же выполнить `restart`, то из-за ошибки весь Prometheus станет недоступен до устранения этой самой ошибки.

## 6. Проверяем, что правила появились.

Открываем UI Prometheus, раздел `rules`. Для этого открываем в браузере: `http://<адрес сервера monitoring>:9090/rules`.

# Alerting

## Alerting Rules

За алертинг отвечают 2 подсистемы.

Первая - это часть самого Прометея, которая проверяет текущие данные на соответствие правилам алертинга. Если условие выполнено - выполняется вебхук к сторонней системе (`Alertmanager`, которая уже отправляет алерт по нужным каналам).

Вторая часть - собственно `Alertmanager`.

Такое разделение позволяет иметь один Alertmanager для нескольких Прометеев -> лучшая масштабируемость.

## Настройка Alert Rules

### Общие сведения

Alert Rules очень похожи на Record Rules, рассмотренные в предыдущей главе. Они располагаются в тех же группах, что и rules. Допускается комбинировать alert и record rules в одном файле. Но они также могут быть вынесены и в отдельный файл.

Пример:

```
groups:  
  - name: node_rules  
    rules:  
      - record: job:up:avg  
        expr: avg without(instance)(up{job="node_exporter"})  
      - alert: ManyInstancesDown  
        expr: job:up:avg{job="node_exporter"} < 0.5
```

### Синтаксис Alert Rules

Правила уведомлений также формируются по группам:

```
groups:  
  - name: example  
    rules:  
      - alert: ManyInstancesDown  
        for: 5m  
        expr: avg without(instance)(up{job="node_exporter"}) * 100 < 50  
        labels:  
          severity: warning  
        annotations:  
          summary: 'More than half of instances are down.'  
          dashboard: http://some.grafana:3000/dashboard/db/prometheus
```

**name** – это имя для группы правил. Оно должно быть уникальным и является обязательным.

**alert** – это имя алерта.

**for** – задержка с момента выполнения условий для срабатывания до начала обработки. Пока условие for не выполнено, данные в Alertmanager не отправляются.

**expr** – задает условие, и если оно истинное, то начинается обработка alert.

**labels** – задает произвольный набор labels для alert. NB! alert labels никак не связаны с metrics labels и используются для обработки алертов на стороне Alertmanager.

**annotations** – набор дополнительных произвольных меток (ключ: значение), которые передаются вместе с алертом. В отличие от меток, которые передаются в labels, метки из annotations не используются для группировки и маршрутизации уведомлений.

alert annotation поддерживает шаблоны GO, а также PromQL запросы. Например, можно добавить список недоступных node\_exporter

```

description: >
  Down instances: {{ range query "up{job=\"node\"} == 0" }}}
    {{ .Labels.instance }}
  {{ end }}

```

## Добавление правил уведомлений

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняется с root привилегиями.

1. Сохраняем правила в файл: rules\_alert.yml :

```

cat <<EOF > /etc/prometheus/rules_alert.yml
groups:
- name: PrometheusGroup
  rules:
  - alert: PrometheusConfigurationReload
    expr: prometheus_config_last_reload_successful != 1
    for: 1m
    labels:
      severity: critical
      service: prom
    annotations:
      summary: "Prometheus configuration reload (instance {{ \$labels.instance }})"
      description: "Prometheus configuration reload error\n  VALUE = {{ \$value }}\n  LABELS: {{ \$labels }}"
  - name: NodeExporterGroup
    rules:
    - alert: ExporterDown
      expr: up == 0
      for: 1m
      labels:
        severity: error
        service: prom
      annotations:
        summary: "Exporter down (instance {{ \$labels.instance }})"
        description: "Prometheus exporter down\n  VALUE = {{ \$value }}\n  LABELS: {{ \$labels }}"
    - alert: HighCpuLoad
      expr: 100 - (avg by(instance) (irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 80
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "High CPU load (instance {{ \$labels.instance }})"
        description: "CPU load is > 80%\n  VALUE = {{ \$value }}\n  LABELS: {{ \$labels }}"
    - alert: SystemdServiceCrashed
      expr: node_systemd_unit_state{state="failed"} == 1
      for: 5m
      labels:

```

```

severity: warning
annotations:
  summary: "SystemD service crashed (instance {{ \$labels.instance }})"
  description: "SystemD service crashed\n  VALUE = {{ \$value }}\n  LABELS:
{{ \$labels }}"
EOF

```

## 2. Проверяем правила.

Для проверки правил также используется утилита `promtool`, отличие только в ключах запуска.

```
promtool check rules /etc/prometheus/rules_alert.yml
```

## 3. Добавляем данные о Rules в основной конфигурационный файл Prometheus.

Открываем на редактирование файл: `/etc/prometheus/prometheus.yml` и после `rule_files` добавляем "- `rules_*.yml`". В результате у вас должно получиться:

```

rule_files:
  - rules.yml
  - rules_*.yml

```

## 4. Проверяем корректность внесенных изменений:

```
promtool check config /etc/prometheus/prometheus.yml
```

## 5. Выполните `reload` для Prometheus, чтобы применить новые настройки:

```
sudo systemctl reload prometheus.service
```

**NB!** Для применения изменений в `prometheus.yml` достаточно выполнить `reload` для службы. Этот способ предпочтительней, так как в случае ошибки в конфигурационном файле просто не применяются изменения. Если же выполнить `restart`, то из-за ошибки весь Prometheus станет недоступен до устранения ошибки.

## 6. Проверяем, что правила появились.

Открываем UI Prometheus, раздел `rules`. Для этого открываем в браузере: `http://<адрес сервера monitoring>:9090/rules`. Должны отображаться правила:

NodeExporterGroup		718ms ago	996.3us		
Rule		State	Error	Last Evaluation	Evaluation Time
<pre>alert: ExporterDown expr: up == 0 for: 5m labels:   severity: warning annotations:   description:  -&gt;     Prometheus exporter down     VALUE = {{ \$value }}     LABELS: {{ \$labels }} summary: Exporter down (Instance {{ \$labels.instance }})</pre>		OK		718ms ago	721.5us

Теперь переходим: `http://адрес сервера monitoring:9090/alerts`. На этой странице отображается состояние всех правил.

## Alerts

Show annotations

/etc/prometheus/rules\_alert.yml > NodeExporterGroup

**ExporterDown** (0 active)

```
alert: ExporterDown
expr: up == 0
for: 5m
labels:
  severity: warning
annotations:
  description: |-  
    Prometheus exporter down  
    VALUE = {{ $value }}  
    LABELS: {{ $labels }}  
  summary: Exporter down {instance {{ $labels.instance }}}}
```

**HighCpuLoad** (0 active)

**SystemServiceCrashed** (0 active)

/etc/prometheus/rules\_alert.yml > PrometheusGroup

**PrometheusConfigurationReload** (0 active)

## 7. Тестирование работы правил.

Отключаем node\_exporter, выполнив команду:

```
systemctl stop node_exporter.service
```

Возвращаемся на страницу: <http://<адрес сервера monitoring>:9090/alerts>. Одно из alert rule должно перейти в состояние PENDING

## Alerts

Show annotations

/etc/prometheus/rules\_alert.yml > NodeExporterGroup

**ExporterDown** (1 active)

```
alert: ExporterDown
expr: up == 0
for: 5m
labels:
  severity: warning
annotations:
  description: |-  
    Prometheus exporter down  
    VALUE = {{ $value }}  
    LABELS: {{ $labels }}  
  summary: Exporter down {instance {{ $labels.instance }}}}
```

Labels	State	Active Since	Value
alarmname="ExporterDown" instance="localhost:9100" job="static_config" sd="static" severity="warning"	PENDING	2019-11-15 12:21:30.427863294 +0000 UTC	0

**Annotations**

**description**

Prometheus exporter down VALUE = 0 LABELS: map[\_\_name\_\_:up instance:localhost:9100 job:static\_config sd:static]

**summary**

Exporter down (instance localhost:9100)

По истечению времени, указанного в for(5m), alert rule перейдет в состояние: FIRING

## Alerts

Show annotations

/etc/prometheus/rules\_alert.yml > NodeExporterGroup

**ExporterDown** (1 active)

```
alert: ExporterDown
expr: up == 0
for: 5m
labels:
  severity: warning
annotations:
  description: |-  
    Prometheus exporter down  
    VALUE = {{ $value }}  
    LABELS: {{ $labels }}  
  summary: Exporter down {instance {{ $labels.instance }}}}
```

Labels	State	Active Since	Value
alarmname="ExporterDown" instance="localhost:9100" job="static_config" sd="static" severity="warning"	FIRING	2019-11-15 12:21:30.427863294 +0000 UTC	0

**Annotations**

**description**

Prometheus exporter down VALUE = 0 LABELS: map[\_\_name\_\_:up instance:localhost:9100 job:static\_config sd:static]

**summary**

Exporter down (instance localhost:9100)

**NB!** Обратите внимание: Prom сам не обновляет страницы. Чтобы увидеть добавленные данные, необходимо обновить страницу (нажать F5).

**NB!** Установленная галочка: "Show annotations" позволяет просмотреть Annotations labels с уже подставленными в шаблон значениями.

## Alertmanager

Это больше чем конвертер алертов в SMS, email и сообщения в чате.

В идеальном мире один инцидент генерирует одно уведомление, в реальном мире такого не бывает. Alertmanager выстраивает конвейер обработки алертов, он состоит из следующих стадий:

- **Подавление** - если недоступны физические хосты, то не нужно слать алерты о недоступности каждой службы на этом хосте, а достаточно уведомить о самом хосте. Позволяет задавать период, когда уведомления не будут отправляться.
- **Роутинг** - разные алерты идут на разные команды
- Группировка - организация алертов по типу. Например, группировка уведомлений от СУБД по всем инстансам в одно уведомление.
- **Тротлинг и повтор** - задержка отправки отправлений, удобно при группировке, когда при аварии однотипные алерты приходят хоть и примерно в одно время, но разнесены по времени. Это как будильник, который срабатывает через определенные интервалы.
- **Отправка уведомлений.**

## Установка Alertmanager

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняется с root привилегиями.

1. Создаем пользователя alertmanager:

```
adduser --comment "User to run Alertmanager" --shell /bin/false alertmanager
```

Для запуска Alertmanager не требуется root привилегий и, с точки зрения безопасности, лучше его запускать от непrivилегированного пользователя.

2. Скачиваем архив с **Alertmanager** и распаковываем его.

Для установки используется версия: 0.19.0, последняя стабильная версия на момент подготовки курса.

```
cd /tmp
wget https://github.com/prometheus/alertmanager/releases/download/v0.19.0/alertmanager-0.19.0.linux-amd64.tar.gz
tar xvzf alertmanager-0.19.0.linux-amd64.tar.gz
cd alertmanager-0.19.0.linux-amd64
mv alertmanager /usr/local/bin/
```

Список версий можно посмотреть на странице [github](#).

3. Создаем каталог для конфигурационных файлов, копируем конфигурационные файлы по умолчанию и выставляем на них права:

```
mkdir /etc/alertmanager
cp alertmanager.yml /etc/alertmanager/alertmanager.yml
chown alertmanager:alertmanager -R /etc/alertmanager
```

4. Создаем каталог для хранения данных и выставляем на него права:

```
mkdir /var/lib/alertmanager/
chown alertmanager:alertmanager /var/lib/alertmanager/
```

5. Создадим файл со списком ключей для запуска Alertmanager.

Для того, чтобы каждый раз не перечитывать настройки systemd, список ключей вынесен в файл: /etc/sysconfig/alertmanager

```
cat <<EOF > /etc/sysconfig/alertmanager
OPTIONS="--config.file=/etc/alertmanager/alertmanager.yml \
--storage.path=/var/lib/alertmanager/"
EOF
```

**NB** Поскольку Alertmanager имеет конфигурационный файл и, по умолчанию, ищет его в том же каталоге, где и бинарный файл, то через ключ `--config.file` задаем путь до конфигурационного файла.

Также, с помощью ключа: `--storage.path` задаем путь до каталога, где будут храниться данные Alertmanager.

6. Создаем systemd сервис:

```
cat <<EOF > /usr/lib/systemd/system/alertmanager.service
[Unit]
Description=Prometheus Alertmanager
Documentation=https://github.com/alertmanager/alertmanager

[Service]
User=alertmanager
Group=alertmanager
Restart=always
EnvironmentFile=/etc/sysconfig/alertmanager
ExecStart=/usr/local/bin/alertmanager \$OPTIONS
ExecReload=/bin/kill -HUP \$MAINPID
TimeoutStopSec=20s
SendSIGKILL=no

[Install]
WantedBy=multi-user.target

EOF
```

С помощью директивы: `EnvironmentFile` все переменные из файла, указанного в этой директиве, добавляются в `env`. Это позволяет передавать список ключей для Alertmanager через переменную `$OPTIONS` без изменения службы.

7. Обновляем список служб systemd и запускаем Alertmanager:

```
systemctl daemon-reload
systemctl start alertmanager.service
systemctl enable alertmanager.service
```

8. Проверяем работу.

По **умолчанию** prometheus слушает порт **9093**:

```
curl -LI -XGET http://localhost:9093
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache, no-store, must-revalidate
Content-Length: 1314
Content-Type: text/html; charset=utf-8
Expires: 0
Last-Modified: Thu, 01 Jan 1970 00:00:01 GMT
Pragma: no-cache
Date: Fri, 15 Nov 2019 12:51:41 GMT
```

9. Добавляем отправку alert из Prometheus в Alertmanager.

Для этого добавим `targets` в конфигурационный файл Prometheus:

`/etc/prometheus/prometheus.yml`. Обратите внимание: секция `alerting` уже имеется в конфигурационном файле. Необходимо ее привести к виду:

```
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        - 'localhost:9093'
```

10. Проверяем корректность внесенных изменений:

```
promtool check config /etc/prometheus/prometheus.yml
```

11. Выполните `reload` для Prometheus, чтобы применить новые настройки:

```
sudo systemctl reload prometheus.service
```

**NB!** Для применения изменений в `prometheus.yml` достаточно выполнить `reload` для службы. Этот способ предпочтительней, так как в случае ошибки в конфигурационном файле просто не применяются изменения. Если же выполнить `restart`, то из-за ошибки весь Prometheus станет недоступен до устранения этой самой ошибки.

## Ключи запуска Alertmanager

Далее приведен список наиболее востребованных ключей Alertmanager:

```
--config.file
```

Default: `alertmanager.yml`

Имя конфигурационного файла.

```
--storage.path
```

Default: `data/`

Путь, куда сохраняются данные.

```
--data.retention
```

Default: `120h`

Как долго хранить данные.

```
--web.listen-address
```

Default: "**:9093**"

Адрес и порт, по которому доступны UI и метрики Alertmanager.

```
--cluster.listen-address
```

Default: "0.0.0.0:**9094**"

Адрес для HA кластера. Пустое значение для отключения HA mode.

```
--cluster.advertise-address
```

Default: -

IP для анонса в кластере.

```
--cluster.peer
```

Defalut: -

Адреса peer в HA кластере.

```
--log.level
```

Default: info

Данный ключ устанавливает уровень логирования. Возможные уровни логирования: debug, info, warn, error.

```
--log.format
```

Default: logfmt

Данный ключ устанавливает формат логов. Доступные форматы: logfmt и json.

**Полный список ключей можно просмотреть с помощью команды help:**

```
alertmanager --help
```

## Настройка Alertmanager

### Общие сведения

Настройка Alertmanager производится как с помощью ключей, так и с помощью конфигурационного файла. Формат – yaml. В конфигурационном файле настраивается вся обработкаalerтов. Конфигурационный файл может содержать 5 блоков. Далее рассмотрим каждый блок отдельно.

## Блок global

### resolve\_timeout

default: 5m Время, через которое алерт считается решенным, если в течение этого времени он не был обновлен.

**smtp\_from** default: – email адрес, который будет использован в качестве адреса отправителя. В блоке global объявляются значения по умолчанию, переопределить которые можно в блоке receiver.

### smtp\_smarthost

default:

Адрес SMTP сервера, который используется для отправки. Формат: ip:port . В блоке global объявляются значения по умолчанию, перепределить которые можно в блоке receiver.

### smtp\_require\_tls

default: true

Задает, использовать ли tls при подключении к SMTP. NB! Обратите внимание, что значение по умолчанию true и при использовании с локально установленным postfix без дополнительных настроек работать не будет.

Также в этой секции возможно настроить отправку уведомлений в: slack, hipchat, pagerduty и другие мессенджеры. Но для настройки получателей рекомендуется использовать webhook. Его настройка возможна только в разделе receivers.

## Блок route

В блоке route производится маршрутизация сообщений. В результате получается древовидная структура.

### receiver

Имя конфигурации, которая будет использована для отправки уведомлений.

### group\_by

Набор меток, по которым производится агрегация. Значение [...] отключает группировку.

### continue

default: false

Если значение false, обработка уведомления прекращается при первом совпадении. Если значение true, обработка продолжается и выбирается последний route, удовлетворяющий условиям.

### match

Список меток key: value, при точном совпадении с которыми выбирается этот маршрут.

### match\_re

Список меток key: regex, при совпадении с которыми выбирается этот маршрут.

### group\_wait

Default: 30s

Время задержки перед отправкой сообщений для группы. Позволяет дождаться поступления большего числа алертов и произвести эффективную группировку.

### group\_interval

default: 5m

Время задержки для отправки новых уведомлений, по которым первоначальное уведомление уже было отправлено.

### repeat\_interval

default: 4h

Время задержки для повторной отправки уведомления.

### routes

Дополнительные маршруты. Не являются обязательными.

Пример:

```
route:  
  receiver: 'default-receiver'  
  group_wait: 30s  
  group_interval: 5m  
  repeat_interval: 4h  
  group_by: [cluster, alertname]  
  routes:  
    - receiver: 'db-team'  
      group_wait: 10s  
      match_re:  
        service: mysql|postgres  
    - receiver: 'frontend-team'  
      group_by: [product, environment]  
      match:  
        team: frontend
```

В этом примере все уведомления labels service со значениями mysql или postgres будут отправлены db-team. Уведомления, имеющие label team со значением frontend, будут отправлены frontend-team. Остальные уведомления будут отправлены получателю по умолчанию.

## Блок receivers

Receiver – это именованная конфигурация одного или нескольких получателей уведомлений. На данный момент имеется интеграция с несколькими мессенджерами: slack, hipchat, pagerduty и др. Но для настройки получателей рекомендуется использовать webhook. В настройках указываются данные для отправки. Пример настройки email будет рассмотрен в следующем шаге.

## Блок inhibit\_rules

Блок inhibit\_rules позволяет отключить уведомления для алerts, у которых labels соответствуют набору labels, указанных в target\_match(re), и при условии, что уже есть уведомление, у которого набор labels соответствует набору labels, указанных в source\_match(re).

target\_match

Список labels, при совпадении с которым сообщение будет отключено.

target\_match\_re

Список labels, при совпадении с которым сообщение будет отключено. Для проверки используется регулярное выражение.

source\_match

Набор меток, для которых уже должен существовать алерт, чтобы правило подавления работало.

source\_match\_re

Набор меток, для которых уже должен существовать алерт, чтобы правило подавления работало. Для проверки используется регулярное выражение.

equal

Набор label, которые должны совпадать в alert и inhibit alert, чтобы подавление работало.

## Блок templates

В данном блоке задается список путей для файлов с пользовательскими шаблонами уведомлений. Допускается использование маски.

Пример:

```
templates:  
  - 'templates/*.tmpl'
```

# Настройка alertmanager - практика

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняется с root привилегиями.

## 1. Сохраним конфигурационный файл

Перед началом настройки, необходимо подтвердить пользовательское соглашение для email, с которого будут отправляться email уведомления. Для этого перейдите по [ссылке](#), авторизируйтесь и подтвердите соглашение. Для авторизации - имя пользователя `<user name>@edu-prom.slurm.io`, а пароль совпадает с паролем ssh. Данные можно посмотреть в [личном кабинете](#).

Выполним команду для создания правил алертинга. Перед выполнением необходимо поменять: `<email_1>` и `<email_2>` на реальные email (если нет двух разных email, можно указать один), а так же заменить `<user name>`, `<Password>`. Значения можете посмотреть в [личном кабинете](#).

```
cat <<EOF > /etc/alertmanager/alertmanager.yml
global:
  smtp_smarthost: smtp.yandex.ru:465
  smtp_from: '<user name>@edu-prom.slurm.io'
  smtp_auth_username: '<user name>@edu-prom.slurm.io'
  smtp_auth_password: '<Password>'

route:
  group_by: ['alertname', 'service']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 1h
  receiver: team-monitoring

  routes:
    - match:
        service: prom
        receiver: team-ops
    - match:
        severity: (warning|error|critical)
        receiver: team-monitoring

inhibit_rules:
  - source_match:
      severity: 'critical'
      alertname: PrometheusConfigurationReload
      target_match:
        severity: 'error'

receivers:
  - name: 'team-ops'
    email_configs:
      - to: '<email_1>'
        send_resolved: true
        require_tls: false
  - name: 'team-monitoring'
    email_configs:
      - to: '<email_2>'
        send_resolved: true
```

EOF

Данная конфигурация использует группировку по alertname и service. Все сообщения с `label service: prom` будут отправлены на `email_1`, все алерты с `label severity: critical, error` и `warning` будут отправлены на `email_2`. Если есть алерт с `label service: prom` и `label alertname: PrometheusConfigurationReload`, остальные алерты отправляться не будут.

**NB!** В реальной жизни routing будет намного сложнее, и для его визуализации можно воспользоваться [сайтом](#).

2. Выполните reload для Alertmanager, чтобы применить новые настройки:

```
systemctl reload alertmanager.service
```

3. Тестирование.

- Исправляем конфиг Prometheus.

Добавляем в начало конфигурационного файла Prometheus: 1. И выполняем:

```
systemctl reload prometheus.service
```

Это приведет к отправке алерта об ошибке в конфигурационном файле Prometheus.

- Останавливаем node exporter.

Останавливаем node exporter на monitoring и server1, выполнив команду.

```
systemctl stop node_exporter.service
```

Теперь необходимо дождаться, пока отработает alert rule. Для проверки перейдите:

`http://<monitoring_IP>:9090/alerts`. Должно быть активно два алерта.

## Alerts

Show annotations

/etc/prometheus/rules\_alert.yml > NodeExporterGroup

**ExporterDown** (2 active)

**HighCpuLoad** (0 active)

**SystemServiceCrashed** (0 active)

/etc/prometheus/rules\_alert.yml > PrometheusGroup

**PrometheusConfigurationReload** (1 active)

При этом новых алертов приходить не должно, так как работает подавление.

- Исправляем конфигурационный файл: `/etc/prometheus/prometheus.yml`, удаляем 1. И выполняем:

```
systemctl reload prometheus.service
```

Теперь должно прийти еще 2 уведомления: первое – об исправлении с конфигурацией Prometheus, второе – о недоступности двух exporters.

## Настройка Alertmanager HA

1. Устанавливаем второй alertmanager.

Подключаемся с monitoring сервера к server1 из-под root учетки, адрес можно посмотреть в [личном кабинете](#):

```
ssh root@<адрес сервера server1>
```

и устанавливаем alertmanager по инструкции из шага 2.

2. Настраиваем кластер.

2.1 На server1 пропишем IP-адрес alertmanager установленного на monitoring, адрес можно посмотреть в [личном кабинете](#):

```
OPTIONS="--config.file=/etc/alertmanager/alertmanager.yml --  
storage.path=/var/lib/alertmanager/ --cluster.peer <monitoring IP>:9094"
```

**NB!** Обратите внимание: ip-адрес указывается вместе с портом. При этом порт – **9094**. Данный порт используется для обмена данными по протоколу **gossip**.

2.2 Выполните restart для Alertmanager, чтобы применить новые настройки:

```
systemctl restart alertmanager.service
```

2.3 Возвращаемся на monitoring сервер и пропишем IP-адрес alertmanager установленного на server1, адрес можно посмотреть в [личном кабинете](#):

```
OPTIONS="--config.file=/etc/alertmanager/alertmanager.yml --  
storage.path=/var/lib/alertmanager/ --cluster.peer <server1 IP>:9094"
```

**NB!** Обратите внимание: ip-адрес указывается вместе с портом. При этом порт – 9094. Данный порт используется для обмена данными по протоколу gosip.

2.4 Выполните restart для Alertmanager, чтобы применить новые настройки:

```
systemctl restart alertmanager.service
```

3. Проверяем работу.

Открываем в браузере: `http://<адрес сервера monitoring>:9093#/status`. Результат должен быть примерно таким:

# Cluster Status

Name:	01DSY ZX4A1X219JTCPSZG55ZAD
Status:	ready
Peers:	<ul style="list-style-type: none"><li>• Name: 01DSY ZX4A1X219JTCPSZG55ZAD Address: 192.168.0.4:9094</li><li>• Name: 01DSY ZY8GAETGDRKECTY1PWQ6R Address: 192.168.0.12:9094</li></ul>

## Визуализация данных

### Grafana

После настройки сбора данных нужна визуализация.

В UI Prometheus есть примитивные графики, но нужен отдельный инструмент. Сначала использовался PromDash, но потом был разработан плагин для Grafana и он стал практически стандартом.

Также существуют готовые дашборды под все возможные экспортёры.

### Установка Grafana

Практика выполняется на monitoring сервере, доступ к нему осуществляется по IP адресу и выполняется с root привилегиями.

В отличие от Prometheus, у grafana есть [репозитарий для большинства современных ОС](#).

1. Добавляем репозитарий с grafana:

```
cat <<EOF > /etc/yum.repos.d/grafana.repo
[grafana]
name=grafana
baseurl=https://packages.grafana.com/oss/rpm
repo_gpgcheck=1
enabled=1
gpgcheck=1
gpgkey=https://packages.grafana.com/gpg.key
sslverify=1
sslcacert=/etc/pki/tls/certs/ca-bundle.crt
EOF
```

2. Устанавливаем Grafana:

```
yum install -y grafana
```

3. Запускаем Grafana:

```
systemctl daemon-reload  
systemctl start grafana-server  
systemctl enable grafana-server.service
```

4. Проверяем работу.

По **умолчанию**, grafana слушает порт **3000**:

```
curl -IL http://localhost:3000/login
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK  
Cache-Control: no-cache  
Content-Type: text/html; charset=UTF-8  
Expires: -1  
Pragma: no-cache  
X-Frame-Options: deny  
Date: Mon, 18 Nov 2019 10:46:02 GMT
```

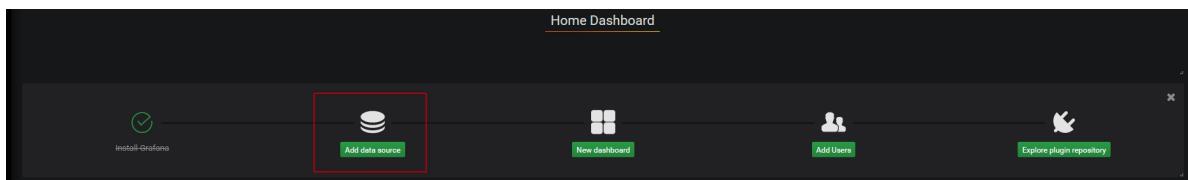
## Настройка источника данных в Grafana

1. Подключение к grafana.

Открываем в браузере: `http://<monitoring IP>:3000`. Имя пользователя и пароль по умолчанию admin/admin. Рекомендуется установить новый пароль, идентичный паролю подключения к стендам.

2. Добавляем Prometheus в качестве источника данных.

На главном экране выбираем Add data source:



Также это можно сделать, перейдя: Settings -> Data Source.

В качестве источника данных выбираем Prometheus:

The screenshot shows the configuration page for a data source named "Prometheus". At the top, there is a "Name" field set to "Prometheus", a "Default" toggle switch which is turned on, and a "Help" button. Below this, the "HTTP" section is visible, containing fields for "URL" (set to "http://localhost:9090"), "Access" (set to "Server (Default)"), and "Whitelisted Cookies" (with an "Add Name" button). The "Auth" section follows, featuring options for "Basic Auth", "TLS Client Auth", "Skip TLS Verify", and "Forward OAuth Identity", each with a checkbox. Further down are settings for "Scrape interval" (set to "15s"), "Query timeout" (set to "60s"), and "HTTP Method" (set to "GET").

**Name** – имя data source, должно быть уникальным.

**URL** – адрес, где располагается prometheus. Так как grafana установлена на том же сервере, что и Prom, указываем: <http://localhost:9090>

**Access** – способ подключения к Prometheus. Server – запрос из браузера отправляется в grafana, grafana производит подключение к Prometheus. Browser – в этом случае запросы в Prometheus отправляются напрямую из браузера. Метод Browser менее предпочтителен, так как приводит к необходимости открывать прямой доступ к Prometheus.

**Auth** – задаются учетные данные для подключения к Prom.

**Scrape interval** – рекомендуется выставить равным глобальному значению scrape, установленному в Prometheus.

**Query timeout** – задает максимальное время выполнения запроса к Prometheus.

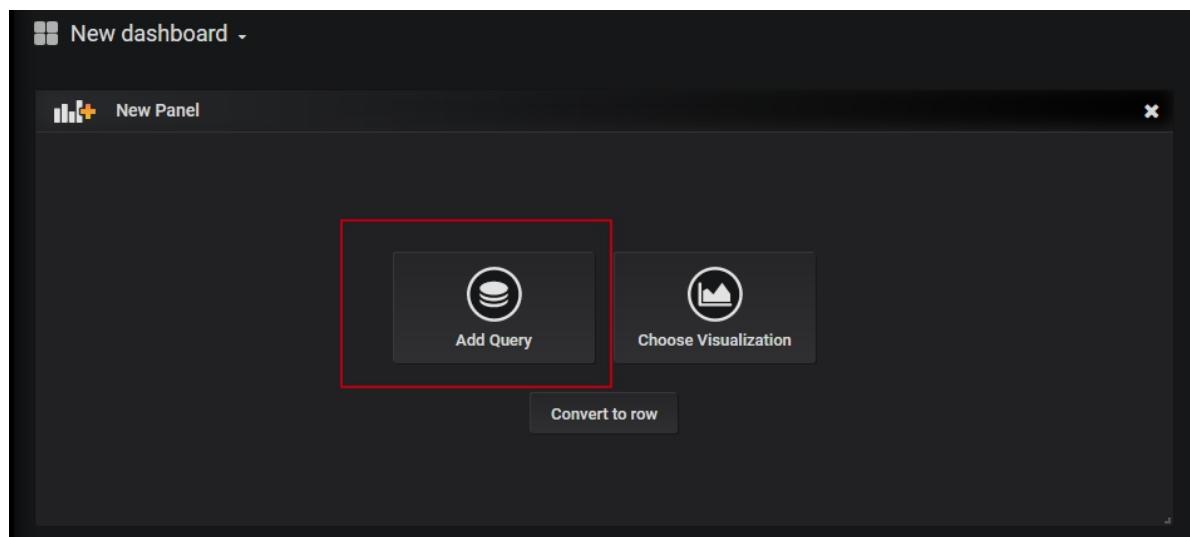
**HTTP Method** – метод для отправки запросов в Prom. Post запросы возможны, начиная с версии: 2.1.0

Сохраняем изменения.

# Настройка dashboard в Grafana

1. Создаем dashboard в grafana.

Чтобы создать новую dashboard, переходим: Create -> Dashboard и выбираем Add Query:



2. Добавляем метрики для визуализации.

Добавляем запрос, который будет использоваться для выборки данных:

```
rate(prometheus_http_requests_total{{job="prometheus"}}[5m])
```

В качестве Legend указываем, какие labels будут отображаться. По умолчанию отображаются все:

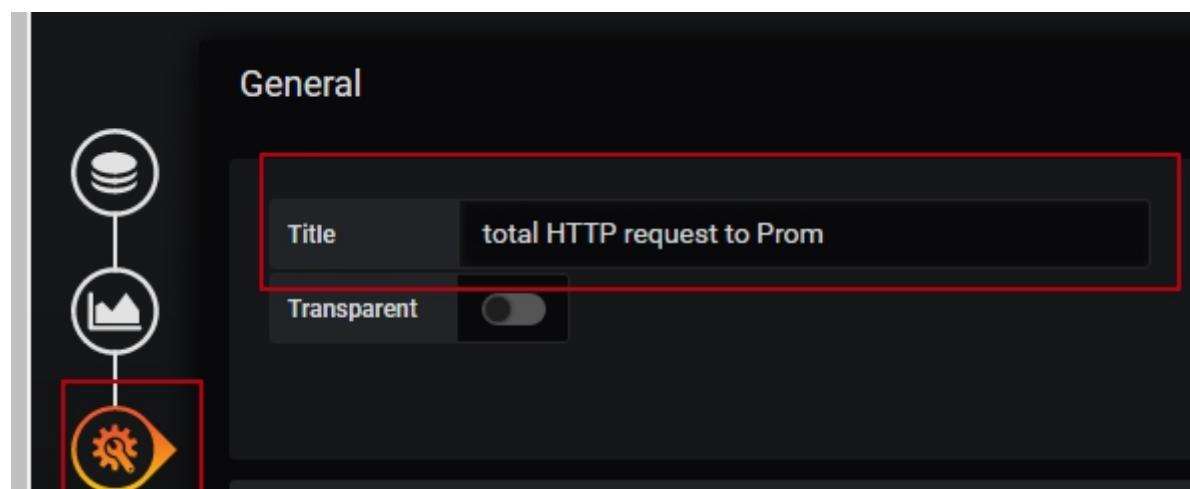
```
{{code}} - {{handler}}
```

Результат должен выглядеть так:



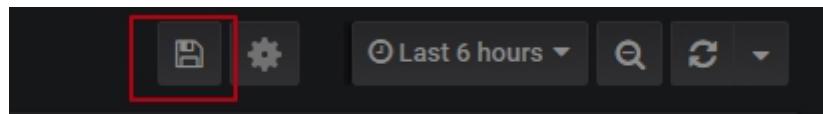
3. Задаем имя панели.

Для этого переходим в раздел "General" и вводим Title: total HTTP request to Prom:

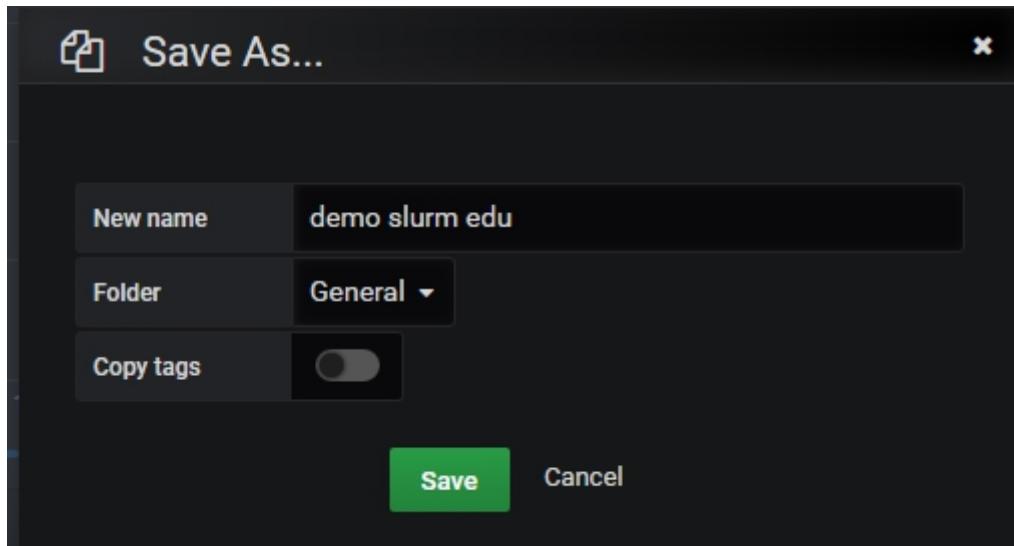


4. Сохраняем dashboard.

Нажимаем "Save dashboard":

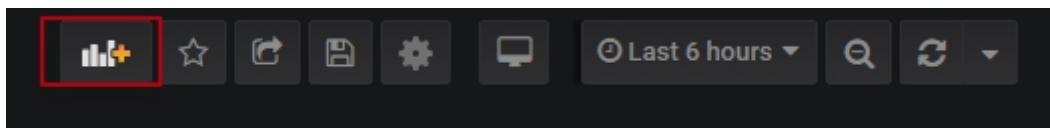


Задаем имя для dashboard: demo slurm edu:



5. Добавляем еще один график.

Для этого выбираем "add panel":



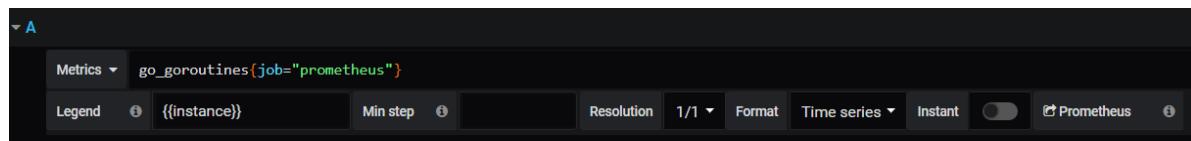
Добавляем запрос, который будет использоваться для выборки данных:

```
go_goroutines{job="prometheus"}
```

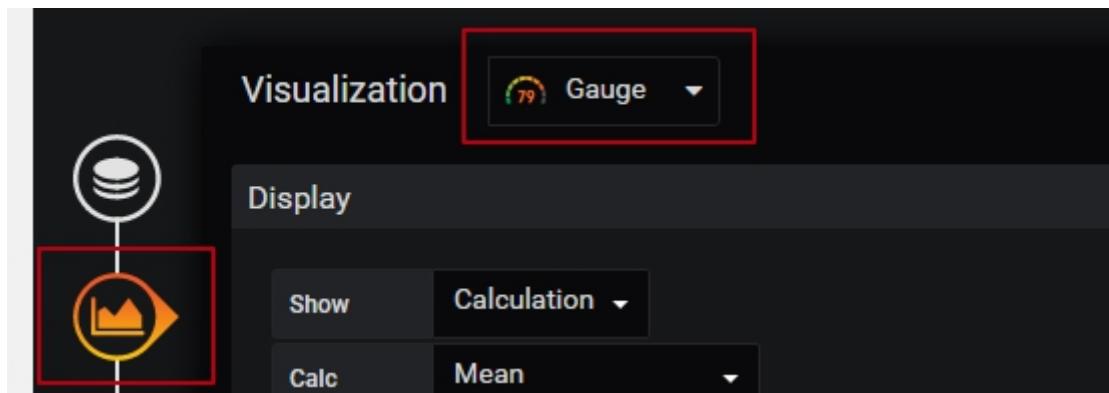
В качестве Legend указываем, какие labels будут отображаться. По умолчанию отображаются все:

```
{{instance}}
```

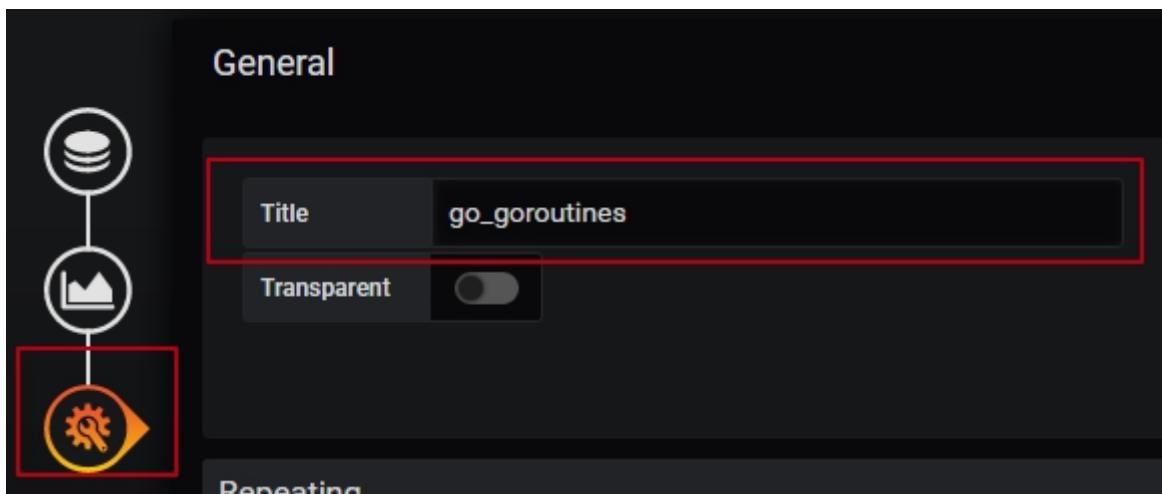
Результат должен выглядеть так:



Далее переходим в "Vizualization" и выбираем "Gauge":

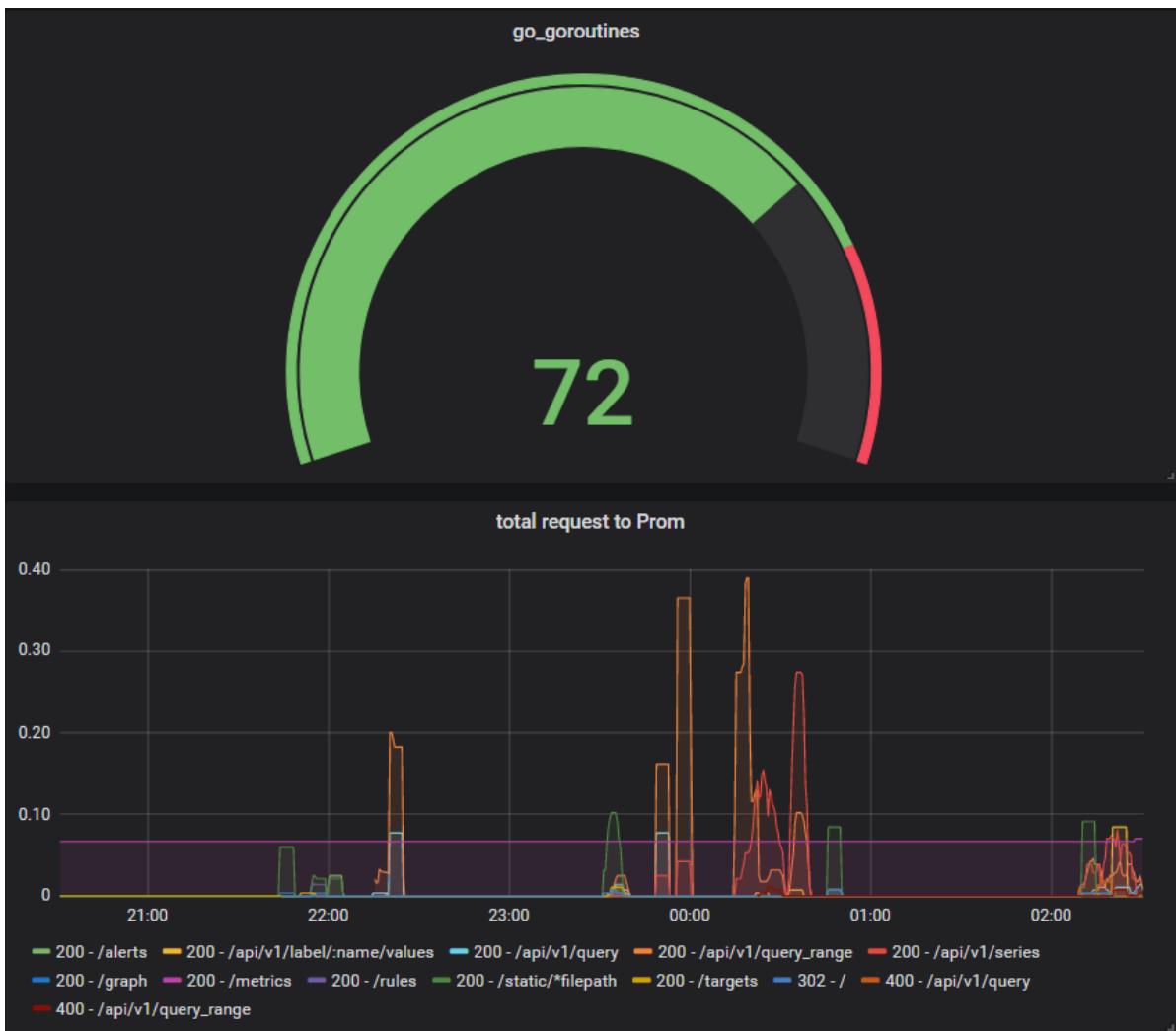


Задаем имя панели. Для этого переходим в раздел "General" и вводим Title: go\_goroutines:



Сохраняем изменения.

В результате у вас должна получиться Dashboard с двумя графиками:



## Импорт готовых dashboard для Grafana

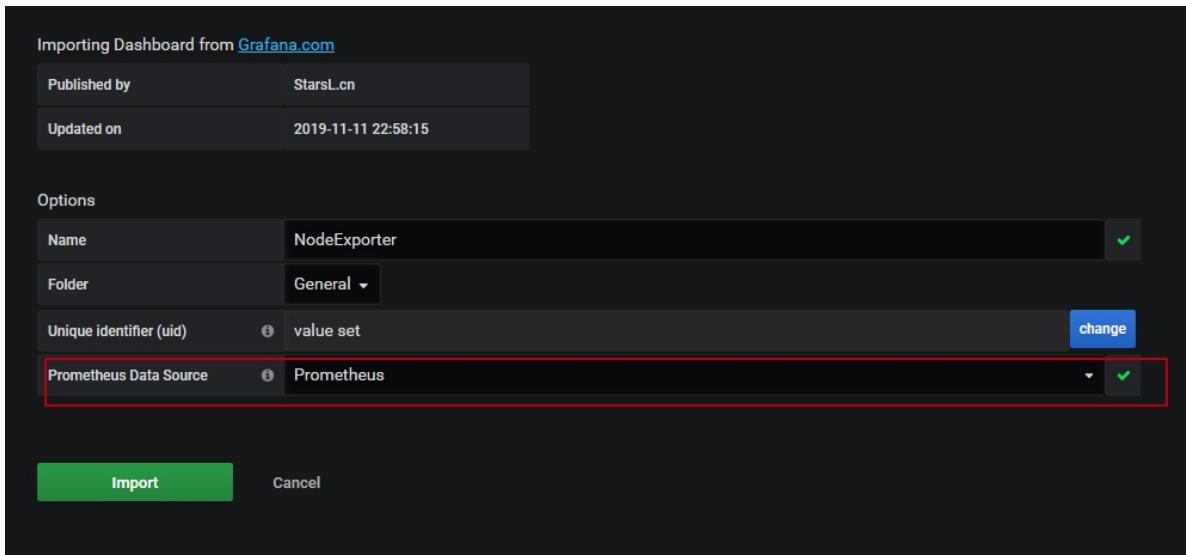
Для большинства exporters уже есть готовые dashboards. Полный список можно посмотреть на официальном сайте [Grafana](#). Далее рассмотрим пример добавления dashboard для node\_exporter.

1. Найдем подходящий dashboard.

Для node exporter есть несколько готовых dashboard. Мы будем использовать с ID: [11074](#)

2. Импорт шаблона dashboard.

Откроем раздел "Create" -> "Import". На открывшейся странице указываем ID. Далее указываем источник данных и нажимаем "Import".



3. Проверяем работу.

После импорта grafana перенаправит нас на страницу с Dashboard.

## Advanced usage of Prometheus

### High Availability

Мониторинг - ключевой элемент любого прод решения. От того, насколько хорошо работает мониторинг и работает ли вообще - зависит, вовремя ли мы узнаем о проблемах и узнаем ли вообще. В Прометея для этого есть несколько подходов.

1. Самое простое - дублирование компонентов. Дублировать экспортеры смысла нет, если он недоступен - мы об этом узнаем. Так как Прометей сам забирает данные с экспортеров, то с его дублированием проблем нет, самый простой вариант - поставить рядом еще один Прометей на те же экспортеры. Единственное, что придется обеспечить - идентичность конфигурации.
2. Недостаточно просто собрать данные, надо их еще и получить. Для этого дублируем еще и Alertmanager. Чтобы не было дублирования алертов - Alertmanager надо настраивать в режиме HA, это позволит им общаться между собой и не дублировать алерты. Если настроить группировку и задержку уведомлений перед отправкой на Alertmanager - то последний уберет дубли.
3. Графану дублировать необязательно. Но между Графаной и Прометеями обычно добавляется HAProxy в режиме hot-standby.

Минусы у такого подхода - излишняя избыточность собираемых данных.

### Настройка Prometheus в режиме HA

**Внимание! Пункты 1-4 необходимо выполнить на server1 и server2.**

1. Установка Prometheus.

Необходимо установить Prometheus на серверы **server1** и **server2**. Подключиться к ним можно с сервера monitoring, по IP-адресу из под учетной записи root.

Установку надо производить по инструкции из главы 3.1.

2. Настройка Prometheus.

После установки на каждом из Prometheus серверов необходимо заменить конфигурационный файл. Перед выполнением команды замените <monitoring\_ip>, <server1\_ip> и <server2\_ip> на реальные ip, которые доступны в ЛК.

```
cat <<EOF > /etc/prometheus/prometheus.yml
global:
  scrape_interval:      15s
  evaluation_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
      - targets:
        - alertmanager:9093

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node_exporter'
    static_configs:
      - targets:
          - <monitoring_ip>:9100
          - <server1_ip>:9100
          - <server2_ip>:9100
EOF
```

3. Перезапускаем Prometheus, чтобы применить настройки:

```
systemctl restart prometheus.service
```

4. Проверяем работу.

По **умолчанию** Prometheus слушает порт **9090**:

```
curl -XGET -IL http://localhost:9090/graph
```

Ответ должен быть примерно таким:

```
HTTP/1.1 200 OK
Date: Thu, 14 Nov 2019 07:05:12 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
```

## Настройка HA Proxy

Для доступа из Grafana к Prometheus необходимо установить HAProxy, для повышения отказоустойчивости. Установка производится на сервер **monitoring**.

1. Устанавливаем HAProxy:

```
yum install -y haproxy
```

2. Настраиваем HAProxy.

Перед выполнением команды необходимо заменить <server1\_IP> и <server2\_IP> на реальные IP серверов:

```
cat <<EOF > /etc/haproxy/haproxy.cfg
#HA Proxy Config
global
  daemon
  maxconn 256
defaults
  mode http
  timeout connect 5000ms
  timeout client 50000ms
  timeout server 50000ms
listen stats
  bind *:9999
  stats enable
  stats hide-version
  stats uri /stats
  stats auth admin:admin@123
frontend prom
  bind *:80
  use_backend prom
backend prom
  server prom1 <server1_IP>:9090 check
  server prom2 <server2_IP>:9090 check backup
EOF
```

Данная конфигурация позволяет проксировать запросы с 80 порта на один из Prom серверов.

Запускаем HAProxy

```
systemctl start haproxy.service
```

3. Настройка Grafana.

3.1 Добавляем новый data source.

Добавление data source производится аналогично тому, как мы это делали в предыдущей главе, за исключением адреса сервера. В качестве URL указываем: <http://localhost:80>. В качестве Name указываем: Prometheus HA.

Settings Dashboards

Name: Prometheus HA Default

### HTTP

URL	http://localhost:80	<a href="#">i</a>
Access	Server (Default)	<a href="#">Help ▾</a>
Whitelisted Cookies	Add Name	<a href="#">i</a>

### Auth

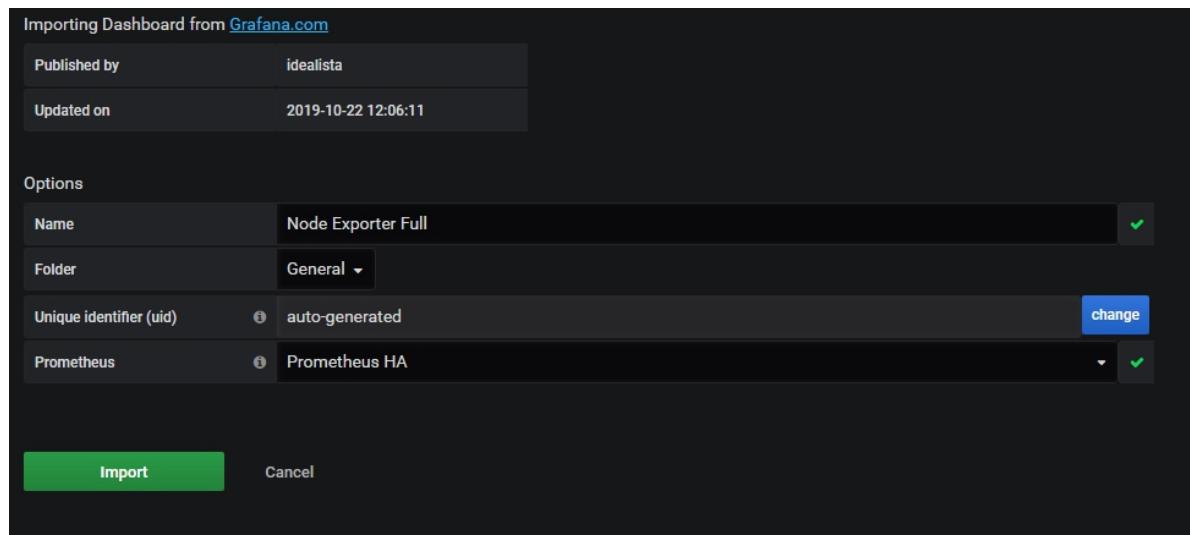
Basic Auth	<input type="checkbox"/>	With Credentials	<input type="checkbox"/>
TLS Client Auth	<input type="checkbox"/>	With CA Cert	<input type="checkbox"/>
Skip TLS Verify	<input type="checkbox"/>		
Forward OAuth Identity	<a href="#">i</a>	<input type="checkbox"/>	

Scrape interval	15s	<a href="#">i</a>
Query timeout	60s	<a href="#">i</a>
HTTP Method	GET	<a href="#">i</a>

[Save & Test](#) [Delete](#) [Back](#)

3.2 Добавляем dashboard.

Добавление dashboard производим аналогично тому, как это делали в прошлой главе, но возьмем dashboard с ID: [1860](#). **Важно!** В качестве Prometheus сервера выбрать "Prometheus HA":



#### 4. Проверка результатов работы.

Для проверки необходимо выключить Prometheus поочередно на server1 и server2 и убедиться, что графики продолжают отображаться:

```
systemctl stop prometheus.service
```

## Federation

Прометей может выступать источником данных для других Прометеев - эта конструкция называется федерацией. Можно получать как все данные, так и фильтровать их при помощи RegExp.

Федерация подходит, если разные проекты в одном ЦОДе и обслуживаются разными Прометеями.

## Настройка Prometheus Federation

### 1. Настройка Prometheus.

Настроим Prom сервера на серверах: server1 и server2 на сбор данных только с локального node\_exporter.

#### 1.1 Обновляем конфигурационный файл.

Обратите внимание, что до выполнения команды `<server_ip>` надо заменить на локальный IP того сервера, где применяется конфиг. `<monitoring_IP>` необходимо заменить на реальный ip monitoring сервера.

```
cat <<EOF > /etc/prometheus/prometheus.yml
global:
  scrape_interval:      15s
  evaluation_interval: 15s

  alerting:
    alertmanagers:
      - static_configs:
          - targets:
              - <monitoring_IP>:9093
```

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node_exporter'
    static_configs:
      - targets:
          - <server_ip>:9100
EOF
```

1.2 Перезапускаем Prometheus, чтобы применить настройки:

```
systemctl restart prometheus.service
```

## 2. Настройка Federation.

На сервере monitoring заменяем конфигурацию:

```
cat <<EOF > /etc/prometheus/prometheus.yml
global:
  scrape_interval:      15s
  evaluation_interval: 15s

  alerting:
    alertmanagers:
      - static_configs:
          - targets:
              - localhost:9093

  scrape_configs:
    - job_name: 'federate'
      honor_labels: true
      metrics_path: '/federate'

    params:
      'match[]':
        - '{job=~"node.+"}'

  static_configs:
    - targets:
        - '<server1_IP>:9090'
        - '<server1_IP>:9090'
EOF
```

Специальный endpoint `/federate` позволяет получить значения все метрики одним запросом.

**NB!** Обратите внимание: `honor_labels: true` позволяет выставить приоритет `labels` источника, что позволяет сохранять значения для `labels: job` и `instance`.

**NB!** Параметр `match` является обязательным. С его помощью задается, для каких метрик будет производиться scraping. Фильтрация производится на основание `labels`.

Перезапускаем Prometheus, чтобы применить настройки:

```
systemctl restart prometheus.service
```

### 3. Проверка работы.

Открываем в браузере: `http://<monitoring>/targets`. Должно отображаться два сервера.

federate (2/2 up) <small>show less</small>		State	Labels	Last Scrape	Scrape Duration	Error
<code>http://192.168.0.12:9090/federate</code>	<code>match[] = "job=~\"node.+\""</code>	UP	<code>instance="192.168.0.12:9090" job="federate"</code>	14.507s ago	10.02ms	
<code>http://192.168.0.7:9090/federate</code>	<code>match[] = "job=~\"node.+\""</code>	UP	<code>instance="192.168.0.7:9090" job="federate"</code>	8.541s ago	11.9ms	

При выполнении запроса: `node_load1`, через 15 минут после изменения настроек, должно вернуться 2 вектора:

The screenshot shows the Grafana interface with a query editor. The query text is `node_load1`. Below the query, there are buttons for **Execute** and **- insert metric at cursor -**. The results section shows a table with two rows of data. The columns are **Element** and **Value**. The first row is `node_load1{instance="192.168.0.12:9100",job="node_exporter"}` with a value of 0. The second row is `node_load1{instance="192.168.0.7:9100",job="node_exporter"}` with a value of 0. The top right of the results area displays metrics: Load time: 22ms, Resolution: 14s, Total time series: 2. Navigation buttons for **Graph** and **Console** are also visible.

Element	Value
<code>node_load1{instance="192.168.0.12:9100",job="node_exporter"}</code>	0
<code>node_load1{instance="192.168.0.7:9100",job="node_exporter"}</code>	0

## Remote read/write

Одно из узких мест любых систем мониторинга - дисковое пространство.

В Прометея есть возможность использования удаленной системы хранения данных, при этом разделяются процессы чтения и записи данных - в некоторые системы можно только писать, из некоторых - только читать.

Прометей отправляет данные в адаптер и уже тот отвечает за сохранение их на удаленном хранилище.

Можно расширять список поддерживаемых удаленных хранилищ без изменения самого Прометея.

Для чтения Прометей отправляет, какие данные за какие промежутки времени надо получить.

Важно то, что Прометей читает все данные сразу, поэтому надо просчитывать место на самом Прометеи, если запрашиваются данные за длительный период.

### Настройка Remote read/write

#### 1. Настройка Remote read.

Ниже приведен список параметров и их назначение:

`url`

URL, куда будут отправляться запросы для удаленного чтения.

`required_matchers`

Список `labels`, для которых будет производиться удаленное чтение. Не является обязательным.

#### `remote_timeout`

Default: 1m

Максимальное время ожидания для операции удаленного чтения.

#### `read_recent`

Default: false

Выполнять ли удаленный запрос для временного интервала, который имеется в локальном хранилище.

#### `basic_auth`

- `username` – имя пользователя, которое используется для авторизации на проверяемом сайте.
- `password` – пароль, который используется для авторизации на проверяемом сайте.

#### `bearer_token`

Токен для bearer авторизации.

#### `bearer_token_file`

Файл, который содержит токен для bearer авторизации.

#### `proxy_url`

Адрес proxy сервера, если проверку необходимо выполнить через proxy сервер.

#### `tls_config`

- `insecure_skip_verify` – проверять ли валидность сертификата. Значение по умолчанию: `false`.
- `ca_file` – путь к файлу с корневыми сертификатами.
- `cert_file` – путь к файлу с клиентским сертификатом.
- `key_file` – путь к файлу с клиентским ключом.
- `server_name` – строка для проверки имени сервера.

## 2. Настройка Remote write.

#### `url`

URL куда будут отправляться запросы для удаленного чтения.

#### `remote_timeout`

Default: 1m

Максимальное время ожидания для операции удаленного чтения.

#### write\_relabel\_configs

Настройки для изменения labels. Задаются аналогично relabel\_configs.

#### basic\_auth

- **username** – имя пользователя, которое используется для авторизации на проверяемом сайте.
- **password** – пароль, который используется для авторизации на проверяемом сайте.

#### bearer\_token

Токен для bearer авторизации.

#### bearer\_token\_file

Файл, который содержит токен для bearer авторизации.

#### proxy\_url

Адрес proxy сервера, если проверку необходимо выполнить через proxy сервер.

#### tls\_config

- **insecure\_skip\_verify** - проверять ли валидность сертификата. Значение по умолчанию: false.
- **ca\_file** – путь к файлу с корневыми сертификатами.
- **cert\_file** – путь к файлу с клиентским сертификатом.
- **key\_file** – путь к файлу с клиентским ключом.
- **server\_name** – строка для проверки имени сервера.

#### queue\_config

В данном блоке настроек производится настройка очереди для удаленной записи. Доступны следующие параметры:

#### capacity

default: 500

Количество сэмплов для буферизации.

#### max\_shards

default: 1000

Максимальное количество шардов.

#### min\_shards

default: 1

Минимальное количество шардов.

`max_samples_per_send`

default: 100

Максимальное количество сэмплов для отправки.

`batch_send_deadline`

default: 5s

Максимальное время, в течение которого сэмпл будет ждать в буфере.

`min_backoff`

default: 30ms

Перерыв между попытками повторной отправки данных. Удваивается после каждой неудачной попытки.

`max_backoff`

default: 100ms

Максимальная задержка между попытками отправки данных.

### 3. Поддерживаемые удаленные хранилища:

- [AppOptics](#): write
- [Azure Data Explorer](#): read and write
- [Chronix](#): write
- [Cortex](#): read and write
- [CrateDB](#): read and write
- [Elasticsearch](#): write
- [Gnocchi](#): write
- [Graphite](#): write
- [InfluxDB](#): read and write
- [IRONdb](#): read and write
- [Kafka](#): write
- [M3DB](#): read and write
- [OpenTSDB](#): write
- [PostgreSQL/TimescaleDB](#): read and write
- [SignalFx](#): write
- [Splunk](#): read and write
- [TiKV](#): read and write
- [Thanos](#): write
- [VictoriaMetrics](#): write
- [Wavefront](#): write

## Thanos

Прометей изначально делался для оперативного мониторинга, в него не были заложены инструменты для длительного хранения метрик.

Решение данной проблемы - использование **Thanos**.

Состоит из нескольких компонентов:

- sidecar, который ставится рядом с каждым экземпляром Прометея (в Kubernetes). Его задача - копировать метрики на удаленные хранилища данных (Google Cloud Storage, S3, Openstack SWIFT, Azure Blob Storage)
- Thanos Query - реализует API, аналогичное Прометеевскому. Он используется при исполнении запросов от Графаны.
- Thanos Push Gateway - адаптер для универсального доступа к удаленному хранилищу.
- Thanos Compact - дедупликация данных, полученных с Prometheus HA.
- Thanos Rules - правила для алертинга, учитывающие данные с удаленных хранилищ.

## HTTP API

---

### Общие сведения для запросов API

Текущая версия API - v1 и доступна по /api/v1/.

Формат ответа – json.

Коды ответа:

2xx – успешный запрос

400 – когда отсутствуют необходимые параметры или они не верные

422 – запрос не может быть выполнен

503 – время ожидания ответа превышено.

Метки времени могут быть переданы либо в формате [RFC3339](#), либо в формате Unix timestamp.

Метод запроса может быть как GET, так и POST. Если используется GET, то параметры запроса добавляются к url, а если POST, то параметры запроса передаются как body.

### Запросы мгновенного вектора

**Endpoint:** /api/v1/query

**Method:** GET | POST

**Параметры запроса:**

- query – строка запроса PromQL.
- time – временная метка, за которую надо выбрать данные. Если параметр не задан, берется текущее время сервера.
- timeout – максимальное время выполнения запроса. Параметр не является обязательным. Если параметр не задан, используется значение query.timeout.

Во всех примерах для выполнения запросов будет использоваться утилита curl.

Пример запроса с использованием метода GET. Запрос ip, время: 19.11.2019 09:10:51. Перед выполнением запроса измените время на текущее. Время указывается в UTC:

```
curl 'http://localhost:9090/api/v1/query?query=up&time=2019-11-19T09:10:51.781Z'
```

Ответ должен быть примерно таким:

```
{  
    "status": "success",  
    "data": {  
        "resultType": "vector",  
        "result": [  
            {  
                "metric": {  
                    "__name__": "up",  
                    "instance": "192.168.0.12:9100",  
                    "job": "openstack"  
                },  
                "value": [  
                    1574154651.781,  
                    "1"  
                ]  
            },  
            {  
                "metric": {  
                    "__name__": "up",  
                    "instance": "192.168.0.4:9100",  
                    "job": "openstack"  
                },  
                "value": [  
                    1574154651.781,  
                    "1"  
                ]  
            },  
            {  
                "metric": {  
                    "__name__": "up",  
                    "instance": "192.168.0.7:9100",  
                    "job": "openstack"  
                },  
                "value": [  
                    1574154651.781,  
                    "1"  
                ]  
            },  
            {  
                "metric": {  
                    "__name__": "up",  
                    "instance": "localhost:9090",  
                    "job": "prometheus"  
                },  
                "value": [  
                    1574154651.781,  
                    "1"  
                ]  
            }  
        ]  
    }  
}
```

Пример аналогичного запроса методом POST:

```
curl -XPOST -H 'Content-Type: application/x-www-form-urlencoded' -d "query=up" -d "time=2019-11-19T09:10:51.781Z" http://localhost:9090/api/v1/query
```

NB! Content-Type используется: application/x-www-form-urlencoded.

Результат ответа должен быть аналогичным предыдущему запросу.

## Запрос вектора за период времени

**Endpoint:** /api/v1/query\_range

**Method:** GET | POST

**Параметры запроса:**

- query – строка запроса PromQL.
- start – временная метка начала вектора.
- end – временная метка окончания вектора.
- step – шаг выборки данных; допустимый формат [0-9]+[smhdwy], например 5s, либо число float(секунды).
- timeout – максимальное время выполнения запроса. Необязательный параметр. Если параметр не задан, используется значение query.timeout.

**Пример.** Запрос: up, за период с 19.11.2019 10:00.50 по 19.11.2019 10:15.50, с шагом в 30 секунд:

```
curl 'http://localhost:9090/api/v1/query_range?query=up&start=2019-11-19T09:10:51.781Z&end=2019-11-19T09:15:51.781Z&step=30s'
```

Ответ должен быть примерно таким:

```
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "__name__": "up",
          "instance": "192.168.0.12:9100",
          "job": "openstack"
        },
        "values": [
          [
            1574154651.781,
            "1"
          ],
          [
            1574154681.781,
            "1"
          ],
          [
            1574154711.781,
            "1"
          ],
          [
            1574154741.781,
            "1"
          ]
        ]
      }
    ]
  }
}
```

```
        "1"
    ],
    [
        1574154771.781,
        "1"
    ],
    [
        1574154801.781,
        "1"
    ],
    [
        1574154831.781,
        "1"
    ],
    [
        1574154861.781,
        "1"
    ],
    [
        1574154891.781,
        "1"
    ],
    [
        1574154921.781,
        "1"
    ],
    [
        1574154951.781,
        "1"
    ]
]
},
{
    "metric": {
        "__name__": "up",
        "instance": "192.168.0.4:9100",
        "job": "openstack"
    },
    "values": [
        [
            1574154651.781,
            "1"
        ],
        [
            1574154681.781,
            "1"
        ],
        [
            1574154711.781,
            "1"
        ],
        [
            1574154741.781,
            "1"
        ],
        [
            1574154771.781,
            "1"
        ]
    ]
}
]
```

```
],
[
  1574154801.781,
  "1"
],
[
  [
    1574154831.781,
    "1"
  ],
  [
    [
      1574154861.781,
      "1"
    ],
    [
      [
        1574154891.781,
        "1"
      ],
      [
        [
          1574154921.781,
          "1"
        ],
        [
          [
            1574154951.781,
            "1"
          ]
        ]
      ]
    ],
    {
      "metric": {
        "__name__": "up",
        "instance": "192.168.0.7:9100",
        "job": "openstack"
      },
      "values": [
        [
          [
            1574154651.781,
            "1"
          ],
          [
            [
              1574154681.781,
              "1"
            ],
            [
              [
                1574154711.781,
                "1"
              ],
              [
                [
                  1574154741.781,
                  "1"
                ],
                [
                  [
                    1574154771.781,
                    "1"
                  ],
                  [
                    [
                      1574154801.781,
                      "1"
                    ],
                    [
                      [
                        1574154831.781,
                        "1"
                      ],
                      [
                        [
                          1574154861.781,
                          "1"
                        ],
                        [
                          [
                            1574154891.781,
                            "1"
                          ],
                          [
                            [
                              1574154921.781,
                              "1"
                            ],
                            [
                              [
                                1574154951.781,
                                "1"
                              ]
                            ]
                          ]
                        ]
                      ]
                    ]
                  ]
                ]
              ]
            ]
          ]
        ]
      ]
    }
  ]
]
```

```
[  
    1574154831.781,  
    "1"  
,  
[  
    1574154861.781,  
    "1"  
,  
[  
    1574154891.781,  
    "1"  
,  
[  
    1574154921.781,  
    "1"  
,  
[  
    1574154951.781,  
    "1"  
]  
]  
]  
,  
{  
    "metric":{  
        "__name__":"up",  
        "instance":"localhost:9090",  
        "job":"prometheus"  
    },  
    "values": [  
        [  
            1574154651.781,  
            "1"  
,  
        [  
            1574154681.781,  
            "1"  
,  
        [  
            1574154711.781,  
            "1"  
,  
        [  
            1574154741.781,  
            "1"  
,  
        [  
            1574154771.781,  
            "1"  
,  
        [  
            1574154801.781,  
            "1"  
,  
        [  
            1574154831.781,  
            "1"  
,  
        [  
            1574154861.781,  
            "1"  
,  
        [  
            1574154891.781,  
            "1"  
,  
        [  
            1574154921.781,  
            "1"  
,  
        [  
            1574154951.781,  
            "1"  
        ]  
    ]  
]
```

```

        1574154861.781,
        "1"
    ],
    [
        1574154891.781,
        "1"
    ],
    [
        1574154921.781,
        "1"
    ],
    [
        1574154951.781,
        "1"
    ]
]
}
]
}
}

```

## Поиск временных рядов на основании labels

**Endpoint:** /api/v1/series

**Method:** GET | POST

**Параметры запроса:**

- match[] – задает список меток, соответствие которым учитывается при поиске. Необходимо задать хотя бы одно условие.
- start – временная метка начала вектора.
- end – временная метка окончания вектора.

**Пример.** В результате выполнения запроса мы получим данные по всем временным рядам, которые имеют имя `node_disk_write_time_seconds_total` и label device со значением sda:

```

curl -XPOST -H 'Content-Type: application/x-www-form-urlencoded' -d
'match[]=node_disk_write_time_seconds_total{device="sda"}'
http://localhost:9090/api/v1/series

```

Ответ будет примерно таким:

```

{
  "status": "success",
  "data": [
    {
      "__name__": "node_disk_write_time_seconds_total",
      "device": "sda",
      "instance": "192.168.0.12:9100",
      "job": "node_exporter"
    },
    {
      "__name__": "node_disk_write_time_seconds_total",
      "device": "sda",
      "instance": "192.168.0.7:9100",
      "job": "node_exporter"
    }
  ]
}

```

```
        },
        {
            "__name__": "node_disk_write_time_seconds_total",
            "device": "sda",
            "instance": "localhost:9100",
            "job": "prometheus"
        }
    ]
}
```

## Получение списка меток

**Endpoint:** /api/v1/labels

**Method:** GET | POST

**Параметры запроса:** -

Данный запрос позволяет получить список всех labels.

**Пример:**

```
curl 'http://localhost:9090/api/v1/labels'
```

Ответ будет примерно таким:

```
{
    "status": "success",
    "data": [
        "__name__",
        "address",
        "alertmanager",
        "alertname",
        "alertstate",
        "branch",
        "broadcast",
        "call",
        "code",
        "collector",
        "config",
        "cpu",
        "device",
        "dialer_name",
        "domainname",
        "endpoint",
        "event",
        "fstype",
        "goversion",
        "handler",
        "instance",
        "interval",
        "job",
        "le",
        "listener_name",
        "machine",
        "mode",
        "mountpoint",
```

```
        "name",
        "nodename",
        "operstate",
        "quantile",
        "reason",
        "release",
        "revision",
        "role",
        "rule_group",
        "scrape_job",
        "severity",
        "slice",
        "sysname",
        "version"
    ]
}
```

## Запрос значений для label

**Endpoint:** /api/v1/labels

**Method:** GET

**Параметры запроса:** -

Данный endpoint позволяет получить все значения для определенного label. Синтаксис запроса: /api/v1/labels/<label\_name>/values . Где <label\_name> это имя label, для которого необходимо получить значения.

**Пример:**

```
curl 'http://localhost:9090/api/v1/label/instance/values'
```

Ответ будет примерно таким:

```
{
  "status": "success",
  "data": [
    "129.168.0.7:9090",
    "192.168.0.12:9090",
    "192.168.0.12:9100",
    "192.168.0.7:9090",
    "192.168.0.7:9100",
    "localhost:9090",
    "localhost:9100"
  ]
}
```

## API для получения конфигурации

### Конфигурация сервера

**Endpoint:** /api/v1/status/config

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить текущую конфигурацию сервера.

**Пример:**

```
curl 'http://localhost:9090/api/v1/status/config'
```

В ответ будет возвращена конфигурация сервера, в yaml формате.

## Список ключей запуска

**Endpoint:** /api/v1/status/flags

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить список ключей и их значений, с которыми в данный момент запущен сервер.

**Пример:**

```
curl 'http://localhost:9090/api/v1/status/flags'
```

## Список targets

**Endpoint:** /api/v1/series

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить информацию о всех targets

**Пример:**

```
curl 'http://localhost:9090/api/v1/targets'
```

В ответ Вы получите список targets с их параметрами.

## Rules и Alerts API

### Alertmanagers

**Endpoint:** /api/v1/alertmanagers

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить список всех настроенных alertmanagers и их статус.

**Пример:**

```
curl 'http://localhost:9090/api/v1/alertmanagers'
```

Ответ должен быть примерно таким:

```
{  
  "status": "success",
```

```
"data":{  
    "activeAlertmanagers": [  
        {  
            "url":"http://127.0.0.1:9093/api/v1/alerts"  
        },  
        {  
            "url":"http://192.168.0.7:9093/api/v1/alerts"  
        }  
    ],  
    "droppedAlertmanagers": [  
    ]  
}
```

## Alerts

**Endpoint:** /api/v1/alerts

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить список активных алертов.

**Пример:**

```
curl http://localhost:9090/api/v1/alerts
```

Ответ должен быть примерно таким:

```
{  
    "status":"success",  
    "data":{  
        "alerts": [  
        ]  
    }  
}
```

## Rules

**Endpoint:** /api/v1/rules

**Method:** GET

**Параметры запроса:** -

Данный запрос позволяет получить список всех правил: как record rules, так и alert rules.

**Пример:**

```
curl http://localhost:9090/api/v1/rules
```

Ответ должен быть примерно таким:

```
{  
    "status":"success",  
}
```

```

"data": {
  "groups": [
    {
      "name": "slurm-edu-example-node-exporter-rules",
      "file": "/etc/prometheus/rules.yml",
      "rules": [
        {
          "name": "instance:node_cpus:count",
          "query": "count without(cpu, mode) (node_cpu_seconds_total{mode=\"idle\"})",
          "health": "ok",
          "type": "recording"
        },
        {
          "name": "instance_cpu:node_cpu_seconds_not_idle:rate5m",
          "query": "sum without(mode) (rate(node_cpu_seconds_total{mode!="idle\"})[5m])",
          "health": "ok",
          "type": "recording"
        },
        {
          "name": "instance_mode:node_cpu_seconds:rate5m",
          "query": "sum without(cpu) (rate(node_cpu_seconds_total[5m]))",
          "health": "ok",
          "type": "recording"
        },
        {
          "name": "instance:node_cpu_utilization:ratio",
          "query": "sum without(mode) (instance_mode:node_cpu_seconds:rate5m{mode!="idle\"}) / instance:node_cpus:count",
          "health": "ok",
          "type": "recording"
        }
      ],
      "interval": 15
    },
    {
      "name": "NodeExporterGroup",
      "file": "/etc/prometheus/rules_alert.yml",
      "rules": [
        {
          "name": "ExporterDown",
          "query": "up == 0",
          "duration": 300,
          "labels": {
            "severity": "error"
          },
          "annotations": {
            "description": "Prometheus exporter down\n  VALUE = {{ $value }}\n  LABELS: {{ $labels }}",
            "summary": "Exporter down (instance {{ $labels.instance }})"
          },
          "alerts": [
            {
              "health": "ok",
              "type": "alerting"
            }
          ]
        }
      ]
    }
  ]
}

```

```

},
{
    "name": "HighCpuLoad",
    "query": "100 - (avg by(instance) (irate(node_cpu_seconds_total{mode=\"idle\"}[5m])) * 100) >= 80",
    "duration": 300,
    "labels": {
        "severity": "warning"
    },
    "annotations": {
        "description": "CPU load is >= 80%\n VALUE = {{ $value }}"
    },
    "LABELS": "{{ $labels }}",
    "summary": "High CPU load (instance {{ $labels.instance }})"
},
"alerts": [
],
"health": "ok",
"type": "alerting"
},
{
    "name": "SystemdServiceCrashed",
    "query": "node_systemd_unit_state{state=\"failed\"} == 1",
    "duration": 300,
    "labels": {
        "severity": "warning"
    },
    "annotations": {
        "description": "SystemD service crashed\n VALUE = {{ $value }}"
    },
    "LABELS": "{{ $labels }}",
    "summary": "SystemD service crashed (instance {{ $labels.instance }})"
},
"alerts": [
],
"health": "ok",
"type": "alerting"
}
],
"interval": 15
},
{
    "name": "PrometheusGroup",
    "file": "/etc/prometheus/rules_alert.yml",
    "rules": [
    {
        "name": "PrometheusConfigurationReload",
        "query": "prometheus_config_last_reload_successful != 1",
        "duration": 300,
        "labels": {
            "severity": "error"
        },
        "annotations": {
            "description": "Prometheus configuration reload error\n VALUE = {{ $value }}"
        },
        "LABELS": "{{ $labels }}",
        "summary": "Prometheus configuration reload (instance {{ $labels.instance }})"
    }
]
}

```

```
        },
        "alerts": [
            ],
            "health": "ok",
            "type": "alerting"
        }
    ],
    "interval": 15
}
]
}
```

# Prometheus в Kubernetes

## Установка

В Кубере есть нюансы - K8S не работает с персистентными данными, нужно использовать Persistent Volumes.

Service Discovery тоже работает с оговоркой, и лучше для установки использовать Prometheus Operator - это сторонний проект.

Операторы - это контроллеры, которые расширяют K8S API возможностями создания и настройки других объектов внутри Кубера.

Prometheus постоянно отслеживает изменения в Kube API, и соответственно обновляет настройки Prometheus. Оператор ставит все компоненты - Prometheus, Alertmanager, Grafana, набор метрик.

## Установка Prometheus в kubernetes

1. Авторизация Prometheus в Kubetnetes.

Prometheus ходит к kubelet, авторизуясь по токену. На всех мастер-нодах необходимо добавить в `/etc/kubernetes/kubelet.env`:

```
--authentication-token-webhook \
--authorization-mode=Webhook
```

2. Установка Prometheus.

Установка производится через Helm. В примере используется хранение постоянных данных на cephFS.

2.1 Создаем namespace для Prometheus:

```
kubectl create namespace monitoring
```

2.2 Получаем значения values для изменения значений по умолчанию:

```
helm inspect values stable/prometheus-operator > values.yml
```

В полученном файле:

- Задаем авторизацию для доступа к grafana.
- Выключаем анонимный доступ.
- Включаем ingress для grafana.
- Создаем секрет с паролем basic auth для доступа в alertmanager и prometheus.

```
htpasswd -c auth admin
kubectl create secret generic admin-basic-auth --from-file=auth --
namespace=monitoring
```

Список изменений:

```
alertmanager:
  ingress:
    enabled: true

    annotations:
      nginx.ingress.kubernetes.io/auth-type: basic
      nginx.ingress.kubernetes.io/auth-secret: admin-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required - foo"
    hosts:
      - alertmanager.k8s.example.com
  storage:
    volumeClaimTemplate:
      spec:
        storageClassName: cephfs-provisioner
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
    selector: {}

grafana:
  adminPassword: "topsecret"
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      kubernetes.io/tls-acme: "true"
    hosts:
      - "grafana-k8s.example.com"
    tls:
      - secretName: grafana-k8s-example-com-tls
        hosts:
          - grafana.k8s.example.com

prometheus:
  ingress:
    enabled: true
    annotations:
      nginx.ingress.kubernetes.io/auth-type: basic
      nginx.ingress.kubernetes.io/auth-secret: admin-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required - foo"
    hosts:
      - prometheus.k8s.example.com
  storageSpec:
    volumeClaimTemplate:
```

```
spec:  
  storageClassName: cephfs-provisioner  
  accessModes: ["ReadWriteOnce"]  
  resources:  
    requests:  
      storage: 10Gi  
  selector: {}
```

2.3 Устанавливаем Prometheus:

```
helm install stable/prometheus-operator --name prometheus --namespace monitoring  
-f values.yaml
```