

Monitoring i alerting

w systemach rozproszonych



Monitoring i alerting

Trochę teorii



Terminologia

Monitoring – jest procesem polegającym na gromadzeniu, przetwarzaniu oraz wyświetlaniu w czasie rzeczywistym *metryk* ilościowych¹

Metryka – odzwierciedla jakąś wartość, opisującą badany system w danym punkcie czasu

Aletring – oznacza proces wysyłania powiadomień na podstawie danych z monitoringu, w momencie przekroczenia zdefiniowanych wcześniej progów

1) Za <https://sre.google/workbook/monitoring/>



Co warto monitorować

- Nie ma jednego, z góry narzuconego zestawu metryk, które należy monitorować, wszystko zależy od konkretnego przypadku
- Przed rozpoczęciem gromadzenia metryk, należy uświadomić sobie, jaki cel chcemy osiągnąć. Może to być np:
 - obserwacja danych dla potrzeb biznesowych
 - alerting
 - określenie, jak nasza zmiana wpływa na system (np. który element systemu jest wąskim gardłem po uruchomieniu testów wydajnościowych, lub też jaki wpływ na konwersję ma ostatnio wprowadzona przez nas promocja)



Przykłady metryk

- Zasoby systemowe:
 - Procent użycia procesora
 - Ilość używanej pamięci RAM
 - Ilość operacji wejścia/wyjścia
- Metryki związane z działaniem aplikacji
 - Ilość używanej pamięci z podziałem na obszary (stos, sarta, pamięć natywna)
 - Ilość obsłużonych zapytań http z podziałem na kody odpowiedzi
 - Ilość aktywnych połączeń z bazą danych
 - Czas odpowiedzi (od otrzymania zapytania do odesłania odpowiedzi)
- Metryki biznesowe:
 - Ilość użytkowników, która rozpoczęła proces zakupu, ilość użytkowników, która go dokończyła
 - Ilość zamówień spakowanych przez danego pracownika
 - Kwota udzielonego kredytu dla każdej transakcji



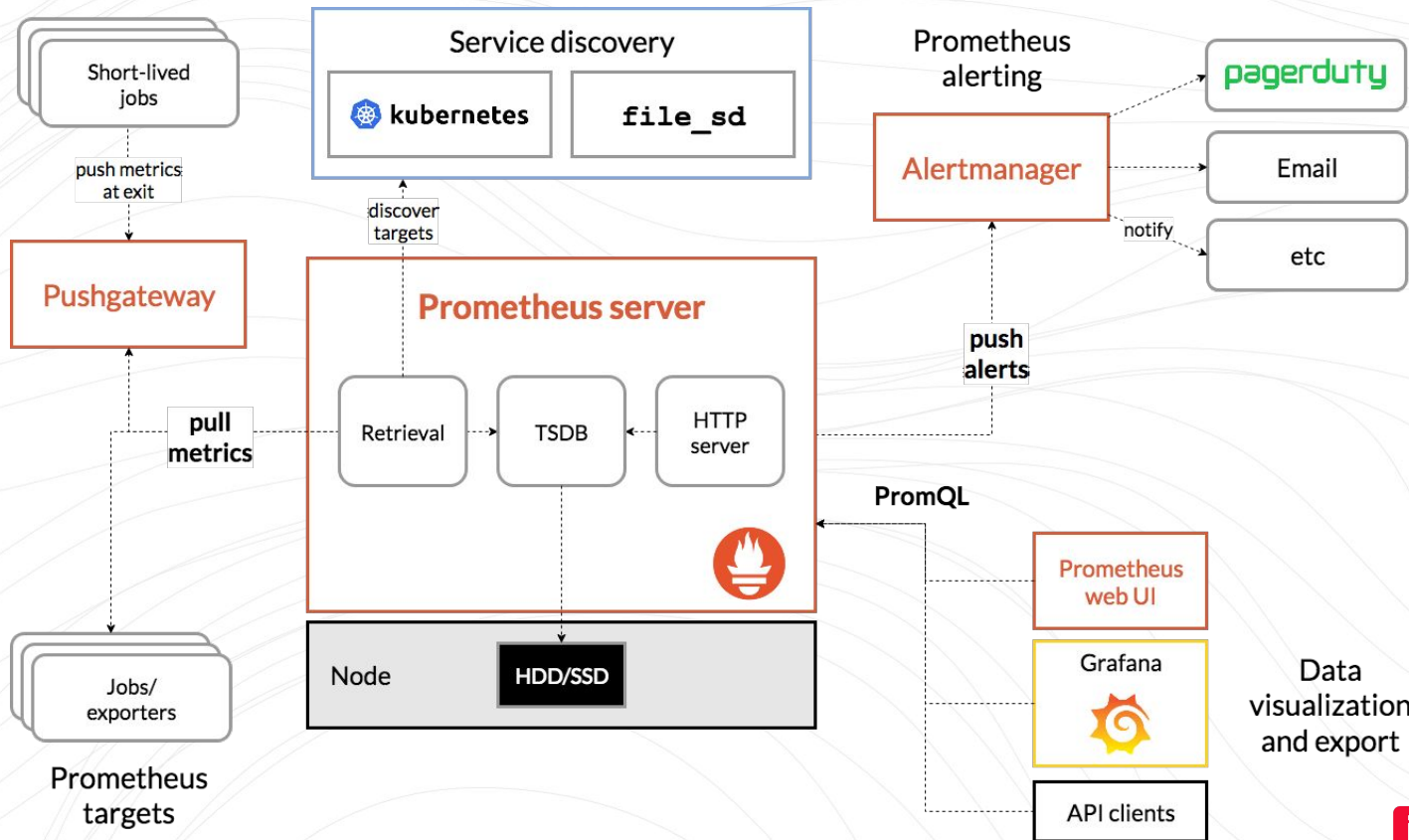
Konfiguracja



Wprowadzenie

- Otwartoźródłowe narzędzie (napisane w języku Go), służące do monitorowania systemów
- Stworzony został odrębny komponent **Alertmanager**, służący do zarządzania alertami (pozwala m.in. unikać zduplikowanych alarmów a także przysyłać je do właściwego celu na podstawie określonych reguł)
- Do pracy z metrykami używany jest dedykowany język zapytań PromQL
- Działanie opiera się na komunikacji HTTP
- Działa w modelu **pull**, tzn. cyklicznie odpytuje zdefiniowane systemy o aktualny zestaw metryk (co ma swoje wady, jak i zalety)
- Jeśli istnieją systemy, które nie mogą udostępniać swoich metryk, można użyć komponentu **Pushgateway** (natomiast autorzy stanowczo odradzają przekształcenie Prometheusa do działania w modelu **push**)
- Istnieje wiele oficjalnych i nieoficjalnych eksporterów, które pozwalają udostępnić metryki w formacie Prometheusa z systemów, które nie robią tego natywnie
- Coraz większa ilość oprogramowania potrafi samodzielnie udostępniać metryki Prometheusa

Architektura





Instalacja i uruchomienie

- Można pobrać, rozpakować i uruchomić plik binarny dla wybranej architektury z <https://prometheus.io/download/>
 - Do archiwum dołączony jest podstawowy plik konfiguracyjny, ustawiający samego siebie jako cel monitoringu
- Dostępny jest oficjalny obraz Docker:
 - Prometheus domyślnie nasłuchuje na porcie **9090**, dlatego też będziemy musieli udostępnić ten port
 - Aby dostarczyć własny plik konfiguracyjny musimy zamontować lokalny plik jako volumen Docker, lub też stworzyć własny obraz, używając oficjalnego obrazu jako bazowego



Pierwsze uruchomienie

← → ↻ localhost:9090/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h

Prometheus Alerts Graph Status Help

☐ Use local time ☐ Enable query history ☒ Enable autocomplete ☒ Enable query hints

Q Expression (press Shift+Enter for newlines)

Table Graph

< Evaluation time >

No data queried yet

Add Panel

Metrics Explorer

Search

- go_gc_duration_seconds
- go_gc_duration_seconds_count
- go_gc_duration_seconds_sum
- go_goroutines
- go_info
- go_memstats_alloc_bytes
- go_memstats_alloc_bytes_total
- go_memstats_buck_hash_sys_bytes
- go_memstats_frees_total
- go_memstats_gc_sys_bytes
- go_memstats_heap_alloc_bytes
- go_memstats_heap_idle_bytes
- go_memstats_heap_inuse_bytes
- go_memstats_heap_objects
- go_memstats_heap_released_bytes
- go_memstats_heap_sys_bytes
- go_memstats_last_gc_time_seconds
- go_memstats_lookups_total
- go_memstats_mallocs_total
- go_memstats_mcache_inuse_bytes
- go_memstats_mcache_sys_bytes
- go_memstats_mspan_inuse_bytes
- go_memstats_mspan_sys_bytes
- go_memstats_next_gc_bytes

Remove Panel

Execute



Prometheus iako cel

localhost:9090/metrics

```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 1.8928e-05
go_gc_duration_seconds{quantile="0.25"} 3.1356e-05
go_gc_duration_seconds{quantile="0.5"} 5.7057e-05
go_gc_duration_seconds{quantile="0.75"} 7.4779e-05
go_gc_duration_seconds{quantile="1"} 0.000231166
go_gc_duration_seconds_sum 0.000741117
go_gc_duration_seconds_count 10
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 32
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.19.4"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.966556e+07
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 5.8374568e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.46767e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 67117
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 9.763288e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 1.966556e+07
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 8.192e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 2.424832e+07
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 77910
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 2.736128e+06
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 3.244032e+07
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage collection.
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.6712306873140547e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 145027
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 9600
# HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures obtained from system.
# TYPE go_memstats_mcache_sys_bytes gauge
go_memstats_mcache_sys_bytes 15600
```



Podstawowe pojęcia

- **Cel (target)** – określa system, który będzie cyklicznie odpytywany w celu pobrania aktualnego zestawu metryk. Jako parametr konfiguracyjny, pozwala również określić dodatkowe ustawienia, które zostaną zastosowane przy odpytywaniu
- **Exporter** – narzędzie, obsługujące protokół celu. Komunikuje się z nim pobierając z niego wybrane metryki a następnie udostępnia je w formacie zrozumiałym dla Prometheusa
- **Etykieta (label)** – określa dodatkowe wymiary metryki. Dzięki temu ta sama metryka może występować wielokrotnie w innym kontekście, jest rozróżniana na podstawie zestawu etykiet (np nazwa hosta, czy też metoda HTTP oraz ścieżka)
- **Próbka (sample)** – pojedyncza wartość danej metryki w konkretnym punkcie czasu



Konfiguracja

- Konfiguracja Prometheusa odbywa się przez edycję pliku YAML
- Domyślnie poszukiwany jest plik *prometheus.yml* w katalogu bieżącym, ale możemy to zmienić poprzez parametr *--config.file*
- Oficjalny obraz Docker uruchamia domyślnie Prometheusa z parametrem *--config.file=/etc/prometheus/prometheus.yml*
- Plik konfiguracyjny zawiera wiele sekcji, z których dwie najważniejsze to **global** oraz **scrape_configs**
- Wszystkie parametry konfiguracyjne znajdziemy w dokumentacji <https://prometheus.io/docs/prometheus/latest/configuration/configuration/>



Sekcja global

- Sekcja **global** zawiera zestaw ustawień, które są wspólne dla innych sekcji, lub mogą dostarczać im wartości domyślne
- Najważniejsze parametry:
 - **scrape_interval** - określa, jak często należy odpytywać cele o bieżące zestawy metryk
 - **scrape_timeout** - jak długo należy czekać na odpowiedź



Sekcja **scrape_config**

- Sekcja **scrape_config** pozwala zdefiniować cele, które będą odpytywane oraz parametry, związane z pobieraniem metryk.
- Najważniejsze parametry:
 - **job_name** - nazwa, zadania, która będzie umieszczona w etykiecie odpowiadającej danej metryce
 - **metrics_path** - ścieżka adresu HTTP, pod jaką dostępne są metryki (domyślnie */metrics*)
 - **honor_timestamps** - określa, czy zbierając metryki, Prometheus powinien uwzględnić znacznik czasu otrzymany wraz z nimi
 - **scheme** - określa protokół celu (http lub https)
 - **static_config** - pozwala zdefiniować statyczną listę celów

- Eksportery pozwalają udostępnić metryki z systemów, które nie mają natywnego wsparcia dla Prometheusa
- Dokumentacja Prometheusa zawiera listę oficjalnych i nieoficjalnych eksporterów
- Niektóre przydatne eksportery:
 - **Node exporter** (https://github.com/prometheus/node_exporter)
 - **Blackbox exporter** (https://github.com/prometheus/blackbox_exporter)
 - **JMX Exporter** (https://github.com/prometheus/jmx_exporter)
 - **nginx-vts-exporter** (<https://github.com/hnlq715/nginx-vts-exporter>)





Typy danych

Wynikiem operacji przeprowadzanych przez Prometheusa może być jeden z następujących typów danych:

- **string** – dowolna wartość tekstowa
- **scalar** – pojedyncza wartość liczbowa (Prometheus operuje na liczbach zmiennoprzecinkowych)
- **instant vector** – zestaw wartości, posiadające ten sam znacznik czasu
- **range vector** – zestaw wyników, z których każda zawiera osobny zestaw wartości, odzwierciedlający stan metryki w poszczególnych punktach czasu, w określonym przedziale



Rodzaje metryk

- **Counter** – służy do pomiaru metryki, której wartość może tylko rosnąć (lub może zostać wyzerowana przy restarcie).
 - Liczba sekund od uruchomienia usługi
 - Ilość żądań HTTP, zakończona powodzeniem
 - Ilość użytkowników, która oddała głos w ankiecie
- **Gauge** – wskazuje na wartość danej metryki w danym punkcie czasu. Wartość może się zarówno zwiększać, jak i zmniejszać przy kolejnych pomiarach.
 - Ilość używanej pamięci
 - Cena danego towaru
 - Ilość aktualnie zalogowanych użytkowników
- **Histogram** – Pozwala grupować wartości w z góry określonych przedziałach, zamiast przechowywać każdą wartość z osobna. Pozwala mierzyć nam rozkład poszczególnych wartości.
 - Czas trwania zapytania http
 - Wartość koszyka klienta
- **Summary** – Podobna do histogram. Metryka jednak wyliczana jest po stronie klienta. Dzięki temu dużo szybciej można generować wyniki zapytania. Nie pozwala jednak na agregowanie wyników z wielu instancji.



Filtrowanie metryk

- Wskazanie samej metryki w zapytaniu zwróci nam jej wartości dla wszystkich wymiarów z osobna (wymiary określone są przez zestaw etykiet): **`jvm_memory_pool_committed_bytes`**

- Możemy ograniczyć wyniki do wybranych etykiet:

`jvm_memory_pool_committed_bytes{job="app", pool="g1-old-gen"}`

- Określając wartości etykiet możemy używać następujących operatorów:
 - **`=`** wybiera tylko te metryki, których wartość wybranej etykiety dokładnie odpowiada wartości z zapytania (**`{job = "app"}`**)
 - **`!=`** wybiera wyłącznie te metryki, dla których wartość etykiety jest różna od podanej w zapytaniu (**`{pool != "g1-old-gen"}`**)
 - **`=~`** wybiera metryki, których wartość etykiety pasuje do podanego wyrażenia regularnego (**`{instance =~ "app:[0-9]+"}}`**)
 - **`!~`** wybiera wyłącznie metryki, których wartość etykiety nie pasuje do wskazanego wyrażenia regularnego (**`{device !~ "/dev/dm-[0-9]+"}}`**)



Operatory arytmetyczne

- Dodawanie (+)
- Odejmowanie (-)
- Mnożenie (*)
- Dzielenie (/)
- Dzielenie modulo (%)
- Potęgowanie (^)



Operatory agregujące

- **sum** - sumuje wartości dla wszystkich lub wybranych wymiarów:
 - `sum(jvm_memory_pool_committed_bytes)`
 - `sum by (instance) (jvm_memory_pool_committed_bytes)`
 - `sum without (pool) (jvm_memory_pool_committed_bytes)`
- **min** - wybiera najmniejszą wartość metryki dla wszystkich lub wybranych wymiarów
 - `min(node_cpu_core_throttles_total)`
- **max** - wybiera największą wartość metryki dla wszystkich lub wybranych wymiarów
 - `max(node_cpu_core_throttles_total)`
- **avg** - wylicza średnią arytmetyczną dla wartości metryk spośród wszystkich lub wybranych wymiarów
 - `avg(node_cpu_scaling_frequency_hertz)`
- **group** - przypisuje wszystkim lub wybranym wymiarom wartość 1. Operator użyteczny, jeśli chcemy np poznać wszystkie wartości danej etykiety:
 - `group by(pool) (jvm_memory_pool_committed_bytes)`
- **stddev** - wylicza odchylenie standardowe dla wszystkich lub wybranych wymiarów
 - `stddev(node_cpu_scaling_frequency_hertz)`



Operatory agregujące

- **stdvar** – oblicza wariancję dla wszystkich lub wybranych wymiarów
 - `stdvar(node_cpu_scaling_frequency_hertz)`
- **count** – zlicza ilość elementów, które zawiera wektor
 - `count(jvm_memory_pool_committed_bytes)`
- **count_values** – grupuje i zlicza wymiary z taką samą wartością. Wynikowa wartość będzie miała dodatkową etykietę, której nazwę przekazujemy w pierwszym argumencie. Wartością tej etykiety będzie unikalna wartość z pierwotnej metryki
 - `count_values("http_response_total", http_response_total)`
- **bottomk** – ogranicza wynik do k najmniejszych wartości (gdzie k przekazywany jest jako pierwszy argument)
 - `bottomk(3, node_cpu_scaling_frequency_hertz)`
- **topk** – ogranicza wynik do k najwyższych wartości (gdzie k przekazywany jest jako pierwszy argument)
 - `topk(3, node_cpu_scaling_frequency_hertz)`
- **quantile** – oblicza wskazany kwantyl (który może przyjmować wartość pomiędzy 0 a 1)
 - `quantile(0.5, jvm_memory_pool_committed_bytes)`



Wybrane funkcje

- Listę dostępnych funkcji znajdziemy w dokumentacji <https://prometheus.io/docs/prometheus/latest/querying/functions/>
- **rate** – jako argument przyjmuje *range vector*. Pozwala obliczyć tempo wzrostu na sekundę wartości metryki, uśrednione dla danego przedziału. Funkcja powinna być używana z metrykami typu *counter*. Dobrze sprawdza się w regułach alertujących
 - `rate(http_response_total[5m])`
- **irate** – działa podobnie do *rate*, jednak zamiast uśredniać wartości w całym przedziale czasowym, bierze pod uwagę wyłącznie dwie ostatnie wartości w nim. Powoduje to, że najczęściej wykres jest mniej wygładzony. Funkcja ta lepiej się sprawdza przy wartościach cechujących się dużą zmiennością
 - `irate(http_response_total[5m])`
- **increase** – działa podobnie do *rate*, jednak zamiast przyrost wartości na sekundę, pokazuje przyrost w całym zakresie czasowym
 - `increase(http_response_total[5m])`



Wybrane funkcje

- **scalar** – zamienia jednoelementowy *instant vector* w wartość typu scalar. Jeśli wektor nie posiada dokładnie jednego elementu, zwrócone zostanie NaN
- **delta** – funkcja użyteczna przy metrykach typu *gauge*. Jako argument przyjmuje *range vector* i zwraca *instant vector*, zawierający wartości stanowiące różnice pomiędzy pierwszą a ostatnią wartością w zakresie.
 - `delta(jvm_threads_states[5m])`
- **resets** – funkcja użyteczna przy metrykach typu *counter*. Jako argument przyjmuje *range vector* i zwraca jak wiele razy dany licznik został zresetowany w danym przedziale czasu
 - `resets(host_storage_device_ops_write_total[1m])`
- **predict_linear** – funkcja służąca do przewidywania przyszłej wartości na podstawie wartości historycznych. Jako pierwszy argument przyjmuje *range_vector*, natomiast drugi argument jest skalarem, wskazującym ile sekund od chwili obecnej chcemy przewidzieć.
 - `predict_linear(host_storage_mount_space_free_bytes[2h], 3600)`



Wybrane funkcje

- **round** – zaokrągla wartość
 - `round(node_cpu_seconds_total)`
- **time** – zwraca bieżący znacznik czasu (dla momentu, w którym wyrażenie było przetwarzane)
 - `time()`
- **<aggregation>_over_time** – gdzie <aggregation> zastępujemy wybranym operatorem agregującym. Funkcje te przyjmują *range vector* jako argument i zwracają *instant vector* zawierający zagregowane wyniki dla wybranego zakresu czasu
 - `avg_over_time(host_storage_device_data_write_bytes_total[1m])`
- **histogram_quantile** – pozwala wyliczyć wybrany kwantyl wartości histogramu:
 - `histogram_quantile(0.95, rate(app_div_result_bucket[30s]))`



Recording rules

- Prometheus obsługuje dwa rodzaje reguł:
 - recording rules
 - alerting rules
- *Recording rules* pozwalają nam wstępnie obliczyć wartość danego wyrażenia i zapisać je podobnie, jak wartości zwykłych metryk
- Jest to szczególnie użyteczne przy tworzeniu zapytań, których wykonanie wymaga sporo pracy, a które wykonujemy często (np przy dashboardach)
- Reguły zapisujemy w pliku yaml a następnie dołączamy ten plik poprzez sekcję *rule_files* w pliku konfiguracyjnym Prometheusa
- Zamiast restartować serwer Prometheusa możemy do jego procesu wysłać sygnał *SIGHUP*, który przeładuje reguły
- Do weryfikacji poprawności składni pliku możemy użyć narzędzia **promtool**:
 - `promtool check rules my_rules.yml`



Przykładowy plik reguł

```
groups:
- name: my_rules
  rules:
    - record: instance_path:node_cpu_seconds_total_by_mode_per_node_cpu_seconds_total:percentage
      expr: (sum by(mode) (node_cpu_seconds_total) / scalar(sum(node_cpu_seconds_total))) * 100
```



Alerting rules

- Reguły alertów pozwalają na definiowanie warunków, których spełnienie pozwoli na wygenerowanie alarmu
- Definiuje się w ten sam sposób, co *recording rules*
- Alarmy można obserwować w interfejsie webowym Prometheusa na zakładce *alerts*
- Mogą one być również przesyłane do zewnętrznego systemu, jak np. *Alertmanager*
- *Alerting rules* posiadają dodatkowe pola:
 - **for** - pozwala opóźnić wywołanie alarmu. Alarm stanie się aktywny dopiero wówczas, gdy wyrażenie będzie spełnione przez zdefiniowany czas (natomiast wcześniej będzie w stanie *pending*)
 - **labels** - dodatkowe etykiety, które zostaną dołączone do alarmu. Używane mogą być np do grupowania alarmów, lub do przekazania ich do właściwego kanału powiadomień
 - **annotations** - dodatkowe, opisowe informacje na temat alarmu
- Zarówno etykiety, jak i adnotacje mogą używać języka szablonów do tworzenia dynamicznej zawartości
- Każdy aktywny alarm generuje dodatkową metrykę, która przyjmuje wartość 1




Przykładowy plik konfiguracyjny

```
global:
  smtp_smarthost: 'mailhog:1025'
  smtp_from: 'alertmanager@example.org'
  smtp_require_tls: false



route:
  group_by: ['service']
  group_wait: 30s
  group_interval: 15s
  receiver: 'team-X-mail'

receivers:
- name: 'webhook'
  webhook_configs:
    - url: 'https://14e80351-6d10-4815-9e3d-b3cd287ffaed.requestcatcher.com/test'
- name: 'team-X-mail'
  email_configs:
    - to: 'foo@example.org'
```

← → ↺ ⓘ localhost:9090/alerts?search=

 Prometheus

Alerts Graph Status ▾ Help

  ⓘ

☒ Inactive (0)

☒ Pending (0)

☒ Firing (1)

🔍

Filter by name or labels

☒ Show annotations

/etc/prometheus/alerts.yml > JVM

firing (1)

▼ JvmTooManyThreads (1 active)

name: JvmTooManyThreads

expr: `sum by (instance) (jvm_threads_states) > 10`

for: 30s

labels:

severity: low

annotations:

description: An instance {{ \$labels.instance }} JVM has too many thereads ({{ \$value }})

summary: Too many JVM threads on instance {{ \$labels.instance }}

Labels	State	Active Since	Value
<div><div>alertname=JvmTooManyThreads</div><div>instance=app:9095</div><div>severity=low</div></div>	<div>FIRING</div>	2022-12-20T00:01:10.112751043Z	34

Annotations

description

An instance app:9095 JVM has too many thereads (34)

summary

Too many JVM threads on instance app:9095



Przykładowa definicja alarmu

```
groups:
- name: JVM
  rules:
  - alert: JvmTooManyThreads
    expr: sum by(instance) (jvm_threads_states) > 15
    for: 3m
    labels:
      severity: low
    annotations:
      summary: "Too many JVM threads on instance {{ $labels.instance }}"
      description: "An instance {{ $labels.instance }} JVM has too many thereads ({{ $value }})"
```




- Alertmanager jest usługą, która odbiera alarmy z systemu monitoringu (jak Prometheus) i zarządza nimi
- Możliwości:
 - kierowanie alarmu do właściwego kanału powiadomień, jak e-mail, sms, Slack itp.
 - wyciszanie alarmów (można sprawić, że nie będą przychodziły powiadomienia o wybranych alarmach przez określony czas)
 - Tworzenie powiązań pomiędzy alarmami tak, że wywołanie alarmu oznaczającego poważną awarię (np niedostępność bazy danych), spowoduje że nie będą dodatkowo wysyłane alarmy dotyczące zależnych usług
 - Grupowanie alarmów (na podstawie etykiet)



Konfiguracja

- Podobnie, jak w przypadku Prometheusa, Alertmanager konfigurujemy przy pomocy pliku YAML, do którego ścieżkę możemy przekazać przy pomocy argumentu wiersza poleceń `--config.file`
- Pełny opis konfiguracji znajdziemy w dokumentacji <https://prometheus.io/docs/alerting/latest/configuration/>
- Najważniejsze elementy:
 - sekcja **global**
 - konfiguracja poszczególnych kanałów komunikacji, jak np serwer pocztowy, adres slack, czy adres PagerDuty
 - sekcja **receivers**, pozwalająca zdefiniować odbiorców wiadomości (np docelowy adres e-mail, czy kanał Slack)
 - sekcja **route**:
 - sekcja **routes**, w której definiujemy, do jakiej grupy odbiorców z sekcji *receivers* mają trafiać konkretne alarmy (na podstawie etykiet)
 - dyrektywa **receiver**, określająca domyślnego odbiorcę



Konfiguracja

- Możemy poinformować Prometheusa, aby przekazywał alarmy do Alertmanagera, dopisując jego adres w konfiguracji: *alerting/alertmanagers/static_configs/targets*
- Alertmanager posiada wbudowane szablony powiadomień. Jeśli jednak nam to nie wystarcza, możemy tworzyć własne.



Przykładowy plik konfiguracyjny

```
global:
  smtp_smarthost: 'mailhog:1025'
  smtp_from: 'alertmanager@example.org'
  smtp_require_tls: false

route:
  group_by: ['service']
  group_wait: 30s
  group_interval: 15s
  receiver: 'team-X-mail'

receivers:
- name: 'webhook'
  webhook_configs:
    - url: 'https://14e80351-6d10-4815-9e3d-b3cd287ffaed.requestcatcher.com/test'
- name: 'team-X-mail'
  email_configs:
    - to: 'foo@example.org'
```





Wprowadzenie

- Grafana jest narzędziem, służącym do wizualizacji metryk a także do przeprowadzania podstawowej analityki
- Istnieje zarówno wersja open source (Grafana OSS), jak i wersja płatna, zawierająca więcej funkcji (Grafana Enterprise)
- Istnieje też usługa Grafana Cloud, której można używać za darmo, z pewnymi ograniczeniami:



Free Forever Cloud

3 users

10,000 active series for metrics

50 GB of logs

50 GB of traces

30 notifications for OnCall

14-day retention



Praca z Grafana

- Dodawanie źródła danych (data source)
- Przeglądanie metryk (explore)
- Utworzenie dashboardu
- Dodanie paneli
- Analiza danych
- Eksport i import dashboardu



Przykładowy dashboard

