Modern Monitoring With Prometheus

Anton Weiss, Otomato Software

@antweiss, @otomato_sw

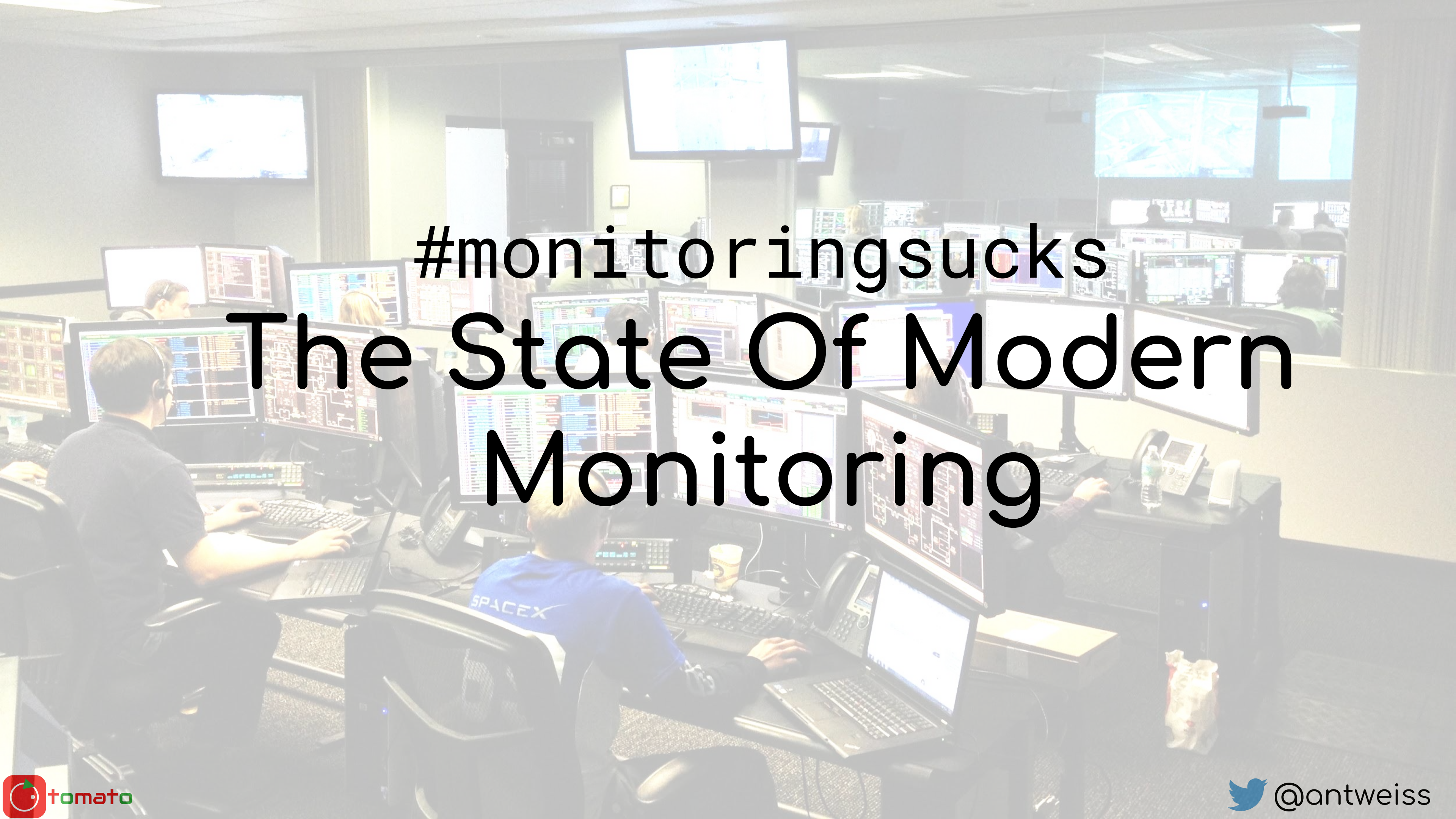
https://otomato.link

https://devopstrain.pro

Setup

- Enter Strigo Lab: https://goo.gl/pf1G44
- Sign in to Slack (will be used for alerts)

Tweet: I'm doing a workshop with @StrigoIO and it's awesome!



#monitoringsucks

The State Of Modern Monitoring

tomato

@antweiss

Types of Monitoring Systems

Single System

- monit
- htop
- sar
- nmon

Distributed

- Nagios
- Zabbix
- collectd
- Ganglia

Types of Monitoring Systems

Self-hosted VS. Monitoring-As-A-Service

- Nagios
- Zabbix
- Sensu
- Icinga
- Prometheus
- Sysdig

- Amazon CloudWatch
- DataDog
- NewRelic
- ServerDensity
- SolarWinds
- Sysdig.io

Types of Monitoring Systems

Pull-based VS. Push-based

- Nagios
- Zabbix
- Icinga
- Prometheus

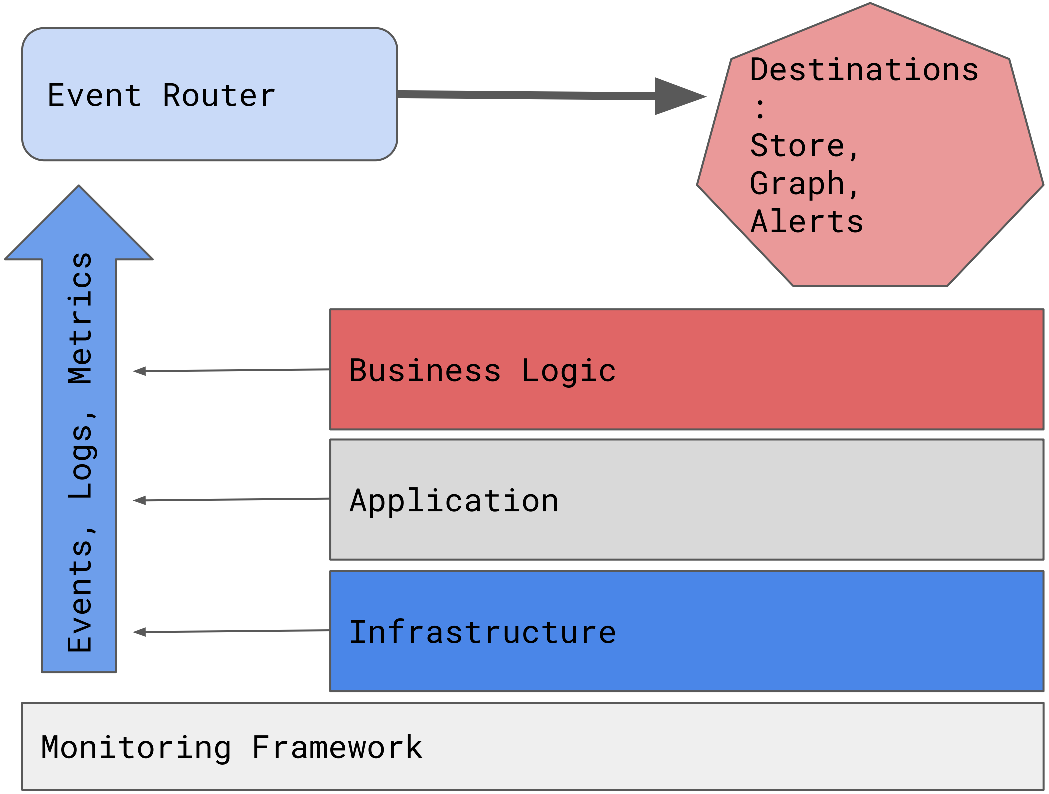
- Graphite
- Sensu
- DataDog
- CloudWatch

https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/

Types of Monitoring Systems

- System (Server) Monitoring
- Network Monitoring
- API Monitoring
- Application Performance Management (APM)

The 3 Layers Of Monitoring



Event Router

Destinations
:
Store,
Graph,
Alerts

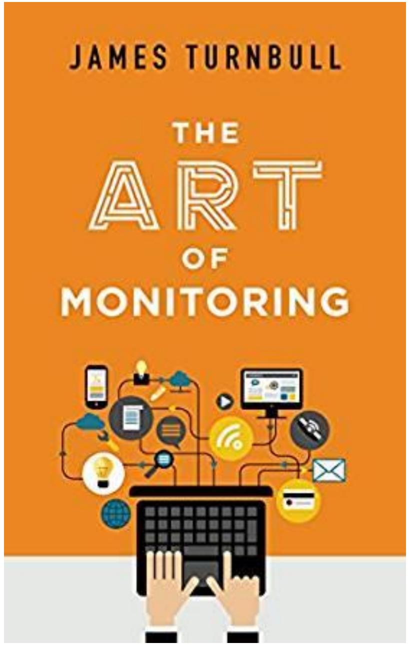
Events, Logs, Metrics

Business Logic

Application

Infrastructure

Monitoring Framework



Business Level Metrics

Number of Signups

Number of Sales

Volume of Transactions

A/B Testing Results

API Usage Rate

Application Level Metrics

Transaction Times

Count of Requests

Response Times

Exceptions

Roundtrip Times and Counts (with the help of distributed tracing)

Infrastructure Level Metrics

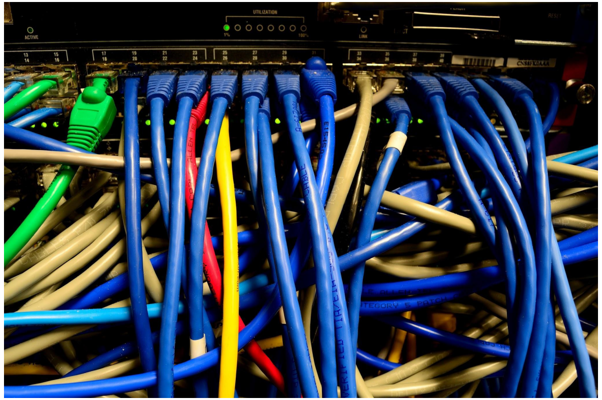
Web Server Traffic

CPU Load

Disk Usage

Number of Open Files

Memory Usage



Time Series Databases

A time series database (TSDB)

is a software system that is optimized for handling time series data - arrays of numbers indexed by time.

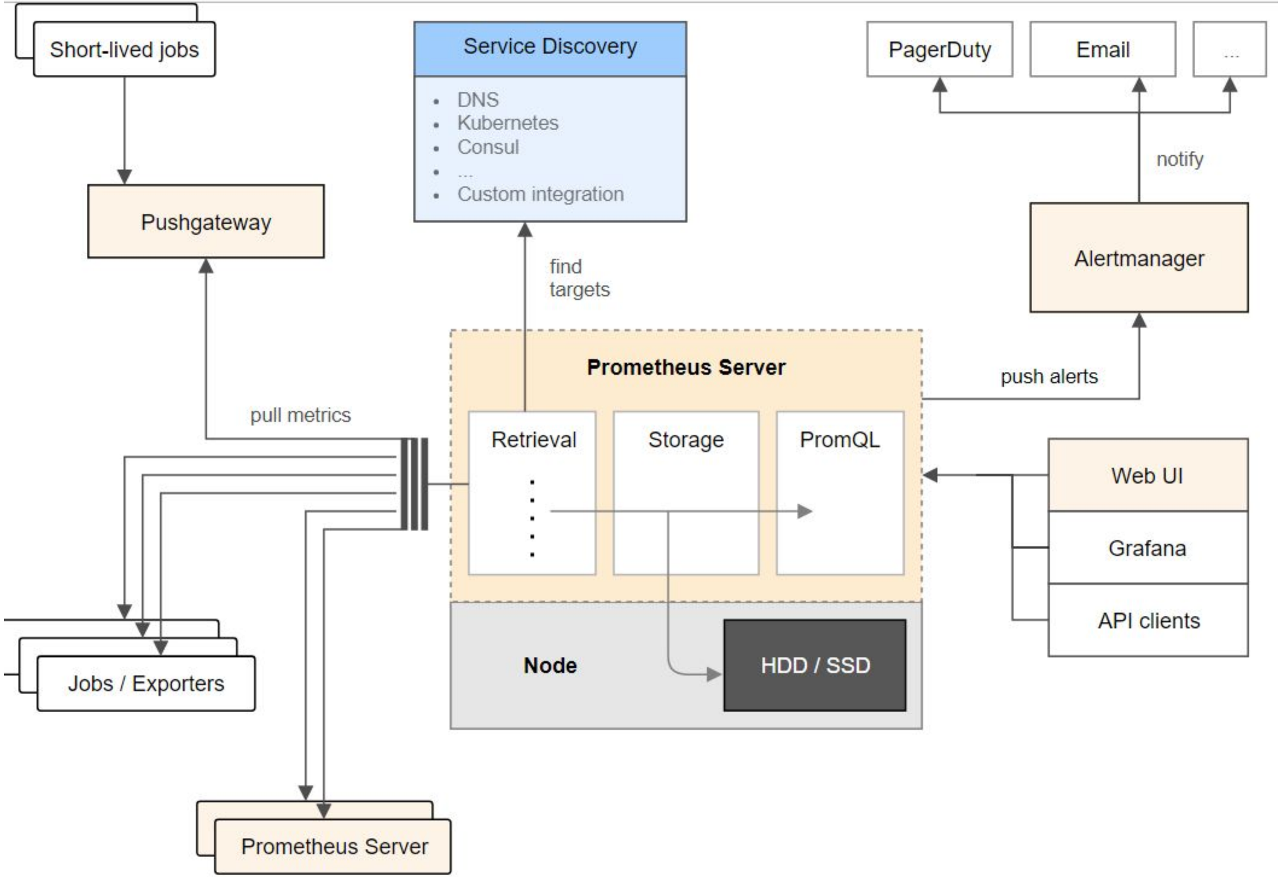
Examples:

RRDTool : stands behind Nagios, Ganglia, Cacti etc.
InfluxDB
Graphite
Prometheus

Prometheus

- Time Series Database
- Pull Mode
- Expressive Query Language (PQL)
- Dimensional data model
- Multiple Client Libraries
- Smart Alerting
- Grafana Integration for Visualization

Prometheus



tomato

@antweiss

Scraping the metrics from:

Application Instrumentation:

```
from flask import Flask
from flask_prometheus import monitor

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello Prometheus!'

monitor(app, port=8000)
app.run(host=’0.0.0.0’)
```

The metrics are available at http://<your_host>:8000

Scraping the metrics from:

Exporters:

Node Exporter: Prometheus exporter for hardware and OS metrics exposed by *NIX kernels, written in Go with pluggable metric collectors.

MySQL server exporter (**official**)

MongoDB exporter

Jenkins Prometheus Plugin - expose HTTP endpoint for scraping

AWS Health Exporter: scrapes the AWS Status (via the AWS Health API) and exports it via HTTP for Prometheus consumption. That allows you to alert on certain AWS status updates or to just make them visible on your dashboards.

Installing Prometheus

Prometheus is a single statically-linked Go binary.
But we can of course run it as a Docker container.
Clone the lab files:

```
git clone https://github.com/otomato-gh/otom8-prometheus.git
```

Look at docker-compose.yml

```
cd otom8-prometheus
docker-compose up -d
```

Prometheus Built-In Exporter

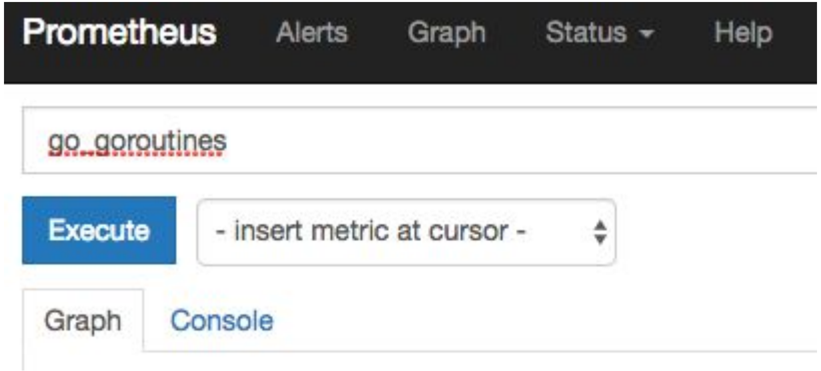
Browse to http://<your_host>:9090/metrics

Now browse to http://<your_host>:9090

Enter: go_goroutines

Click on 'Execute'

Look at the graph.



go_goroutines

Run the Node Exporter

The node exporter in our lab runs in a container

The metrics for node exporter are at http://node:9100

Look at prometheus.yml:

```
scrape_configs:
...
  - job_name: 'node'
...
    scrape_interval: "15s"
    static_configs:
  - targets: ['node:9100']
```

Browse back to http://your_host:9090 and look for metric : *node_load15*

Now look for metric: node_cpu

Jobs and Instances

```
node_load5{instance="node:9100",job="node"}
```

For each target in a job Prometheus automatically creates a new metric labelled with an ‘instance’ corresponding to target address.

```
static_configs:

    - targets: ['some-host:5000', 'other-host:5000']
```

Prometheus Data Model

Every time series is uniquely identified by its *metric name* and a set of *key-value pairs*, also known as *labels*.

Labels enable Prometheus's multi-dimensional data model.

The query language allows filtering and aggregation based on these dimensions.

Changing any label value, including adding or removing a label, will create a new time series.

```
node_cpu{cpu="cpu0",instance="node:9100",job="node",mode="system"}
node_cpu{cpu="cpu1",instance="node:9100",job="node",mode="system"}
```

Querying Prometheus

Prometheus provides a functional expression language (PromQL) that lets the user select and aggregate time series data in real time. The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems via the HTTP API.

Examples:

Filter on ‘job’ label:

```
node_cpu{cpu="cpu1"}  # try this for cpu2
```

Visualize the metrics:

```
process_open_fds  # Switch to ‘Graph’ tab
```

Querying Prometheus

Examples:

Return a whole range of time (in this case 2 minutes) for the same vector, making it a range vector:

```
node_cpu{cpu="cpu0"}[2m]  #note - this can’t be graphed!
```

Apply functions (like ‘sum’, ‘count’, ‘rate’):

```
sum(prometheus_http_request_duration_seconds_count)
    rate(node_cpu{cpu="cpu0"}[2m]) # this can be graphed!
```

Use operators:

```
process_max_fds - process_open_fds {job="prometheus"}
```

Exercise:

- Run a query to find the sum of all the time *all cpus* on our node spent doing kernel work (mode=”system”)
- Change the query to get all the time series from the last 5 minutes
- Now get the per-second rate of change in cpu usage (use the ‘rate’ function)
- Create load on you node with `dd if=/dev/zero of=/dev/null`
- Watch the graph for your last query change

Integrating Prometheus with Grafana

Grafana

Grafana - the open-source graphing, visualization and analytics platform Features native Prometheus integration.

In our lab:

```
    git checkout with-grafana
    #Inspect docker-compose.yml
    docker-compose up -d
```

Grafana listens on port 3000 by default - so please open port 3000 in security group/firewall.

Log in as admin/admin

Creating a Prometheus data source

To create a Prometheus data source:

1. Click on the Grafana logo to open the sidebar menu.
2. Click on "Data Sources" in the sidebar.
3. Click on "Add New".
4. Select "Prometheus" as the type.
5. Set the appropriate Prometheus server URL (http://prometheus:9090/)
6. Adjust other data source settings as desired (for example, turning the proxy access off).
7. Click "Save&test" to save the new data source.

Defining a Dashboard

1. Click on the Grafana logo to open the sidebar menu.
2. Dashboards->New
3. Click on ‘Panel Title’ -> Edit
4. Under the "Metrics" tab, select your Prometheus data source (bottom right).
5. Enter any Prometheus expression into the "Query" field, while using the "Metric" field to lookup metrics via autocompletion.

Alerting

Basic rules of thumb:

- keep alerting simple
- alert on symptoms
- have good consoles to allow pinpointing causes
- avoid having pages where there is nothing to do

Can be created with AlertManager or in Grafana.

Alerting

```
git checkout add_alerts
#Inspect docker-compose.yml
docker-compose up -d
# In Prometheus Web UI - go to ‘Alerts’

 alert: HighLoad
 expr: node_load1{job="node"}
    > 0.5
 for: 1m
 labels:
    severity: page
 annotations:
    summary: High load for the last minute on {{ $labels.instance }}
```

Alerting

Define Rules:

In **prometheus.yml**:

```
# Load rules once and periodically evaluate them according to the
# global 'evaluation_interval'.
rule_files:
  - "/etc/prometheus/rules.yaml"
```

Recording Rules:precompute frequently needed or computationally expensive expressions

Alerting Rules: define alert conditions based on Prometheus expression language expressions and to send notifications about firing alerts to an external service.

Alerting

In **rules.yaml**:

```
groups:
  - name: node
    rules:
    - alert: HighLoad
      expr: node_load1{job="node"} > 0.5
      for: 1m
      labels:
        severity: page
      annotations:
        summary: "High load for the last minute on {{
$labels.instance }}"
```

Alertmanager

Takes care of

- Deduplicating
- Grouping
- Routing alerts to the correct receiver integration such as:
  - Email
  - PagerDuty
  - OpsGenie
  - Slack
  - etc.
- It also takes care of silencing and inhibition of alerts.

Alertmanager - configuration

Take a look at :

- prometheus.yml
- docker-compose.yaml
- alertmanager.yaml

Access alertmanager at http://your_host:9093

Exercise

In *docker-compose.yaml* : replace
`--web.external-url=http://158.177.121.27:9093` with your host's IP

Run: docker-compose up -d

Create load on your node with: `dd if=/dev/zero of=/dev/null`

Watch the alert on Prometheus, Alertmanager and Slack

Bonus: Rename the alert so we know it's from you.

Metric Types

Gauge: a snapshot of state

Usually when aggregating them you want to take a sum, average, minimum, or maximum.

Example:

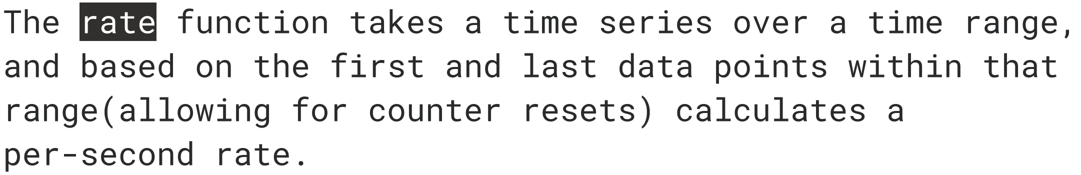
```
process_open_fds
sum(process_open_fds)
```

Metric Types

Counter: tracks the number or size of events.

Value is the total since measurement started.
But what we really want to know is how quickly the counter is increasing over time.
This is usually done using the ***rate*** function:

```
rate(node_cpu{cpu="cpu0", mode="user"}[1m])
```



The rate function takes a time series over a time range, and based on the first and last data points within that range(allowing for counter resets) calculates a per-second rate.

Metric Types

Summary: usually contains both a _sum and _count, and sometimes a time series with no suffix with a *quantile* label. The _sum and _count are both counters.

Summaries are calculated on the client side. Therefore their quantiles can't be combined together.

Summary in our *prometheus* and *node_exporter*:

```
go_gc_duration_seconds
go_gc_duration_seconds_sum
go_gc_duration_seconds_count
```

Metric Types

Histogram: allows sampling the observations in pre-defined buckets. For example, a request latency histogram can have buckets for <10ms, <100ms, <1s, <10s.

Example:

```
    prometheus_http_response_size_bytes_bucket
    prometheus_http_response_size_bytes_count
    prometheus_http_response_size_bytes_sum
```

Analyze the results of:

```
    prometheus_http_response_size_bytes_bucket{handler='/query'}
```

Histograms allow calculating quantiles:

```
histogram_quantile(0.8, prometheus_http_response_size_bytes_bucket)
```

Instrumentation

Allowing our application code to be monitored by exposing metrics and logs.

Can be done by directly exposing metrics or by using a client library.

Official client libraries :

    Go, Python, Java and Ruby.

Unofficial: bash, C++, Node.js, Rust, etc...

Instrumentation

Let's run an application:

```
git checkout add_app
```

It's a REST-api web service based on python-Eve framework

It allows to register and retrieve user records in Mongo.

Instrumentation

Look at the code in app/api.py:

```
from prometheus_client import start_http_server
…
if __name__ == '__main__':
        start_http_server(8000)
```

Metrics are exposed at app_url:8000/metrics

Instrumentation

Let’s run the application:

```
docker-compose app -d
```

Check the metrics from the app at http://your_host:8000/metrics

Adding and retrieving records

You can add users by running:

```
curl -d '[{"firstname": "Bender", "lastname": "Rodriguez",
"type": ["business"]}]' -H 'Content-Type: application/json'
http://localhost:5000/users
```

And retrieve them with:

```
curl http://localhost:5000/users
```

or

```
http://localhost:5000/users/{user_id}
```

Note: you can use data.json and data.csv to load more user records to the database.

Exercise:

Edit prometheus.yml:

- Add a job named ‘usersApi’
    - Set scrape interval to 10 seconds
    - Add a target ‘app:8000’
    - Back on console: run `docker-compose restart prometheus`
    - Check by searching prometheus for ‘python_info’

Let’s add a counter

In app/api.py:

```
    from prometheus_client import start_http_server, Counter
    …
    HELLOREQS = Counter('hello_reqs_total', "Total hellos called")
    @app.route('/hello')
    def hello():
        HELLOREQS.inc()
```

```
> docker-compose restart app
```

Check prometheus for ‘hello_reqs_total’. Reload http://your_host:5000 a few times. Now check again.

Exercise:

- Add a counter for calls to “/version” endpoint
- Verify it works

Counting exceptions

```
import random
...
HELLOEXC = Counter('hello_exceptions_total', "Exceptions in hello")
...
@app.route('/hello')
def hello():
...
with HELLOEXC.count_exceptions():
          if random.random() < 0.2:
            raise Exception
```

A gauge:

Examples of gauges:

- the number of items in a queue
- memory usage of a cache
- number of active threads
- the last time a record was processed

A gauge:

```
from prometheus_client import Gauge
INPROGRESS = Gauge('hello_worlds_inprogress',
        'Number of Hello Worlds in progress.')
LAST = Gauge('hello_world_last_time_seconds',
        'The last time a Hello World was served.')”
#hello starts:
...
INPROGRESS.inc()
#hello ends
...
LAST.set(time.time())
```

Exercise:

Add a gauge to measure the number of ‘/version’ requests in process and the last time ‘/version’ was accessed.

Tip:

Can use a decorator:

```
    @INPROGRESS.track_inprogress()
    def hello():
```

Use histogram to track latency

```
from prometheus_client import Histogram
...
LATENCY = Histogram('ping_latency_seconds',
        'Latency for ping.')
...
@app.route('/ping')
@LATENCY.time()
```

Exercise:

- Rewrite the ping method so that the ping grows with each call
- Create an alert when at least 10% of pings have latency > 5s for the last 3 minutes
- Send the alert to Slack

Bonus Exercise:

Instrument ‘/users’ - the main API endpoint.

Note that you’ll need to add instrumentation to the hook functions: `pre_users_get_callback, pre_users_post_callback`

Add a counter for the number of requests to user endpoint.

Label ‘post’ and ‘get’ requests separately - e.g:

```
c = Counter('my_requests_total', 'HTTP Failures', ['method'])
c.labels('get').inc()
```

Create an alert when the counter grows faster than 5 calls a minute.

Make the alert fire

References:

Metrics Catalog: https://github.com/monitoringsucks/metrics-catalog

Brian Brazil's blog: https://www.robustperception.io/

Brian Brazil's book: https://www.amazon.com/Prometheus-Infrastructure-Application-Performance-Monitoring-ebook/dp/B07FCV2VVG/

devopstrain

Want to learn more DevOps stuff?

https://www.devopstrain.pro/