

Monitoring Microservices with Prometheus - Slides

101: Introduction

- To us?
- To you?
- To the training...

Timings

Workshop Environments

Access to Materials

What on Earth Are We Doing?

- Keep these questions in mind during the training
- Please ask me questions (I might not know the answer, but it makes it more fun for me!)
- Monitoring

Questions: What is monitoring? Why should we do it? When is it necessary? How do we do it?

- Prometheus

Questions: What is it? What is different about it? Why that and not others? How do we use it?

Assumptions

- Always question your assumptions (Is this right? What is that? Why? Etc.)

Why is your business interested in monitoring? What are their goals? What are their key needs?

Why are you interested in monitoring? What do you want to know? Why do you think we need monitoring?

Your Users

The **business** wants to know that their users are being served *appropriately*.

The business is guaranteeing that their users receive their service:

- In a timely manner
- To an expected standard



Your Code

You need to ensure that your work fulfils the business' needs.

You are guaranteeing:

- Availability
- Provides an expected service



What is Normal?

In reality, when your service is running "normally", then your business is providing its service "normally" and the user is "normally" happy with the service.

Normal is the word I use to denote that services are functioning as designed.

- This doesn't necessarily mean that the service can't fail.

When talking about user experience (UX) design, this is known as the "happy path".

What Should we Monitor - In a Nutshell?

- **Most important:** Anything that is not "normal". Anything "off the happy path".

If something goes wrong, it means your service is not providing the business with an expected level of functionality. This (sometimes) has a knock on effect that means that the business is unable to provide the service to a user. Bad!

- **Important in the future:** *How* your users use your service, for subsequent analysis and optimisation.

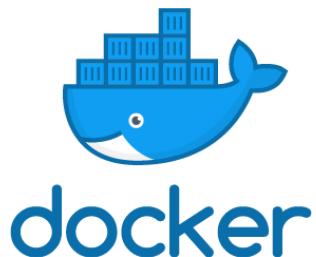
Docker

This course is focused towards Microservices. It is assumed that you have some working knowledge of:

- Microservices: Theory and implementation
- Docker: Practical use of

Given that we're packaging microservices as Docker containers, we're going to look at:

- How to monitor the resources used by containers
- How to monitor the state of a service running in the container



Kubernetes

Because we're assuming containers in a Microservices architecture, we need an orchestrator.

We're going to learn how to:

- Monitor resource usage from Kubernetes
- Use kubernetes to set up our monitoring platform



And the Rest

To achieve that, we're also going to investigate:

- Prometheus, a tool to monitor services in a Microservices architecture
- Theory: why, what, when and how to monitor. Which metrics. And a tiny bit of data science.
- Production use: Scaling, resiliency, alerting.

Many of these topics have practical examples.



102: Monitoring Docker

- We're using Docker.
- What exactly are we monitoring?
- Let's take a look!

Webservers!

Let's run a simple docker-compose file. Inside this file we're running:

- A webserver (nginx base image)
- ApacheBench (A HTTP load generator)

```
$ cd <workshop_directory>/102-monitoring-docker  
$ docker-compose up -d
```



```
version: "3"
services:
  web:
    image: nginx:1.12-alpine
    networks:
      - webnet
  load:
    image: russmckendrick/ab
    command: ["sh", "-c", "sleep 2 ; ab -k -n 1000000 -c 16 http://web/"]
    networks:
      - webnet
networks:
  webnet:
```

See What's Running

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
e198221ba4db      nginx:1.12-alpine   "nginx -g 'daemon ..."
seconds            80/tcp              102monitoringdocker_web_1
0695eb0a2636      russmckendrick/ab   "sh -c 'sleep 2 ; ..."
seconds            80/tcp              102monitoringdocker_load_1
```

- Let's remind ourselves exactly what is going on here.

Resource Usage

- Remember that docker uses Linux cgroups to keep track of and restrict a process in a container.

On linux, (since everything is a directory), we can see the mounted cgroups. (This differs slightly by distro.)

On OSX and Windows, the cgroups would be found inside the VM.

- Each container has it's own cgroup, with it's own accounting system
- Linux aggregates cgroup usage into one or more pseudo-files

`/sys/fs/cgroup/memory/docker/<longid>/`

- `memory.stat`
- `cpuacct.stat`
- etc.

So we could go through the filesystem...



docker stats

Or use a tool that reads that information for us!

A local docker tool to inspect container usage.

```
$ docker stats --no-stream
CONTAINER          CPU %               MEM USAGE / LIMIT      MEM %               NET I/O
BLOCK I/O          PIDS
ea1d4dca1a7a      58.64%              22.11MiB / 1.952GiB  1.11%              685MB /
161MB             0B / 0B              1
4bb988ddbe49      71.75%              1.828MiB / 1.952GiB  0.09%              168MB /
715MB             0B / 0B              2
```

To the Max!

Let's take this to the max!

AB only reads the DNS entry when it starts, so it will just hit one container. Let's create three AB's too, so we (possibly) hit all of them.

```
$ docker-compose down
$ docker-compose up -d --scale web=3 --scale load=3
$ docker stats --no-stream
```

CONTAINER BLOCK I/O	CPU %		MEM USAGE / LIMIT	MEM %	NET I/O
PIDS					
64e09c5481e3 16.5MB 0B / 0B	20.78%	2	3.121MiB / 1.952GiB	0.16%	70.9MB /
7cad1f19b3c3 123MB 0B / 0B	39.85%	2	1.73MiB / 1.952GiB	0.09%	28.8MB /
3f7cc4a082be 141MB 0B / 0B	37.48%	2	1.723MiB / 1.952GiB	0.09%	32.9MB /
8c8dd6ec7c32 0B / 0B	0.00%	2	1.684MiB / 1.952GiB	0.08%	914B / 0B
b79207df3f32 26.2MB 0B / 0B	40.57%	2	4.457MiB / 1.952GiB	0.22%	112MB /
653df376bb6f 17.7MB 0B / 0B	41.00%	2	3.305MiB / 1.952GiB	0.17%	76MB /

Where's my processes?

If you're on Linux, then if you do a `ps -aux`, you'll see all of the processes from the containers.

If you want to see processes from a single container (you could use `grep`) you could use `docker top`

docker top

```
$ docker top 102monitoringdocker_web_1
PID          USER          TIME        COMMAND
14000        root         0:00      nginx: master process nginx -g
daemon off;
14305        chrony       0:00      nginx: worker process
```

103: cAdvisor

cAdvisor <https://github.com/google/cadvisor> is a tool to measure performance and resource usage of a container.

- Supports a range of containers:
 - rkt
 - SystemD
 - cri-o
 - Docker
- On a range of systems:
 - Linux
 - Windows
 - OSX



How Does it Work?

- For each container, it records isolation parameters, resource usage in histogram form (see later) and network statistics.
- It exposes that data via several HTTP based APIs.
- Save measurements to a database: Elasticsearch, Kafka, InfluxDB, BigQuery, Redis, StatsD and yes, Prometheus. Oh, and stdout!
<https://github.com/google/cadvisor/tree/master/docs/storage>

Installation

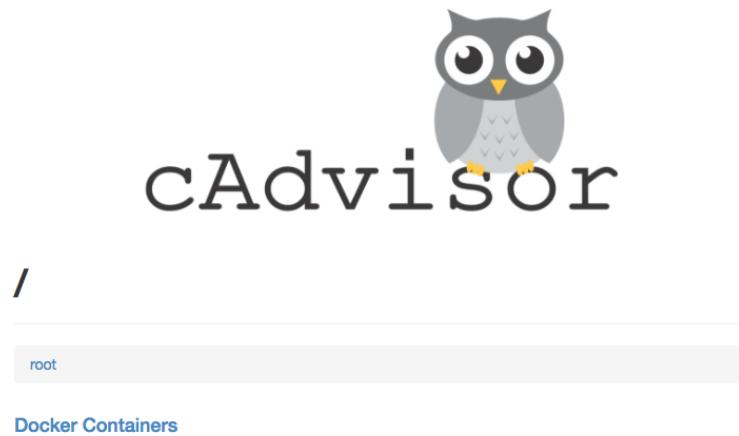
In a container of course! *It's a go application, so statically compiled binaries are available.*

```
$ docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

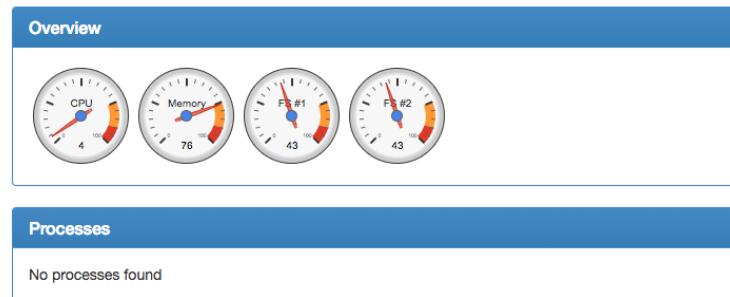
Note: This is a command for OSX. The Linux command differs subtly.

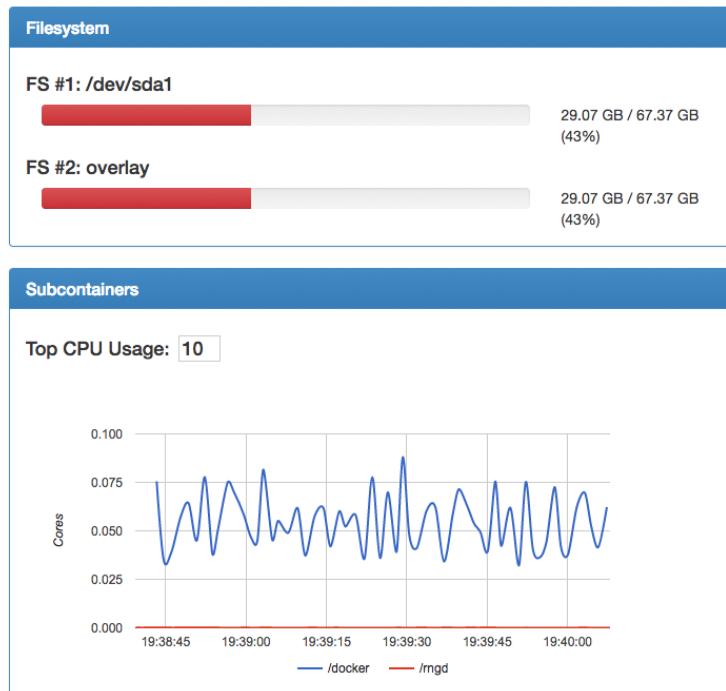
cAdvisor Web Interface

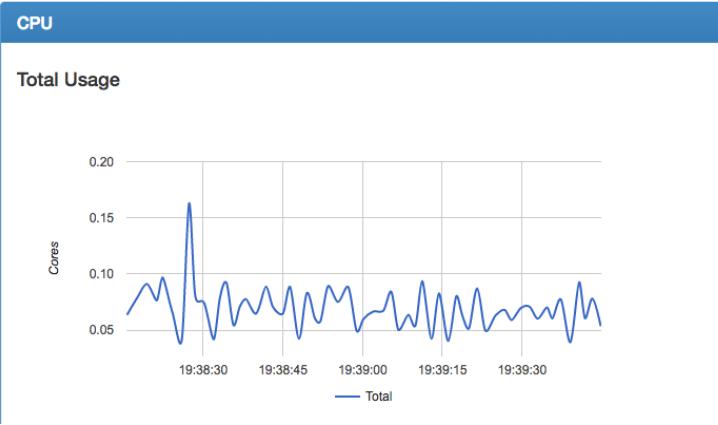
If we browse to `http://<ip address>:8080`, we will see the web interface:



Usage







Great: What's the Catch?

The catch is that this data is currently stored in Memory.

We've seen that there are various tools to expose metrics, but:

- where do we store those metrics?
- how do we use and act upon those metrics?

This is actually a more difficult problem than it first seems (*and we'll get into this later*).

Solution: The Prometheus API

Let's assume for now we're using Prometheus because it's awesome. ;-)

By default, cAdvisor has a `/metrics` endpoint (no trailing slash!).

```
$ curl http://localhost:8080/metrics
# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version,
OS version, docker version, cAdvisor version & cAdvisor revision.
# TYPE cadvisor_version_info gauge
cadvisor_version_info{cadvisorRevision="17543be",cadvisorVersion="v0.25.0",dockerVersion="17.0
ce",kernelVersion="4.9.49-moby",osVersion="Alpine Linux v3.4"} 1
# HELP container_cpu_system_seconds_total Cumulative system CPU time consumed in seconds.
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/" } 2745.43
container_cpu_system_seconds_total{id="/docker" } 1856.69
container_cpu_system_seconds_total{id="/rngd" } 4.95
container_cpu_system_seconds_total{id="/docker/73d0aecda143b9b7c6926c99dc595e4fb300d59395b4e8
10.52
container_cpu_system_seconds_total{container_label_com_docker_compose_config_hash="0ce1f21ba5
Docker Maintainers <docker-
maint@nginx.com>" ,id="/docker/c938d562e3f971190438df29d2a744414103d67f5f50cc1eeaa422da70c636
alpine",name="102monitoringdocker_web_1" } 88.33
# HELP container_cpu_usage_seconds_total Cumulative CPU time consumed per CPU in seconds.
...
```

What is that format?

Good question. It's the Prometheus Text Format.

Pros:

- Human readable
- Easy to assemble
- Easy to parse

Cons:

- Super verbose
- Parsing cost
- No metric validation

(We will look at this in detail right at the end)

Hands-on

201: Introduction to Prometheus

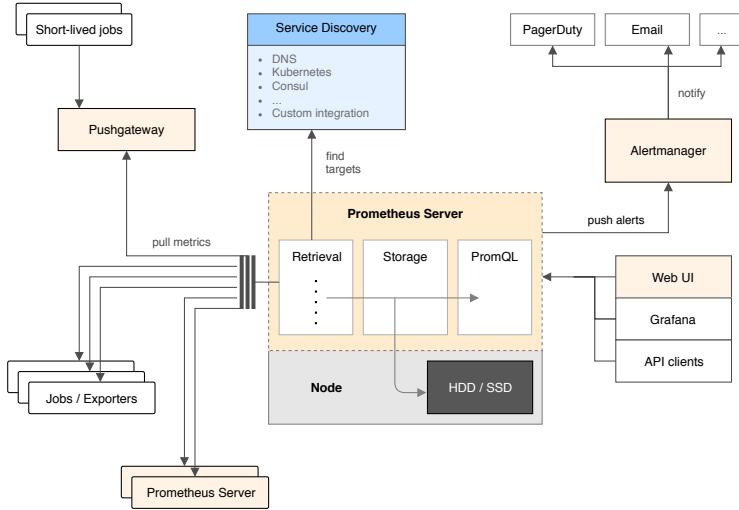
<https://prometheus.io> is an open source time series database that focuses on capturing measurements and exposing them via an API.

Key features

- Very Simple
- Pull data, not push
- No reliance on distributed storage
- No complex scalability problems

Architecture

- **Scrapes metrics** from instrumented applications, either
 - directly, or
 - via an intermediary push gateway
- **Stores** data
- **Aggregates** data and runs **rules** to generate a new time series or generate an alert
- API consumers are used to visualise and act upon this data



Where Does it Work Well?

- **Microservices:** It handles multi-dimensional metrics simply and efficiently.
- **Mission Critical:** Its inherent simplicity means it is dependable. When other parts of your system are down, Prometheus will still be running.



Where Does it Not (try to) Work Well?

- **Accuracy:** Prometheus scrapes data. If you have services that require accuracy (e.g. per usage billing) then Prometheus is not a good fit. Scrapes are not guaranteed to occur.
- **Non HTTP Systems:** There are other encodings, but HTTP is dominant. If you don't use HTTP (e.g. grpc) you're going to have to add some code to expose the metrics. (See [go-grpc-prometheus](#))

Prometheus vs...

You may have some experience with other systems. Let's compare...

... Graphite

- Just a time series database, not a monitoring solution out of the box.
- Common to only store aggregates, not raw time series.
- Has expectations for time of arrival that doesn't fit well with Microservices.

... InfluxDB

- Quite similar
- Commercial offering is distributed, which means you have to manage another distributed system.
- Better at event logging
- More complex than Prometheus

... OpenTSDB

- Hadoop/HBase based, so distributed complexity
- Just a time series database, not a monitoring solution out of the box.
- A possibility if you're already managing Hadoop-based systems

... Nagios

- No notion of labels or query language
- Host based
- Out of the box monitoring
- Possibly ok (but expensive) form of black-box monitoring. Not really suited towards microservices.

... NewRelic

- Fought hard to keep up with Microservices
- Complex
- Focused more on the business side
- Probably a better option than Nagios
- Most features can be replicated with open source equivalents

Generally

I think Prometheus' simplicity is key. All the previous examples are seriously complex in comparison.

But it is not business focused. It's developer focused.

Data Model

All of the data is stored as a *time series*. I.e. a measurement with a timestamp.

Measurements are known as *metrics*.

Each time series is uniquely identified by a *metric name* and a set of *key-value pairs*,
a.k.a. *labels*.

How Is the Database Structured?

The *metric name* is used to denote the feature that is being measured.

E.g. `http_requests_total` - total number of HTTP requests.

Must match the regex: `[a-zA-Z_:][a-zA-Z0-9_:]^*`

Labels represent multiple dimensions of a *metric*.

A combination of a metric name and a label yields a single metric.

E.g. `http_requests_total{service=orders}` – total number of HTTP requests for the orders (value) service (key).

Must match the regex: `[a-zA-Z_][a-zA-Z0-9_]*`

An observation (they call it a sample!) is a combination of a `float64` value and a millisecond precision timestamp.

Prometheus does not store strings! It is not for logging!

Given a metric name and key-value labels, the following format is used to address metrics:

```
<metric name>{<label name>=<label value>, ...}
```

Types of Metrics

Prometheus caters for different types of measurements by having four different types of metrics.

All types are eventually flattened to untyped time series.

Counter

A cumulative metric that only ever increases.

For example:

- Requests served
- Tasks completed
- Errors occurred

Should not be used for metrics that can also go down. (e.g. number of threads)

Gauge

A metric that can arbitrarily go up and down.

E.g.

- Temperature
- Memory usage
- Live number of users

Histogram

Places an observation into configurable buckets.

E.g.

- Request duration
- Response size

A histogram will create several metrics and has some special helper functions (which we will see later)

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

Summary

Basically a pre-configured Histogram. You select the quantiles and aggregation in advance to reduce the server-side calculation burden.

Recommendations:

- Generally don't use this. Use a histogram
- Only use for static, well defined metrics
- Consider its use when performance is a concern

How Does Prometheus Obtain Metrics?

One of the key's to Prometheus' simplicity.

Many other monitoring solutions expect to be handed data. (Push model)

Prometheus reaches out to services and scrapes data itself. (Pull model)



Push vs. Pull Models

Pull Advantages:

- Easy to tell if target is down
- Manually inspect health via a web browser
- Ease of development (just point your laptop at the service)
- Very little load on services
- Services aren't affected by load on the monitoring system

Push Advantages:

- Possibly More secure
- Store events
- Accuracy
- May work in firewalled setups

FAQ 1: Pull Doesn't Scale!

- Nagios uses a pull model, but it uses complex scripts to fetch data and assert state. Because it was so slow, people had to use update rates of 1 per 5 minutes, or so.
- Prometheus uses HTTP and only HTTP. It spins up HTTP connections like lightning, thanks to Go's GoRoutines. The bottleneck is the size of the monitoring data and writing that to disk. Not the connections.
- Proven performance: Given a 10-seconds scrape interval and 700 time series per host, this allows you to monitor over 10,000 machines from a single Prometheus server.
- Simple sharding mechanism allows you to scale past a single server.

Using Metrics

Prometheus is permissive; you can use any metrics or names that you want.

But it is important to think about why you are creating a metric and what is the goal of using a metric.

Instances and Jobs

An individual scrape is called an *instance*.

A collection of replicas of that instance are called a *job*.

E.g.

- job: frontend
 - instance 1: frontend-1:8080
 - instance 2: frontend-2
 - instance 3: 5.6.7.8:5670
 - instance 4: 5.6.7.8:5671

How are names generated?

When Prometheus scrapes an instance, it automatically adds certain labels to the scraped time series:

- `job` : The job name that the target belongs to
- `instance` : The `host:port` combination of the target's url that was scraped.

These can be overwritten by including them in the `/metrics` and altering the `honor_labels` configuration option.

Each scrape also produces metrics about the scrape:

- `up{job="", instance=""}`: 1 if the instance is healthy, i.e. reachable, or 0 if the scrape failed.
- `scrape_duration_seconds{job="", instance=""}`: duration of the scrape.
- `scrape_samples_post_metric_relabeling{job="", instance=""}`: the number of samples remaining after metric relabeling was applied.
- `scrape_samples_scraped{job="", instance=""}`: the number of samples the target exposed.

The `up` time series is particularly useful for availability monitoring.

For all other metrics, **you** decide the metric name and labels.



How To Pick Names

Organisations should generate a naming convention so that:

- You know what to call your metrics during development
- Users can understand what your metric means with just a glance

Labels should be chosen so that they differentiate the context of the metric.

Everyone should use the same convention.

Metric Name Best Practices

The following is a list of best practices. Following these rules should help you avoid common pitfalls ([read more](#)).

Consistent Domain-based Prefixes

A prefix is the first word in a metric name. Often called a `namespace` by client libraries.

Choose a prefix that defines the **domain** of the measurement.

Examples:

- HTTP related metrics should all have a prefix of `http`: `http_request_duration_seconds`
- Application specific metrics should refer to a domain: `users_total`
- Process level metrics are exported by many libraries by default:
`process_cpu_seconds_total`

Consistent Units

Use SI (International System of Units) units.

E.g.

- Seconds
- Bytes
- Metres

Not milliseconds, megabytes, kilometres, etc.

A Single Unit Per Metric

Do not mix metrics. E.g.:

- One instance that reports seconds with another reporting hours
- Aggregate metrics. E.g. bytes/second. Use two metrics instead.

Suffix Should Describe the Unit

In plural form.

Examples:

- `http_request_duration_seconds`
- `node_memory_usage_bytes`
- `http_requests_total` (for a unit-less accumulating count)
- `process_cpu_seconds_total` (for an accumulating count with unit)

Mean the Same Thing

The metric name should mean the same thing across all label dimensions.

E.g.

The metric `http_requests_total` means the same thing, despite having different labels as below:

- `http_requests_total{status=404}`
- `http_requests_total{status=200}`
- `http_requests_total{status=200, path="/users/"}`

Testing That Names Make Sense

One tip provided by the authors is very useful:

either the sum or average over all dimensions should be meaningful (though not necessarily useful)

E.g.

- the capacity of various queues in one metric is good, while mixing the capacity of a queue with the current number of elements in the queue is not.

Generally, split metrics into single unit types.

Label Best Practices

- Used to differentiate context

`api_http_requests_total` - differentiate request types: `type="create|update|delete"`

`api_request_duration_seconds` - differentiate request stages:
`stage="extract|transform|load"`

Do not put context in metric names as it will cause confusion.

Also, when we perform aggregations, it's really nice to see the aggregations for different contexts.

E.g. does one stage take significantly longer than the others?

1 Label === 1 Dimension

Each new key-value pair represents a new dimension.

If you pick a key that has many values, this dramatically increases the amount of data that must be stored.

For example, **DO NOT STORE:**

- ID's
- Email addresses
- Timestamps of any kind
- Anything unbounded

Generally, **All key values must be a bounded set.**

No workshop here. Next we'll look at spinning up Prometheus.
Probably time for a break. :-)



202: Using Prometheus

This section is all about starting and configuring Prometheus.

Hopefully you're excited to get your hands dirty. At the end there's a big workshop.

Up and Running

How to install? You know the drill, Docker!

```
$ docker run -p 9090:9090 prom/prometheus
```

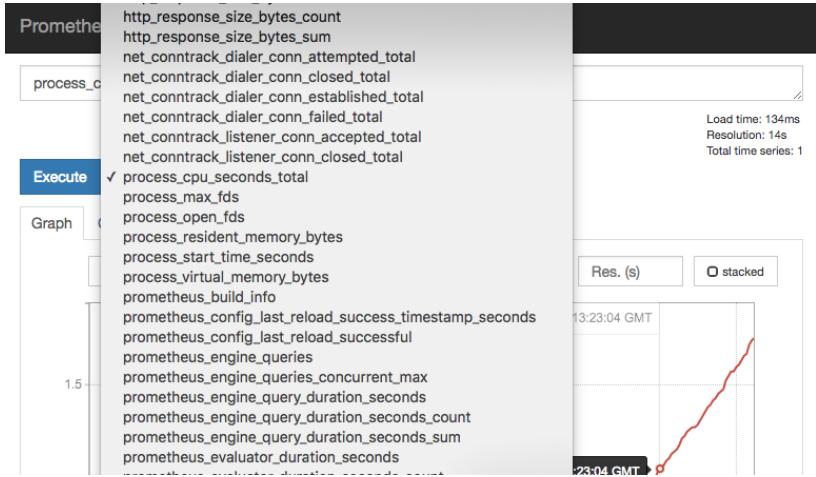
Prometheus UI

When you first visit `http://<public ip address>:9090` you will be redirected to the **Graph** page. (Not a graph! A plot!)

Here you can type your *query* and view the results as a table or a plot.

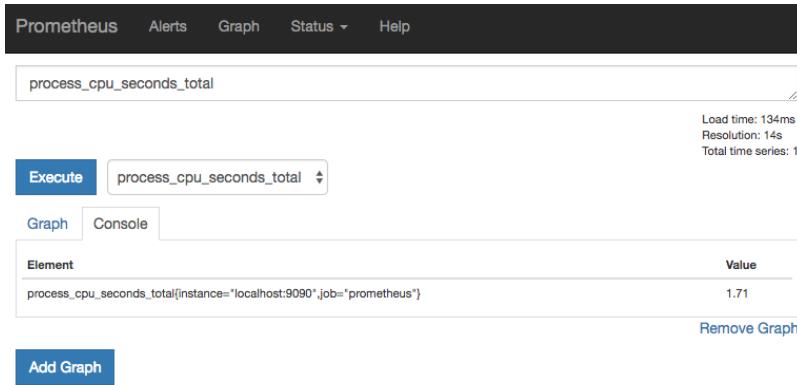
The screenshot shows the Prometheus UI interface. At the top, there is a dark navigation bar with tabs for 'Prometheus', 'Alerts', 'Graph', 'Status', and 'Help'. Below the navigation bar is a search bar labeled 'Expression (press Shift+Enter for newlines)'. To the right of the search bar is a blue 'Execute' button and a dropdown menu with the placeholder '- insert metric at cursor -'. Below the search bar, there are two tabs: 'Graph' (which is selected) and 'Console'. Under the 'Graph' tab, there is a table with one row. The table has two columns: 'Element' and 'Value'. The 'Element' column contains the text 'no data'. To the right of the table is a blue 'Remove Graph' button. At the bottom left of the main area is a blue 'Add Graph' button.

Selecting Metrics



Results as a Table

This is the reading of this metric at this moment.



Results as a Plot

These are the readings of the metric over a time period.

Status

The status tab shows the current configuration of Prometheus. It is very useful for debugging configuration.

A screenshot of the Prometheus web interface showing the 'Status' tab selected. The main content area is titled 'Configuration' and displays the following YAML configuration:

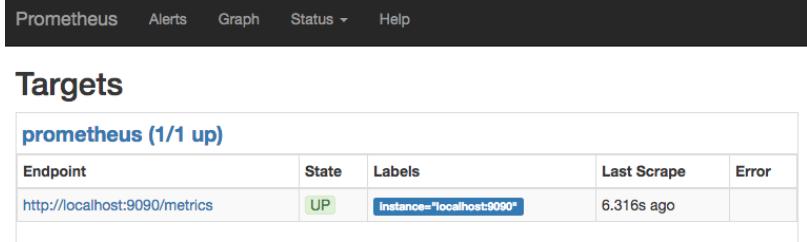
```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 15s
alerting:
  alertmanagers:
  - static_configs:
    - targets: []
      scheme: http
      timeout: 10s
scrape_configs:
- job_name: prometheus
  scrape_interval: 15s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: http
  static_configs:
  - targets:
    - localhost:9090
```

To the right of the configuration, a dropdown menu is open under the 'Status' tab, listing:

- Runtime & Build Information
- Command-Line Flags
- Configuration
- Rules
- Targets

Target Status

The Status->Targets option shows the scraping status at this moment. This is very useful for debugging service status and scrape settings.



A screenshot of the Prometheus web interface. The top navigation bar includes links for Prometheus, Alerts, Graph, Status (with a dropdown arrow), and Help. Below the navigation is a section titled "Targets". Under "Targets", there is a header "prometheus (1/1 up)". A table lists one target endpoint: "http://localhost:9090/metrics". The table columns are Endpoint, State, Labels, Last Scrape, and Error. The "State" column shows "UP" in a green button, and the "Labels" column shows "instance=\"localhost:9090\"". The "Last Scrape" column shows "6.316s ago".

prometheus (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	6.316s ago	

Alerts

Not Yet!

We'll talk about alerts later.

Help

This redirects to the Prometheus documentation.

Prometheus Configuration

Prometheus is configured via command line flags and a configuration file.



Configuration File

The configuration file contains the settings that you will alter on a day-to-day basis.
Like adding new endpoints to scrape, etc. ([more info](#))

Global Configuration

Parameters that are valid in all configuration contexts. Also contain some defaults that can be overridden in other sections. These are the most important global settings.

```
global:  
  # How frequently to scrape targets by default.  
  [ scrape_interval: <duration> | default = 1m ]  
  
  # How long until a scrape request times out.  
  [ scrape_timeout: <duration> | default = 10s ]  
  
  # How frequently to evaluate rules.  
  [ evaluation_interval: <duration> | default = 1m ]  
  
  # A list of scrape configurations.  
  scrape_configs:  
    [ - <scrape_config> ... ]
```

Scrape Configuration

The scrape configuration is the core of the configuration file. Here you specify how, where and when Prometheus should scrape your metrics endpoint.

```
# The job name assigned to scraped metrics by default.  
job_name: <job_name>  
  
# How frequently to scrape targets from this job.  
[ scrape_interval: <duration> | default = <global_config.scrape_interval> ]  
  
# Per-scrape timeout when scraping this job.  
[ scrape_timeout: <duration> | default = <global_config.scrape_timeout> ]  
  
# The HTTP resource path on which to fetch metrics from targets.  
[ metrics_path: <path> | default = /metrics ]  
  
# Configures the protocol scheme used for requests.  
[ scheme: <scheme> | default = http ]
```

Scrape HTTP Settings and Security Credentials

```
# Optional HTTP URL parameters.
params:
  [ <string>: [<string>, ...] ]

# Sets the `Authorization` header on every scrape request with the
# configured username and password.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]

# Sets the `Authorization` header on every scrape request with
# the configured bearer token. It is mutually exclusive with `bearer_token_file`.
[ bearer_token: <secret> ]

# Sets the `Authorization` header on every scrape request with the bearer token
# read from the configured file. It is mutually exclusive with `bearer_token`.
[ bearer_token_file: /path/to/bearer/token/file ]

# Configures the scrape request's TLS settings.
tls_config:
  [ <tls_config> ] # See the documentation for specifics.

# Optional proxy URL.
[ proxy_url: <string> ]
```

Discovery Configuration

Here you specify how and where Prometheus should resolve IP addresses.

```
# List of Kubernetes service discovery configurations.  
kubernetes_sd_configs:  
  [ - <kubernetes_sd_config> ... ]  
  
# List of labeled statically configured targets for this job.  
static_configs:  
  [ - <static_config> ... ]
```

There are also discovery mechanisms for: Azure, Consul, DNS, EC2, OpenStack, file, GCE, Marathon, Nerve, ServerSet (Zookeeper), Triton

Static Config

This allows you to specify a list of hard-coded targets and common label for them. You wouldn't normally use this unless you are testing.

```
# The targets specified by the static config.  
targets:  
  [ - '<host>' ] # DNS hostname or IP address followed by port number.  
  
# Labels assigned to all metrics scraped from the targets.  
labels:  
  [ <labelname>: <labelvalue> ... ]
```

Kubernetes Config

These options allow you to specify what it is you want to discover. The `role` setting must be one of: `endpoints`, `service`, `pod`, or `node`. Typically your default k8s config will have four jobs each pointing to one of these roles.

There are also options to allow you to connect to a remote K8s cluster (not common).

```
# The API server addresses. If left empty, Prometheus is assumed to run inside
# of the cluster and will discover API servers automatically and use the pod's
# CA certificate and bearer token file at /var/run/secrets/kubernetes.io/serviceaccount/ .
[ api_server: <host> ]

# The Kubernetes role of entities that should be discovered.
role: <role>

# Optional namespace discovery. If omitted, all namespaces are used.
namespaces:
  names:
    [ - <string> ]

# Optional authentication information used to authenticate to the API server.
# This is the same as the http security configuration above.
```

And More...

In my experience, they are the main settings you'll have to alter.

But there are a lot more. If you're doing this for real please do look at the [documentation](#).

CLI Flags

The CLI flags are more related to the operation of the binary. E.g. information about storage drivers, logging level, etc.

```
$ docker run -p 9090:9090 prom/prometheus -h
usage: prometheus [<flags>]

The Prometheus monitoring server

Flags:
  -h, --help               Show context-sensitive help (also try
                           --help-long and --help-man).
  ...
```

Important CLI Options

Here are some important cli parameters. There are more relating to how the data is stored and options for hosting Prometheus behind a reverse proxy.

```
--config.file="prometheus.yml"          Prometheus configuration file path.  
--web.listen-address="0.0.0.0:9090"      Address to listen on for UI, API, and  
                                         telemetry.  
--storage.tsdb.path="data/"            Base path for metrics storage.  
--storage.tsdb.retention=15d           How long to retain samples in the storage.  
--log.level=info                      Only log messages with the given severity or  
                                         above. One of: [debug, info, warn, error]
```

Example Configuration file: Simple

This is the simplest example you can have. A configuration scrape itself.

Go Scrape Yourself!

```
global:  
  scrape_interval:      15s  
scrape_configs:  
  - job_name: 'prometheus'  
    scrape_interval: 5s  
    static_configs:  
      - targets: ['localhost:9090']
```

Example Configuration with Docker

1. Write the configuration to a file
2. Pass the file into the container as a volume.

```
$ docker run -p 9090:9090 -v ${PWD}/prometheus.yml:/etc/prometheus/prometheus.yml  
prom/prometheus
```

When you run that, you'll see that we have Prometheus as a target. (This is the same as the default configuration!)

The screenshot shows the Prometheus web interface. At the top, there is a dark navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below this, the main content area has a title 'Targets' and a sub-section header 'prometheus (1/1 up)'. A table displays the status of a single target:

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	1.671s ago	

Example Configuration file: Our Custom App

```
global:  
  scrape_interval: 5s  
scrape_configs:  
  - job_name: 'prometheus'  
    static_configs:  
      - targets: ['localhost:9090']  
  - job_name: 'python-app'  
    static_configs:  
      - targets: ['python-app:5000']
```

Before we add the app

The screenshot shows the Prometheus Targets page. At the top, there is a navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below the navigation bar, the title "Targets" is displayed. The page lists two target groups: "prometheus (1/1 up)" and "python-app (0/1 up)".

prometheus (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	1.337s ago	
python-app (0/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:5000/metrics	DOWN	instance="localhost:5000"	1.548s ago	Get http://localhost:5000/metrics: dial tcp 127.0.0.1:5000: getsockopt: connection refused

After we add the app

```
$ docker run -p 5000:5000 -d --name python-app vect0r/python-app:latest
$ docker run -p 9090:9090 --link python-app -v
${PWD}/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

Prometheus Alerts Graph Status ▾ Help

Targets

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	3.061s ago	

python-app (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://python-app:5000/metrics	UP	instance="python-app:5000"	2.534s ago	

Example Configuration file: cAdvisor

```
global:  
  scrape_interval: 5s  
scrape_configs:  
  - job_name: 'prometheus'  
    static_configs:  
      - targets: ['localhost:9090']  
  - job_name: 'python-app'  
    static_configs:  
      - targets: ['python-app:5000']  
  - job_name: 'cadvisor'  
    static_configs:  
      - targets: ['cadvisor:8080']
```

And using the same docker command as we saw in the previous section results in:

cAdvisor Target

Don't forget the extra link.

```
$ docker run -p 9090:9090 --link cadvisor --link python-app -v  
${PWD}/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

Prometheus Alerts Graph Status ▾ Help

Targets

cadvisor (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://cadvisor:8080/metrics	UP	instance="cadvisor:8080"	4.517s ago	

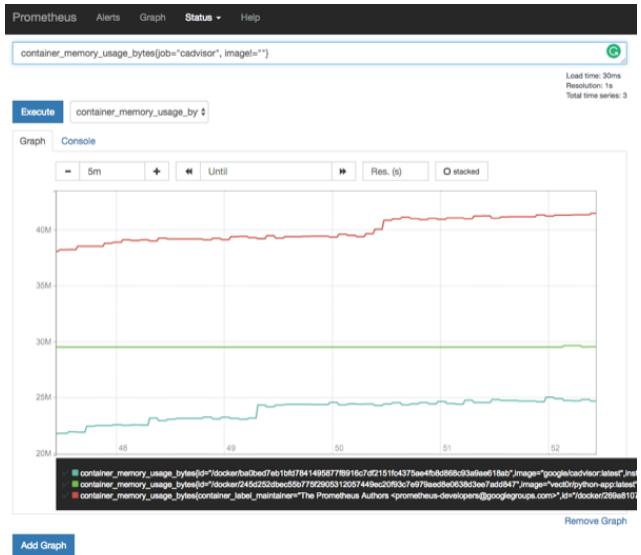
prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	3.299s ago	

python-app (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://python-app:5000/metrics	UP	instance="python-app:5000"	2.774s ago	

cAdvisor Plot



What's the Catch: Part Duex

- Just a database
- Plots are fine for development, but we can't continue using this for operational visualisation
- No alerting (yet!)
- Need to create lots of `/metrics` APIs

Hands on!

Finally, some hands on time with Prometheus!

203: Monitoring Theory

After the practical excitement of using Prometheus, let's go back to the theory.

- What do we mean by monitoring?
- Why do we need it?
- What are our goals of monitoring?
- What happens, really?



Logging vs. Tracing vs. Instrumentation

These are completely different, and serve different purposes.

More information:

- [Logging V Instrumentation – Peter Bourgon](#)
- [Monitoring Performance in Microservice Architectures – Lukasz Gumiński](#)

Logging

- Applications produce logs. Logs represent state transformations within an application.
- Obtaining, transferring, storing and parsing logs is expensive.
- Log only what is necessary to act upon. Log only actionable information.
- This means: panic-level information for humans, structured data for machines (Question whether this is really necessary).

Instrumentation (a.k.a. Monitoring)

- Instrumentation represents the use of an application of system. It is used for diagnostic purposes.
- Instrumentation is cheap and structured. (We've seen prometheus is simply a set of key-value pairs)
- The more data you have, the more useful it becomes.
- Hence, instrument everything that is meaningful in the domain of your application.



Tracing

- A trace represents a single user's journey and use of the system as a whole. Similar to logging, but different requirements.
- Often used for performance optimisation.
- Only makes sense when systems become complex enough
- Add significant complexity to both code and architecture
- Same performance concerns as logging; it's expensive.

So, Monitoring != Logging != Tracing.

Site Reliability Engineering

- Championed by Google.
- Essentially a specific implementation of DevOps.

More information:

- [Google's SRE Book - Free](#)

Responsibilities

A Site Reliability Engineer is responsible for:

- availability
- latency
- performance
- efficiency
- change management
- monitoring
- emergency response
- capacity planning

Very similar to a traditional SysAdmin.

Key Differences

- SRE's are software engineers.
- When combining with DevOps, SREs *are* the engineers building the platform.
- Focused towards service availability as opposed to feature set or performance
- Hence, significant efforts are spent automating and implementing monitoring solutions.

Risk

- Quantify risk: how important is a service to a business?
 - What is expected?
 - Is availability tied to revenue?
 - Is it a free service?
 - Competitive advantage?
 - Consumers or Enterprise?
- Quantify performance:
 - Similar to above.

Ideally, Engineers want to be given clear objectives to which their service must meet.
These are called *service metrics*...

Service Metrics

Service Metrics can be split up into:

- Service Level Objectives (SLO)
- Service Level Indicators (SLI)

These then go on to create a *Service Level Agreement* (SLA) which externally defines the availability and performance of your system to your customers.

Service Level Objectives

Represent material impacts to the business.

Might include:

- Performance metrics, e.g. less than 100ms latency
- Availability metrics, e.g. available for 99.9% of the time (three nines)
- Durability, e.g. data is stored for one year with 99.999% availability)

Google Compute Engine's SLA states 99.95% availability.

Key takeaway: defined SLO's state that downtime is ok.

99.9% availability equates to this amount of downtime:

- **Daily:** 1m 26.4s
- **Weekly:** 10m 4.8s
- **Monthly:** 43m 49.7s
- **Yearly:** 8h 45m 57.0s

Aside: Don't be Too Good

When services approach 99.9%+ levels of uptime, users often assume services will always work.

Google talks about one of their internal tools called Chubby, which is a distributed lock. People started using it as a dependency, because it never went down.

Now, Google sticks to their SLO budgets and intentionally kills the service to weed out any services that have accidentally used it as a dependency!

Service Level Indicators

- Sometimes there are indicators that directly measure an SLO.

For example, we can measure availability directly.

- Other times, we have to use a proxy that approximately equates to a SLO.

For example, we can time requests, but that doesn't take latency between the user and the server into account.

Make sure you have indicators for all of your SLOs

Toil

Toil is the term given to "doing work that people don't want to do".

For example, being on call or dealing with an incident.

People don't want to do that, so it is worth investing time into automating these tasks or preventing them from happening. Toil adds stress, lowers morale, slows progress, promotes attrition.

If possible, toil should be measured quantitatively so that engineering effort can be optimised.



What is Normal?

If a human operator needs to touch your system during normal operations, you have a bug. The definition of normal changes as your systems grow.

Carla Geisser, Google SRE

But what is normal?

Largely a philosophical debate. But must include what is expected of your system.

Normal is: "usual, typical, or expected" - OED.

What Should we Monitor?

The question of *what* to monitor is a good one. But it depends on context.

From a development perspective, more is better. It is very easy to filter/aggregate out any unnecessary metrics.

However, from an operational standpoint **be careful not to burden users with too much information.**

Microservice Level Indicators

Service Level Indicators are of primary concern. These should be displayed front-and-centre.

But Microservice Level Indicators ensure that your microservice is operating normally.

Generally, metrics fall into two groups:

- Service usage
- Resource usage

Resource Usage

Simple. Just monitor the resource usage of your services.

Ensure that services don't unexpectedly use more resources. E.g. Memory leak.

Service Usage

Several different rules of thumb.

RED Metrics:

- Request Rate: The number of requests per second
- Request Errors: Number of failed requests per second
- Request Duration: Distribution of the amount of time requests take

The RED Method: key metrics for microservices architecture

Which is a simplification of the *Four Golden Rules*...

Four Golden Rules

- Latency: Important to measure latency of 200 and 400/500 errors separately.
- Traffic: Measure of demand placed on a system
- Errors: Rate at which requests fail
- Saturation: How "full" the service is (e.g. memory/cpu saturation, does the service degrade before saturation?)

[Google SRE Book](#)

Use Method

- Resources: Cpu/Ram/etc.
- Utilisation: Average time that the resources was busy
- Saturation: The degree to which the resource has extra work, which it can't service
- Errors: Number of errors

The USE method

Personally I think this mixes the resource monitoring and service monitoring. I think they should be separate monitoring tasks.

Being On-Call

Being on call during an emergency is very stressful and hard work (much like training!).

But it is necessary to ensure a service runs smoothly.

To prevent burnout/attrition/loss of moral, on call incidents should be limited. Google suggests no more than 25% of an Employee's time.

Alert Fatigue

The Broken Windows theory: maintaining and monitoring neighbourhoods for petty crimes helps prevent more serious crimes from occurring.

In a sea of alerts, it's impossible to see those that will breach the SLO.

Low priority alerts that frequently bother the on-call Engineer disrupts productivity and introduces fatigue.

Serious alerts can be treated with less attention than necessary.

Reduce the number of alerts for a single incident. There's no need to swamp a system with many alerts for the same underlying cause.

Emergency Response

Your systems will break. What do you do? Follow a procedure.

- Don't Panic!
- Observe: Find and use the monitoring and logging systems
- Orient: Establish what has gone wrong
- Decide: Plan a course of action
- Act: Enact the plan
- Review: Record what happened for posterity. Learn from mistakes.

More information:

- [OODA loop](#)

Keep Your Users Informed

Users are not stupid.

They will resort to twitter to complain that a service is down.

It is always better to be open and honest about availability.

Always keep them in the loop.

Incident Management

- **Prepare:** Develop incident management procedures in advance
- **Prioritise:** "Stop the bleeding", restore the service and preserve the evidence
- **Trust:** Give full autonomy to all participants
- **Introspect:** If you start feeling stressed or panicky, ask for help
- **Consider Alternatives:** When stressed, the instinctive decision might not be the best

Postmortems

After an incident, hold a postmortem to review the incident.

- Keep a live document to collaborate and add information

Key tasks of a postmortem:

- Was key data collected for posterity?
- Are impact assessments complete?
- Was the root cause established?
- Is the action plan appropriate?
- Are fixes at an appropriate priority?
- Has the outcome been shared with stakeholders?

Chaos Engineering

A developing discipline where systems are intentionally harmed to simulate failure.

- Practice incident failure
- Test system's resiliency
- Identify weaknesses (e.g. improper timeouts, retry storms, cascading failures, etc.)
- Builds confidence in a system



The Future

- Stateful vs. Semantic monitoring

We're currently monitoring the state of a system. Semantic monitoring is about modelling a system and alerting when the model deviates from the norm.

E.g. We don't monitor the health of cells in the heart to see if the heart is healthy. Instead, we listen to the electrical impulses that a heart makes when it beats. If the resultant signal is not normal, we can say that the heart is not healthy.

[More info](#)

- Automation

Data Science will creep into monitoring systems. It already has at a low level, but common tasks like anomaly detection will increasingly be performed automatically. Marketing likes to call this AI. :-/

[More info](#)

203: Instrumenting an App in...

This sections is all about adding Prometheus instrumentation code to your apps.

We will cover a few languages, but feel free to use another language.

This is a very open practical.

General Setup

I think it is simplest if we code the same thing in different languages.

Then we can see all the boilerplate Java needs! :-D

So let's create a simple webserver to host an API and a /metrics endpoint.

We'll use the various client libraries to add instrumentation:

<https://prometheus.io/docs/instrumenting/clientlibs/>

Client Library Implementations

There are a range of implementations. **Go**, **Java**, **Python** and **Ruby** are official, but the third party ones are perfectly good too. The implementation interface is very small and simple, so it's easy for the client libraries to keep up to date.

- Bash
- C++
- Common Lisp
- Elixir
- Erlang
- Haskell
- Lua for Nginx
- Lua for Tarantool
- .NET / C#
- Node.js
- PHP

Python

I'll start with Python because I think it's one of the easiest languages to understand, even for people that haven't used Python before.

I have provided an example Python application that is instrumented with Prometheus and uses Flask as a webserver.

You can find the code here: <https://github.com/phillwinder/prometheus-python>

Let's take a quick look at the code (all the code doesn't fit on one slide)...

Python imports and standard Flask stuff. Note the Prometheus client.

```
import random
import time

from flask import Flask, render_template_string
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)
```

Create a Counter, a Gauge and a Histogram.

```
# A counter to count the total number of HTTP requests
REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method',
'endpoint'])

# A gauge (i.e. goes up and down) to monitor the total number of in progress requests
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')

# A histogram to measure the latency of the HTTP requests
TIMINGS = Histogram('http_requests_latency_seconds', 'HTTP request latency (seconds)')
```

The routes are just combinations of this. Note the use of labels in the counter and the helpers to do the timing and gauge incrementing/decrementing.

```
# Standard Flask route stuff.  
@app.route('/')  
# Helper annotation to measure how long a method takes and save as a histogram metric.  
@TIMINGS.time()  
# Helper annotation to increment a gauge when entering the method and decrementing when  
leaving.  
@IN_PROGRESS.track_inprogress()  
def hello_world():  
    REQUESTS.labels(method='GET', endpoint="/").inc() # Increment the counter  
    return 'Hello, World!'
```

Finally, we just need to expose the metrics endpoint, instrumented of course!

```
@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def metrics():
    REQUESTS.labels(method='GET', endpoint="/metrics").inc()
    return generate_latest(REGISTRY)

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Running the Python example

You know the drill:

```
docker run -p 5000:5000 --name python-app philwinder/prometheus-python
```

Then we can hit:

- GET /
- GET /hello/<your name>
- GET /slow
- GET /metrics

```
$ curl localhost:5000/slow
<h1>Wow, that took 0.8827602169609146 s!</h1>%
$ curl localhost:5000/hello/phil
<b>Hello phil</b>!%
```

And the metrics endpoint (which isn't going to fit on one slide)...

```
$ curl localhost:5000/metrics
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 101838848.0
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 25690112.0
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1510221818.16
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 1.03
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 6.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1048576.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="6",patchlevel="3",version="3.6.3"} 1.0
# HELP http_requests_total Total HTTP Requests (count)
# TYPE http_requests_total counter
http_requests_total{endpoint="/slow",method="GET"} 15.0
http_requests_total{endpoint="/metrics",method="GET"} 2.0
http_requests_total{endpoint="/",method="GET"} 1.0
http_requests_total{endpoint="/hello/<name>",method="GET"} 1.0
# HELP http_requests_inprogress Number of in progress HTTP requests
# TYPE http_requests_inprogress gauge
http_requests_inprogress 1.0
# HELP http_requests_latency_seconds HTTP request latency (seconds)
```

Java Spring-Boot

I like to use Spring Boot here, because it reduces the amount of boilerplate for a simple Java Application. However, consider whether Spring Boot provides enough room for customisation for more complex tasks.

I have provided an example that is using a version of the Java "SimpleClient" for Prometheus that uses Spring Boot.

You can find the code here: <https://github.com/philwinder/prometheus-java-spring-boot>

Let's take a look at the code.

Maven POM file

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient</artifactId>
        <version>0.1.0</version>
    </dependency>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient_common</artifactId>
        <version>0.1.0</version>
    </dependency>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient_spring_boot</artifactId>
        <version>0.1.0</version>
    </dependency>
</dependencies>
```

Application.java Class

Imports

```
package com.github.philwinder.prometheus.java.springboot;  
  
import io.prometheus.client.Histogram;  
import io.prometheus.client.spring.boot.EnablePrometheusEndpoint;  
import io.prometheus.client.spring.boot.EnableSpringBootMetricsCollector;  
import org.springframework.boot.*;  
import org.springframework.boot.autoconfigure.*;  
import org.springframework.web.bind.annotation.*;
```

Spring boot annotations and Prometheus annotations.

```
// Standard Spring boot annotation
@SpringBootApplication
// Add a Prometheus metrics endpoint to the route `/prometheus`. `/metrics` is already taken
// by Actuator.
@EnablePrometheusEndpoint
// Pull all metrics from Actuator and expose them as Prometheus metrics. Must have
// permission to do this.
@EnableSpringBootMetricsCollector
// For route annotations below.
@RestController
```

Main Application class and instantiating the histogram. This must be done only once!

```
// Main application class. Keep it in one file for simplicity.  
public class Application {  
    // A Histogram Prometheus Metric  
    static final Histogram requestLatency = Histogram.build()  
        .name("http_request_duration_seconds")  
        .help("HTTP request duration (seconds).")  
        .register(); // Register must be called to add it to the output
```

Route mapping and start and stop the histogram timer.

```
// Standard MVC style route mapping
@RequestMapping("/")
// Note that we could have used the Spring AOP annotation @PrometheusTimeMethod too.
String root() {
    // Start the histogram timer
    Histogram.Timer requestTimer = requestLatency.startTimer();
    try {
        return "Hello Spring Boot World!";
    } finally {
        // Stop the histogram timer.
        requestTimer.observeDuration();
    }
}
```

Standard boilerplate. Move along. Nothing to see here.

```
// Standard Spring boot main.  
public static void main(String[] args) throws Exception {  
    SpringApplication.run(Application.class, args);  
}
```

Running the Java example

```
docker run -p 8080:8080 --name spring-boot-app philwinder/prometheus-java-spring-boot
```

Then we can hit:

- GET /
- GET /prometheus
- All the other Spring Boot endpoints. See their docs.

```
$ curl localhost:8080/  
Hello Spring Boot World!%
```

And the metrics endpoint (which isn't going to fit on one slide)...

```
$ curl localhost:8080/prometheus
```

```
# HELP http_request_duration_seconds HTTP request duration (seconds).
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.005"} 1.0
http_request_duration_seconds_bucket{le="0.01"} 1.0
http_request_duration_seconds_bucket{le="0.025"} 1.0
http_request_duration_seconds_bucket{le="0.05"} 1.0
http_request_duration_seconds_bucket{le="0.075"} 1.0
http_request_duration_seconds_bucket{le="0.1"} 1.0
http_request_duration_seconds_bucket{le="0.25"} 1.0
http_request_duration_seconds_bucket{le="0.5"} 1.0
http_request_duration_seconds_bucket{le="0.75"} 1.0
http_request_duration_seconds_bucket{le="1.0"} 1.0
http_request_duration_seconds_bucket{le="2.5"} 1.0
http_request_duration_seconds_bucket{le="5.0"} 1.0
http_request_duration_seconds_bucket{le="7.5"} 1.0
http_request_duration_seconds_bucket{le="10.0"} 1.0
http_request_duration_seconds_bucket{le="+Inf"} 1.0
http_request_duration_seconds_count 1.0
http_request_duration_seconds_sum 1.9943E-5
# HELP httpsessions_max httpsessions_max
# TYPE httpsessions_max gauge
httpsessions_max -1.0
# HELP httpsessions_active httpsessions_active
# TYPE httpsessions_active gauge
httpsessions_active 0.0
# HELP mem mem
# TYPE mem gauge
mem 252486.0
# HELP mem_free mem_free
# TYPE mem_free gauge
mem_free 88750.0
# HELP processors processors
"processors"
```

Your Turn!

Now it's over to you.

If you don't feel very confident, start with the provided examples.

Once you've got to grips with the examples, try something new. A new language. A new container. New endpoints.

1. Pick a language
2. Write a webserver
3. Add the instrumentation
4. Wrap in a docker container

You may encounter a range of issues. E.g. build tools, firewalls, etc.

We're going to spend quite a bit of time doing this, to allow you to experiment. Feel free to go off-piste!

Hands On

Prometheus Queries

Up to now we haven't considered how to use the data we are collecting.

The most we've done is plotted a metric in the Prometheus Graph UI.

One important part of Prometheus is its query engine.

This allows you to perform simple mathematical operations of the time series data.



What is a Query?

- Uses a functional expression language
- Select and aggregate time series data in real time
- Can be viewed as a plot, in a table or exported via the API.

The UI

You've already seen where you would type the queries in the UI. This is often the best place to start developing a query.

The API

Prometheus has a simple HTTP api to access all it's internal data. [docs](#)

This is what external tools use to get access.

```
$ curl 'localhost:9090/api/v1/query?query=process_cpu_seconds_total'
```

Where the query parameters may be:

- `query=<string>` : Prometheus expression query string.
- `time=<rfc3339 | unix_timestamp>` : Evaluation timestamp. Optional.
- `timeout=<duration>` : Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout`

```
{  
  "status": "success",  
  "data": {  
    "resultType": "vector",  
    "result": [  
      {  
        "metric": {  
          "__name__": "process_cpu_seconds_total",  
          "instance": "localhost:9090",  
          "job": "prometheus"  
        },  
        "value": [  
          1510237891.436,  
          "0.42"  
        ]  
      }  
    ]  
  }  
}
```

Range of Data

To get a range of data we can use the `query_range` api.

```
$ curl 'localhost:9090/api/v1/query_range?query=process_cpu_seconds_total&start=2017-11-09T14:28:30.000Z&end=2017-11-09T14:29:30.000Z&step=10s'
```

Where the query parameters are:

- `query=<string>` : Prometheus expression query string.
- `start=<rfc3339 | unix_timestamp>` : Start timestamp.
- `end=<rfc3339 | unix_timestamp>` : End timestamp.
- `step=<duration>` : Query resolution step width.
- `timeout=<duration>` : Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout` flag.

```
{  
  "status": "success",  
  "data": {  
    "resultType": "matrix",  
    "result": [  
      {  
        "metric": {  
          "__name__": "process_cpu_seconds_total",  
          "instance": "localhost:9090",  
          "job": "prometheus"  
        },  
        "values": [  
          [  
            1510237710,  
            "0.28"  
          ],  
          [  
            1510237720,  
            "0.29"  
          ],  
          [  
            1510237730,  
            "0.29"  
          ],  
          [  
            1510237740,  
            "0.31"  
          ],  
          [  
            1510237750,  
            "0.32"  
          ]  
        ]  
      }  
    ]  
  }  
}
```

And More

Theres more to the API. You can also:

- Query by labels
- Targets
- Alert Managers

Query Types

There are three main types of return values in Prometheus:

- **Instant vector** – a set of time series containing a single sample for each time series, all sharing the same timestamp
- **Range vector** – a set of time series containing a range of data points over time for each time series
- **Scalar** – a simple numeric floating point value

Depending on your query, you will receive these different results. [docs](#)

Instant and Range Vectors

When you use the table format, you will receive an instantaneous measurement.

When you use the plot format, you will receive a range of measurements.

Under the hood the two are using the `query` and `query_range` APIs we saw earlier.

Filtering by Label

You may have already noticed this in previous examples.

To filter results by label, we provide key-value pairs in curly braces:

```
process_cpu_seconds_total{job="prometheus"}
```

This allows us to select dimensions of interest.

We can use `!` to negate the equality:

```
process_cpu_seconds_total{job!="prometheus"}
```

And we can also use `=~` as a regex and `!~` as a negative regex.

```
http_requests_total{environment=~"staging|testing|development",method!="GET"}
```

Query Ranges

To select a range we use square braces (e.g. like indexing) to select an amount of time back from the current instant. For example:

```
http_requests_total{job="prometheus"}[5m]
```

Will select the previous 5 minutes worth of data from the prometheus job.

Note that you won't be able to plot this in the Graph UI because that is performing a range query (i.e. it's trying to append [5m] for you).

Images on the next few pages...

The screenshot shows the Prometheus web interface. At the top, there is a dark header bar with the following navigation items: Prometheus (selected), Alerts, Graph, Status ▾, and Help. Below the header is a sub-header with the text "Rewind the end time." followed by a small link icon.

The main area contains a search bar with the query "process_cpu_seconds_total[job=\"prometheus\"][5m]". To the right of the search bar, there is some performance information: "Load time: 16ms", "Resolution: 14s", and "Total time series: 0".

Below the search bar is a blue "Execute" button and a dropdown menu with the placeholder "- insert metric at cursor -". A red error message box is displayed, containing the text "Error executing query: invalid expression type \"range vector\" for range query, must be Scalar or instant Vector".

At the bottom of the interface, there are two tabs: "Graph" (selected) and "Console".

Prometheus Alerts Graph Status ▾ Help

```
process_cpu_seconds_total{job="prometheus"}[5m]
```

Load time: 9ms
Resolution: 14s
Total time series: 1

Execute - insert metric at cursor - ▾

Graph **Console**

Element	Value
process_cpu_seconds_total{instance="localhost:9090",job="prometheus"}	0.88 @1510238379.059 0.89 @1510238394.058 0.9 @1510238409.062 0.92 @1510238424.06 0.93 @1510238439.062 0.94 @1510238454.061 0.96 @1510238469.058 0.97 @1510238484.061 0.99 @1510238499.059 1 @1510238514.062 1.01 @1510238529.061 1.03 @1510238544.059 1.04 @1510238559.062 1.05 @1510238574.06 1.06 @1510238589.061 1.08 @1510238604.059 1.09 @1510238619.061 1.11 @1510238634.062 1.12 @1510238649.058 1.2 @1510238664.057

Remove Graph

Add Graph

Time durations are specified as a number, followed immediately by one of the following units:

- s - seconds
- m - minutes
- h - hours
- d - days
- w - weeks
- y - years

Offsetting Time

It's not something that you need to do often (maybe for a comparison to yesterday?) but you can also offset the time:

```
http_requests_total{job="prometheus"}[1m] offset 5m
```

That is: "get a minute's worth of data from 5 minutes ago".

Query Operators

There's the usual mathematical operators that do what you'd expect. [docs](#)

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulo)
- ^ (power/exponentiation)
- == (equal)
- != (not-equal)
- > (greater-than)
- < (less-than)
- >= (greater-or-equal)
- <= (less-or-equal)

Query Aggregations

There's a range of functions (similar to most math packages) to do sums, averages, min/max, etc. [docs](#)

Here are some of the most common...

sum

Calculate the sum over all dimensions.

```
sum(http_requests_total)
```

You can specify the dimension to preserve when performing a sum. By default it sums everything together. But we can provide:

```
sum(http_requests_total[5m]) by (job)
```

To provide a sum over all dimensions except job .

This is very useful for separating services/instances/API paths/API request types/errors/etc.

avg

Calculate the average over all dimensions

```
avg(http_requests_total)
```

Same as: sum(http_requests_total) / count(http_requests_total)

Query Functions

Prometheus has a range of helper functions. [docs](#)

Here is a description of some of the most important.

rate

docs

This is used all the time.

Calculates the per-second average rate of increase in a range vector.

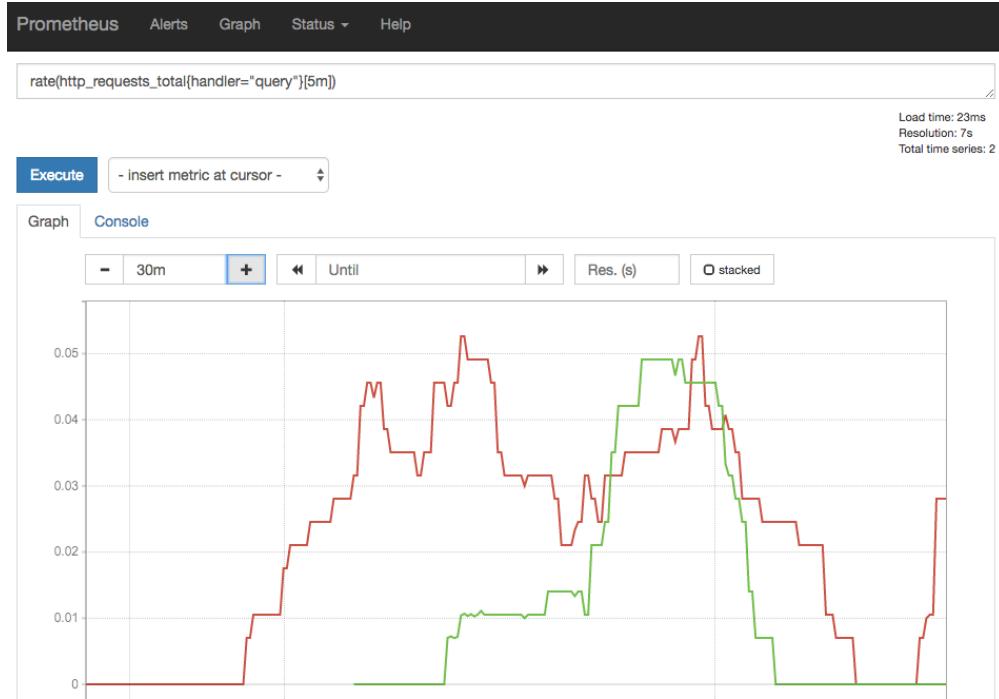
I.e. It performs the calculation `observation[now] - observation[-1s]`

- Should only be used with counts
- When combining with another operator, always use the `rate` first.

E.g. Calculate the per-second rate of HTTP requests over the last 5 minutes.

```
rate(http_requests_total{handler="query"}[5m])
```

Image on next page...



histogram_quantile

Calculate a percentile ($0 \leq q \leq 1$) from the buckets of a histogram metric.

E.g. The HTTP request duration of the 99.9th percentile (three nines):

```
histogram_quantile(0.999, rate(http_request_duration_seconds_bucket[10m]))
```

E.g. The median:

```
histogram_quantile(0.5, rate(http_request_duration_seconds_bucket[10m]))
```

E.g. Quantiles over all labels (e.g. over all APIs):

```
histogram_quantile(0.9, sum(rate(http_request_duration_seconds_bucket[10m])) by (le))
```

The `by (le)` is saying, sum for each `le` where `le` is the internal notation for a histogram bucket boundary.

And More

There are loads more. But it depends on what you're doing.

Check out the docs if you get stuck.

<https://prometheus.io/docs/prometheus/latest/querying/functions/>

Queries For Our Applications

Both our applications exposed the Histogram `http_request_duration_seconds`.

(Hopefully yours did too!)

We can use that to generate our own metrics.

Let's start with something simple

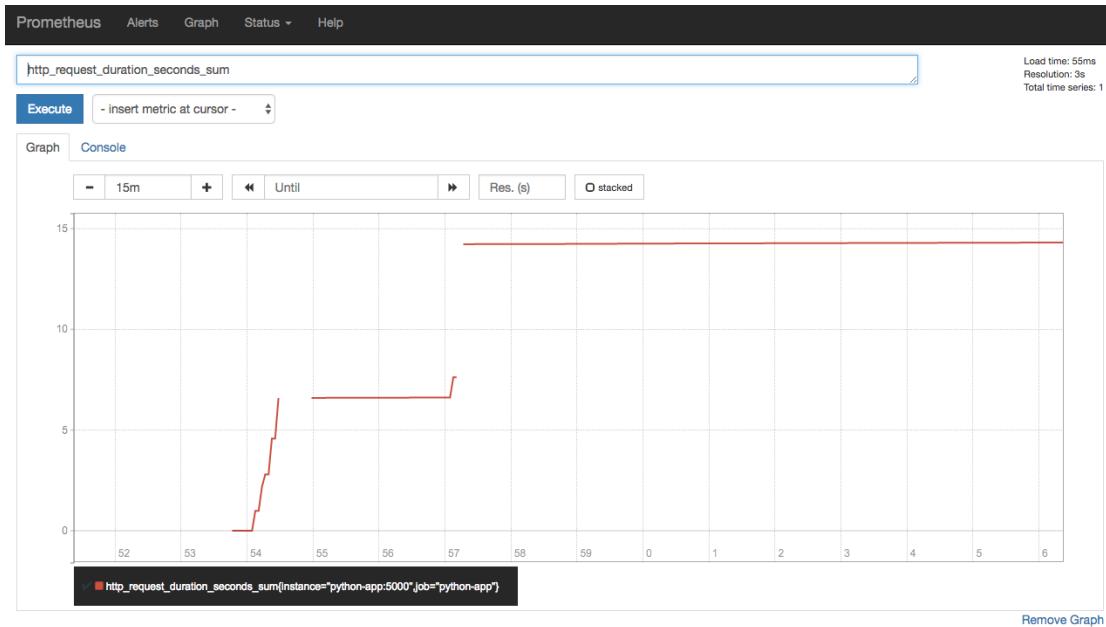
`http_request_duration_seconds_count`

This shows the instantaneous value for the total number requests.

Here we're using the histogram to provide us with a counter. A histogram will create the following metrics:

- `http_request_duration_seconds_count` – Count of requests
- `http_request_duration_seconds_sum` – Sum of all durations
- `http_request_duration_seconds_bucket` – Histogram buckets

Image on next page...



Convert That To a Rate

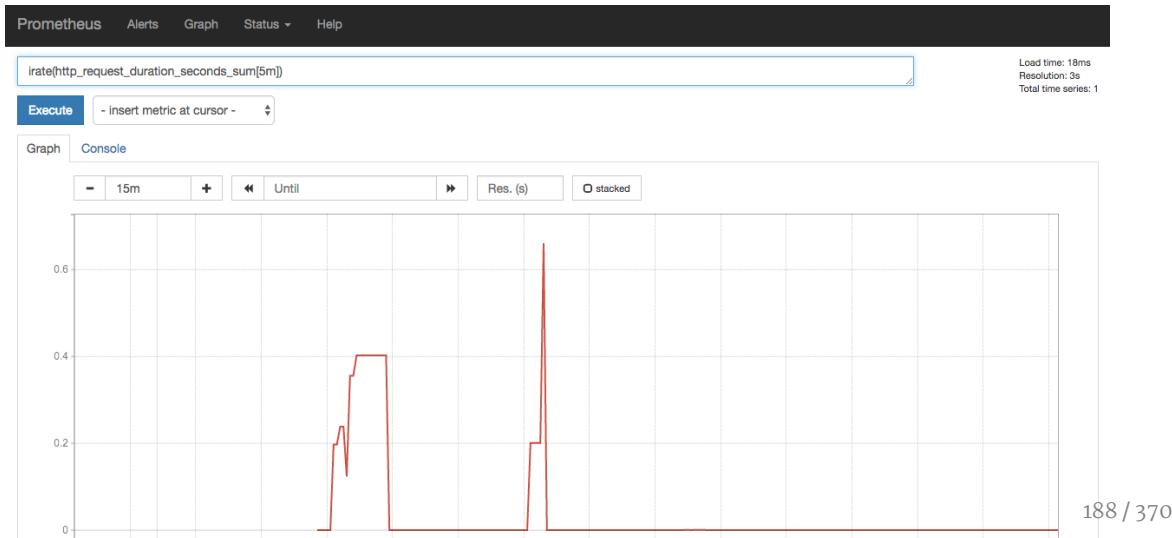
```
rate(http_request_duration_seconds_sum[5m])
```

This provides us with the rate of the number of requests over 5 minute sliding windows.
I.e. this is being smoothed.

Look at the instantaneous rate (`irate`)

```
irate(http_request_duration_seconds_sum[5m])
```

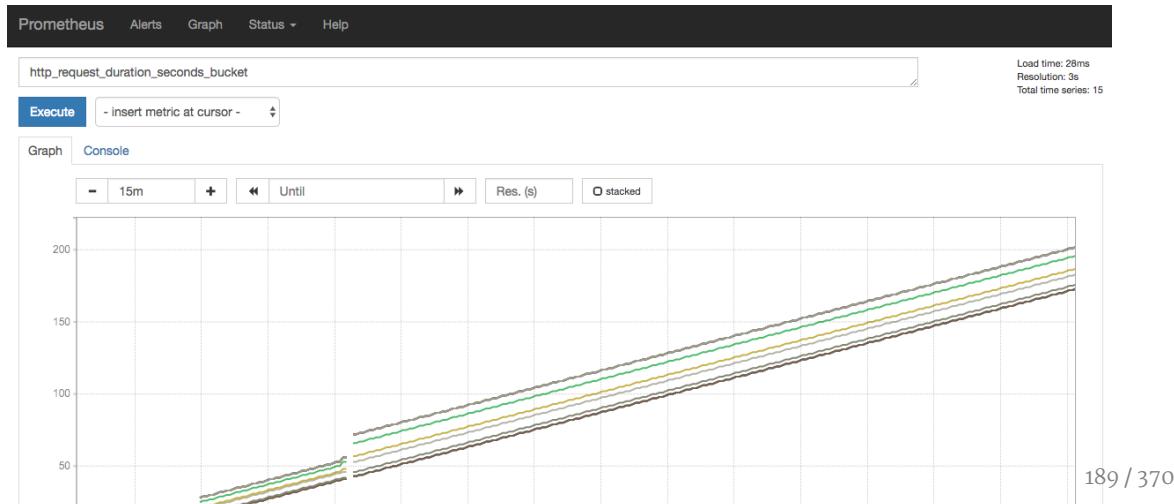
Cool, that's just about queries per second.



Histogram

`http_request_duration_seconds_bucket`

If we look at the raw histogram data, you'll see that it's a combination of buckets, each of which corresponds to a time "bin".



Median Request Duration

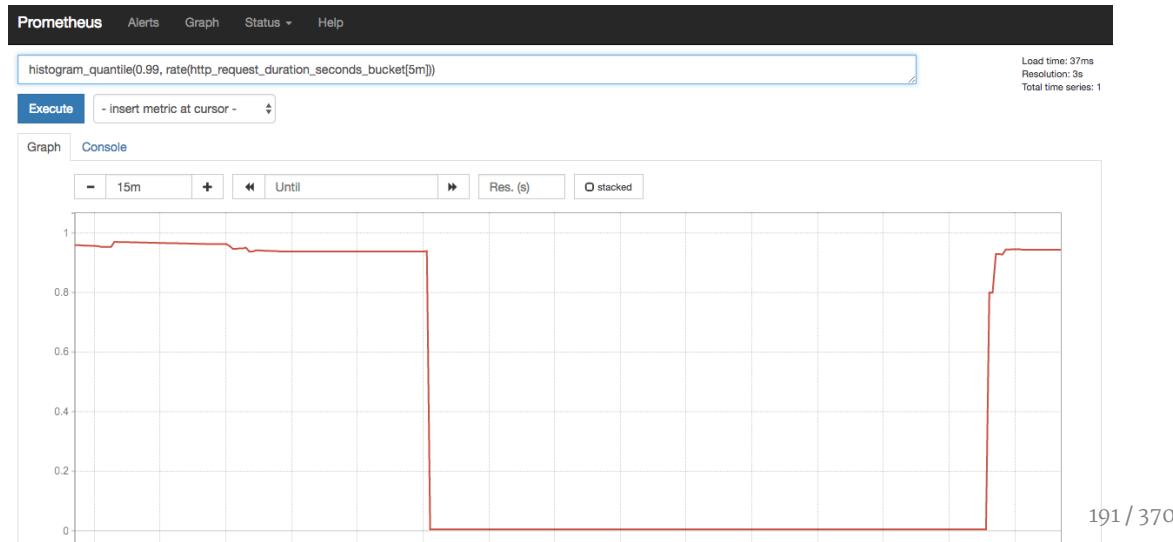
```
histogram_quantile(0.5, rate(http_request_duration_seconds_bucket[5m]))
```

Note that you'll probably need to use a slow (delayed) API to get it to work slower than the lowest default bucket.

99% Request Duration

```
histogram_quantile(0.99, rate(http_request_duration_seconds_bucket[5m]))
```

This one corresponds to your SLO/SLI.



191 / 370

Hands On

Let's implement some queries for your apps.

302: Data Science

What is data science? Data Science is a combination of:

- Statistics
- Software Engineering
- Analysis
- Research

Sub-disciplines include:

- Machine Learning
- Big Data
- Analytics
- So called A.I.
- etc.

[Find out more](#)

What's Data Science got to do with Monitoring?

Good question.

Think about what we're doing. We're creating metrics. These metrics are time series.

We are working with data. We need to understand a little about the Science so we can make better use of the data.

This section introduces Data Science, in relation to monitoring.

Statistics

Statistics is a mathematical discipline that helps us describe *non-stationary* data.

Non-stationary means that if we were to measure the same thing twice, we wouldn't get the same result each and every time.

Statistics is all based around the idea of *probability*. This is the chance of an event occurring.

E.g. We use probability in our language all of the time without noticing it.

- I'm never going to win the lottery => a low probability
- It's probably going to be sunny in Spain tomorrow => a high probability.
- What are the chances of that! England won the World Cup! => A very low probability.

Probability

One way of defining probability is on a scale of 0 to 1. 0 means it will never happen, 1 means it will always happen.

Nearly everything you can ever think of is greater than 0 probability and less than 1.

How can we use Probability?

Thinking probabilistically allows us to quantify how probable an event is.

"What are the chances that production is going to go down whilst I'm on call?"

"Given this strange data, what is the probability that this is going to affect production?"

But first, we need to understand what "random" looks like...

What is Random?

A random, or stochastic event is one that can only ever be represented by a range of values. There is no exact answer.

A large proportion of all data falls into this category. Even think that you think of as stationary.

For example, how much money do you have in your bank account?

- Really?
- Are you sure?
- What's that? The Bank uses atomic locks on their Microsoft SQL server so it's perfectly safe?
- Oh no. The value of your currency has just changed and now you can't buy as much as you could before.

Even the value of cold hard cash is a random variable. It is not stationary.

Probability Distributions

Random variables can be described in a number of ways. Most often they are described by a *summary statistic*.

That is a set of numbers that summarizes the tendency of the data.

But a random variable varies according to how probable each value is, not how the summary statistic states it should.

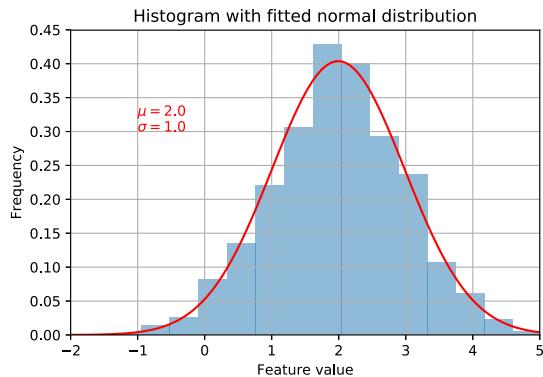
The probability of each value is called the *probability distribution*. A distribution of probabilities for each value.

Another way is to present the spread of the data visually, in something called a *histogram*.

(Have you seen that word before?!)

Histograms

Histograms are counts of the number of times an *observation* falls into a *bin*.



Different Types of Histograms

When viewing the data like this, you can see a whole range of distributions.

Yeah, So What?

Wait, we'll get there. :-)

A Quick Note on Terminology

- **Observation** – A single measurement
- **Population** – All possible measurements that could ever be made
- **Sample** – A subset of the population. Ideally a *representative* subset.

Modelling

Modelling is the task of summarising a non-stationary metric as a simpler set of summary statistics.

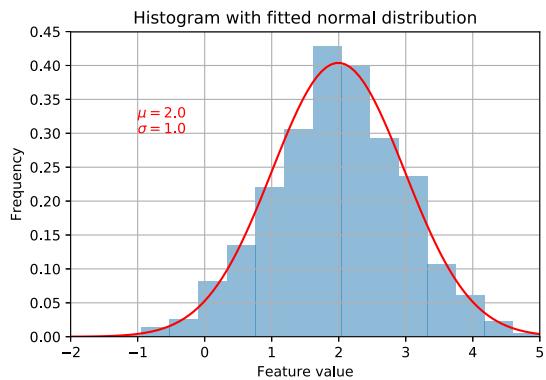
You've probably used one model a lot, the *mean*.

The mean is a measure of the central tendency of a Normal distribution.

The *Standard Deviation* is the amount of spread around that central tendency.

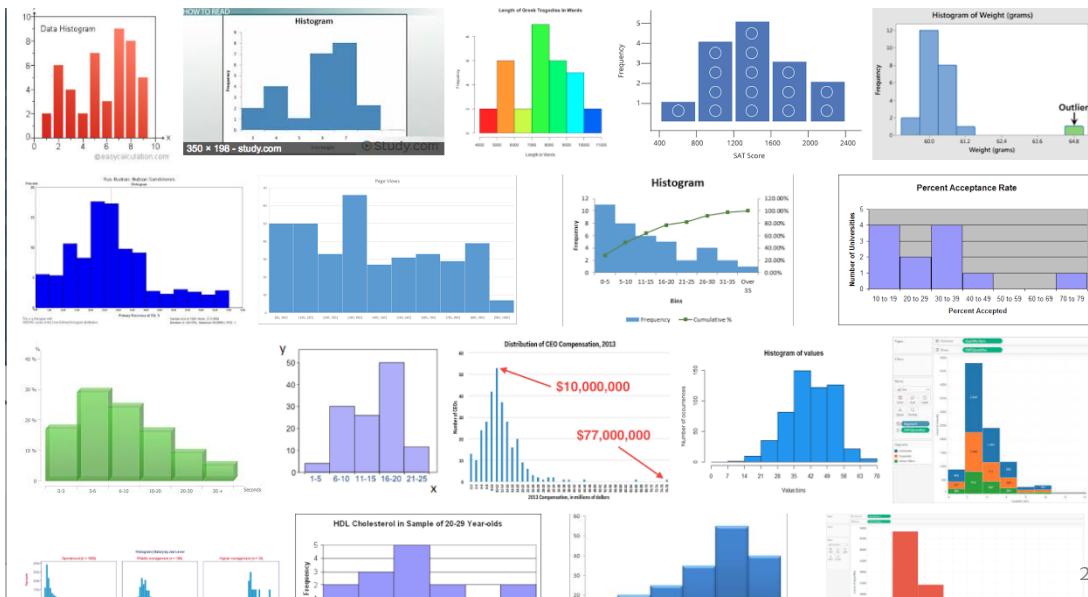
[Learn more about why we use standard deviation as a measure of spread here](#)

Let's look at that plot again:



Yeah, So What? (Part Duex)

The issue is that data never looks like that histogram.



What Does that Mean? Get to the Point!

Remember what we said earlier?

The *Mean* and *Standard Deviation* are summary statistics for a **Normal Distribution**.

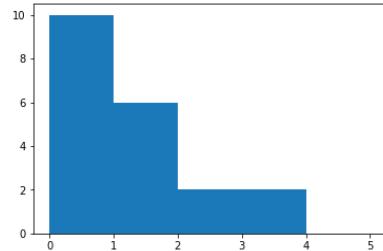
And **ONLY** a Normal Distribution.

Because our data is hardly ever normal (especially when working with monitoring data) the mean mis-represents the **most likely value** and the **most likely spread** of that value.

So, if you take one thing away from today, I want it to be this.

Don't use the mean!

Let's take a quick example. Imagine you had a set of HTTP request durations.



```
bad = [0,1,0,0,0,0,1,1,0,0,1,2,0,3,1,3,0,2,0,1]
print("\mu =", np.mean(bad), ", \sigma =", np.std(bad))
```

```
\mu = 0.8 , \sigma = 0.979795897113
```

The Median

So what do we do instead?

We use the **median**.

The median is the value that separates splits the probability distribution in half.

In other words, if we ordered all the values, the median is the one in the middle.

The key is that the median is not skewed as much by the mean and quite often (not always) represents the most common value.

Percentile

The median is just a special case of a percentile, the 50th percentile.

This is stating that 50% of the data is lower than this value.

We can also use the 99th percentile, to state that 99% of the values are lower than this value.

Full circle:

We use percentiles to represent HTTP durations because

- a) the mean isn't accurate and
- b) it's easy to create SLO's that correspond to "most of our users have this level of service".

Anomaly Detection

A large subject, but here's a quick primer.

Anomaly detection is detecting when things are not "normal". It comes down to two key steps:

1. Define a model
2. Alert when values deviate from that model

Creating an Anomaly Detector

To create an anomaly detector we need a model.

Thankfully you now know all about models! ;-)

Now you just need an alert that tells you that you are deviating from that model.

Most often this is something you can do manually.

- You know how systems should be operating.
- Sometimes you have SLIs that you have to meet.
- So simply set the "threshold" at that level.

Creating a More Advanced Anomaly Detector

The next level of sophistication is setting a baseline, which is the basis of your model.

Then you set your threshold at something higher than your baseline.

This works really well when you have metrics that change over time.

For example,

- maybe load is greater during the day.
- So you'd have to set the threshold higher than the daytime use.
- But that is way higher than it needs to be at night.

So we could have a trailing median, and set a threshold twice as high as that baseline.

This way, the threshold will change over time.

Anything more than this and we're coming dangerously close to doing a proper Data Science course!

More Information

- Free and commercial training: <https://TrainingDataScience.com>

Talk to Dr. Phil Winder, an expert in Data Science: <https://WinderResearch.com>.

Hands On

303: Prometheus Exporters

This is a quick section on Prometheus exporters.

What are Exporters?

Exporters are scrapers for tools and technologies that you can't instrument, even though you would like to.

E.g.

- Linux
- HAProxy
- Redis

They are required because we don't have access to that codebase! So we use a little *sidecar* to sit alongside the tool, probe for and expose metrics.

Obviously, each technology has a slightly different method of integration.

Thankfully, there are a number of tools that already export metrics...

Tools That Expose Prometheus Metrics Directly

- [cAdvisor](#)
- [Ceph](#)
- [Collectd](#)
- [CRG Roller Derby Scoreboard](#) ([direct](#))
- [Doorman](#) ([direct](#))
- [Etcd](#) ([direct](#))
- [FreeBSD Kernel](#)
- [Kubernetes](#) ([direct](#))
- [Linkerd](#)
- [Netdata](#)
- [Pretix](#)
- [Quobyte](#) ([direct](#))
- [RobustIRC](#)
- [SkyDNS](#) ([direct](#))

Notable Exporters

- [Consul exporter](#) (official)
- [ElasticSearch exporter](#)
- [Memcached exporter](#) (official)
- [MongoDB exporter](#)
- [MySQL server exporter](#) (official)
- [PostgreSQL exporter](#)
- [Redis exporter](#)
- [Node/system metrics exporter](#) (official)
- [Kafka exporter](#)
- [NATS exporter](#)
- [RabbitMQ exporter](#)
- [Hadoop HDFS FSImage exporter](#)

- [Apache exporter](#)
- [HAProxy exporter \(official \)](#)
- [Nginx metric library](#)
- [Nginx VTS exporter](#)
- [AWS ECS exporter](#)
- [AWS Health exporter](#)
- [AWS SOS exporter](#)

- [Fluentd exporter](#)
- [Google's mtail log data extractor](#)
- [Grok exporter](#)
- [AWS CloudWatch exporter \(official \)](#)
- [Google Stackdriver exporter](#)
- [Nagios / Naemon exporter](#)
- [New Relic exporter](#)
- [Jenkins exporter](#)
- [JIRA exporter](#)
- [Minecraft exporter module](#)

^^^ My favourite!

And more. Basically everything you can think of.

Using Exporters

How to use an exporter depends entirely on the thing being used.

For example, the *node exporter*, the exporter that monitors machine metrics, needs to run on each node.

However something like the redis-exporter can run anywhere and connect to the redis instance over the network.

Let's have a quick go with redis...



Hands on

304: Monitoring Kubernetes

Kubernetes, you know, the orchestrator, integrates with Prometheus really well.

In this section we'll be discussing how to do that and also get some hands on experience doing it.



kubernetes

Bonus #1: K8s Already Exposes Metrics!

K8s exposes the following subsystems:

- API Server
- Node information
- Resource usage (via cAdvisor)
- Service status (via blackbox exporter)
- Ingress status (via blackbox exporter)
- All scrapable services
- All scrapable pods!

So essentially, it just comes down to configuration.

Configuration

There are two elements to the configuration.

1. Prometheus configuration
2. Kubernetes configuration

As you probably know, the k8s configuration is quite verbose. So I'll be skipping a lot.

Prometheus Configuration

```
scrape_configs:  
- job_name: 'kubernetes-apiservers'  
  kubernetes_sd_configs:  
    - role: endpoints  
  scheme: https  
  tls_config:  
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt  
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token  
  relabel_configs:  
    - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name,  
      __meta_kubernetes_endpoint_port_name]  
      action: keep  
      regex: default;kubernetes;https
```

```
- job_name: 'kubernetes-nodes'
scheme: https
tls_config:
  ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [__meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/metrics
```

```
- job_name: 'kubernetes-cadvisor'
scheme: https
tls_config:
  ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [__meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
```

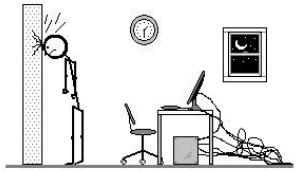
```
- job_name: 'kubernetes-service-endpoints'
kubernetes_sd_configs:
- role: endpoints
relabel_configs:
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
  action: keep
  regex: true
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
  action: replace
  target_label: __scheme__
  regex: (https?)
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
  action: replace
  target_label: __metrics_path__
  regex: (.+)
- source_labels: [__address__, __meta_kubernetes_service_annotation_prometheus_io_port]
  action: replace
  target_label: __address__
  regex: ([^:]+)(?::\d+)?;(\d+)
  replacement: $1:$2
- action: labelmap
  regex: __meta_kubernetes_service_label_(.+)
- source_labels: [__meta_kubernetes_namespace]
  action: replace
  target_label: kubernetes_namespace
- source_labels: [__meta_kubernetes_service_name]
  action: replace
  target_label: kubernetes_name
```

```
- job_name: 'kubernetes-services'
metrics_path: /probe
params:
  module: [http_2xx]
kubernetes_sd_configs:
- role: service
relabel_configs:
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_probe]
  action: keep
  regex: true
- source_labels: [__address__]
  target_label: __param_target
- target_label: __address__
  replacement: blackbox-exporter.example.com:9115
- source_labels: [__param_target]
  target_label: instance
- action: labelmap
  regex: __meta_kubernetes_service_label_(.+)
- source_labels: [__meta_kubernetes_namespace]
  target_label: kubernetes_namespace
- source_labels: [__meta_kubernetes_service_name]
  target_label: kubernetes_name
```

```
- job_name: 'kubernetes-ingresses'
metrics_path: /probe
params:
  module: [http_2xx]
kubernetes_sd_configs:
  - role: ingress
relabel_configs:
  - source_labels: [__meta_kubernetes_annotation_prometheus_io_probe]
    action: keep
    regex: true
  - source_labels:
      [__meta_kubernetes_ingress_scheme,__address__,__meta_kubernetes_ingress_path]
    regex: (.+);(.+);(.+)
    replacement: ${1}://${2}${3}
    target_label: __param_target
  - target_label: __address__
    replacement: blackbox-exporter.example.com:9115
  - source_labels: [__param_target]
    target_label: instance
  - action: labelmap
    regex: __meta_kubernetes_ingress_label_(.+)
  - source_labels: [__meta_kubernetes_namespace]
    target_label: kubernetes_namespace
  - source_labels: [__meta_kubernetes_ingress_name]
    target_label: kubernetes_name
```

And finally...

```
- job_name: 'kubernetes-pods'
  kubernetes_sd_configs:
    - role: pod
  relabel_configs:
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
      action: keep
      regex: true
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
      action: replace
      target_label: __metrics_path__
      regex: (.+)
    - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
      action: replace
      regex: ([^:]+)(?::(\d+)?);(\d+)
      replacement: $1:$2
      target_label: __address__
    - action: labelmap
      regex: __meta_kubernetes_pod_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      action: replace
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_pod_name]
      action: replace
      target_label: kubernetes_pod_name
```



What Was All That Labelling Stuff?

Let's slow down for a second.

Firstly, this is kept up to date by the authors of Prometheus. It's there in the examples folder.

Secondly, all that relabelling really caused a lot of confusion. What is it?

Relabeling allows us to do things like:

- Take input variables and then recursively scrape new targets
- Drop unnecessary metrics
- Drop unnecessary time-series
- Drop sensitive or unwanted labels
- Amend label formats
- [Link to repo config](#)
- [Example 1](#)
- [Example 2](#)

Kubernetes Manifests

How do we get this into k8s?

Let's go through a manifest.

Starting off slow...

```
---  
apiVersion: v1  
kind: Namespace  
metadata:  
  name: monitoring
```

```
---  
apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
  name: prometheus-core  
  namespace: monitoring  
  labels:  
    app: prometheus  
    component: core  
spec:  
  replicas: 1  
  template:  
    metadata:  
      name: prometheus-main  
      labels:  
        app: prometheus  
        component: core  
  ...
```

```
spec:
  serviceAccountName: prometheus-k8s
  containers:
  - name: prometheus
    image: prom/prometheus:v2.0.0
    args:
      - '--storage.tsdb.retention=12h'
      - '--config.file=/etc/prometheus/prometheus.yaml'
    ports:
    - name: webui
      containerPort: 9090
    volumeMounts:
    - name: config-volume
      mountPath: /etc/prometheus
  volumes:
  - name: config-volume
    configMap:
      name: prometheus-core
```

```
---
apiVersion: v1
data:
  prometheus.yaml: |
    global:
      scrape_interval: 10s
      scrape_timeout: 10s
      evaluation_interval: 10s
    # A scrape configuration for running Prometheus on a Kubernetes cluster.
    # This uses separate scrape configs for cluster components (i.e. API server, node)
...
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: prometheus-core
  namespace: monitoring
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
  labels:
    app: prometheus
    component: core
  annotations:
    prometheus.io/scrape: 'true' # <-- Look! Cool!
spec:
  type: NodePort
  ports:
    - port: 9090
      protocol: TCP
      name: webui
  selector:
    app: prometheus
    component: core
```



I'll stop there, as you're probably going code-blind.

But there are more required manifests:

- RBAC
- node-metrics exporter
- directory-size exporter
- state-metrics exporter
- And accompanying services and daemonSets.

Key Things to Remember

- RBAC
- This amount of configuration is usually done only once at the start, then for major version upgrades.
- Is mostly provided by the vendors. Don't sweat all that code, it's copy/paste.

What About Our Apps?

We may have mentioned it, you might have been code blind. I'm not sure.

One of the relabellers contained a special config which introduced a new set of annotations:

- `prometheus.io/scrape` : Only scrape services that have a value of `true`
- `prometheus.io/scheme` : If the metrics endpoint is secured then you will need to set this to `https` & most likely set the `tls_config` of the scrape config.
- `prometheus.io/path` : If the metrics path is not `/metrics` override this.
- `prometheus.io/port` : If the metrics are exposed on a different port to the service then set this appropriately.

What does that mean? All we need to do is...

Adding Scrape Annotations

```
spec:  
  template:  
    metadata:  
      annotations:  
        prometheus.io/scrape: 'true'
```

That's it! :-D

Because of all that crazy relabelling, the actual day-to-day use of Prometheus/K8s is remarkably simple.

(We can alter the default path and port if necessary)

- Obviously, we can set this to false to disable scraping for that service.

Summary

1. Copy paste a load of boilerplate
2. Add `scrape: true` to your manifest
3. Do the Monitoring Success Dance.



Hands On

Let's do this ourselves. We'll take a little time here as there are a few hoops to jump through.

401: Visualisation 1 - Grafana

A little recap.

- Lots of data.
- Lots of plots
- How on earth are you expected to be able to act upon metric names?

We don't. The first step is a dashboard.

A dashboard is a tool to provide you will all the information you need to know to

a) do your job (i.e. achieve the SLO's by monitoring the SLI's) b) fix problems.

The Perfect Dashboard

There are a few about, and you might even have a proprietry one that you use within your business.

And that's fine, you can use the Prometheus HTTP API.

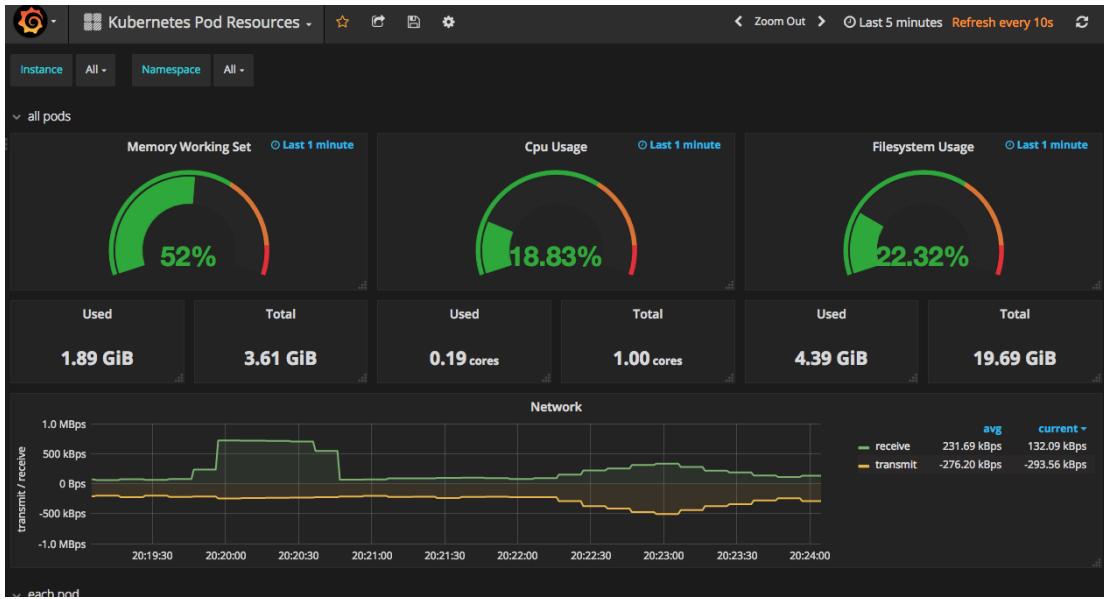
But the default choice for everything else is Grafana.

We're not going to all the details of Grafana, because there's quite a lot. And there's lots of pointing and clicking.

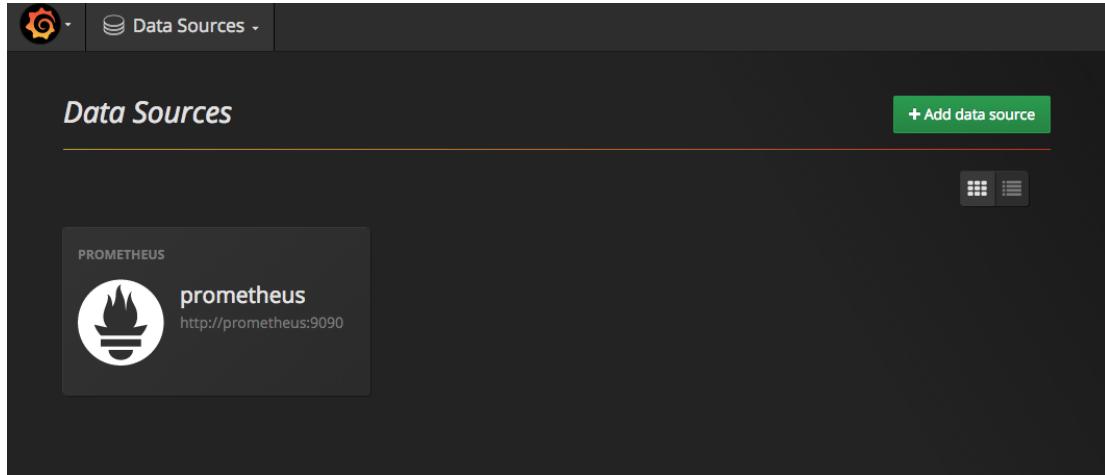
If you want to use it for real, then spend some time reading the documentation and looking at lots of examples. It's very powerful.



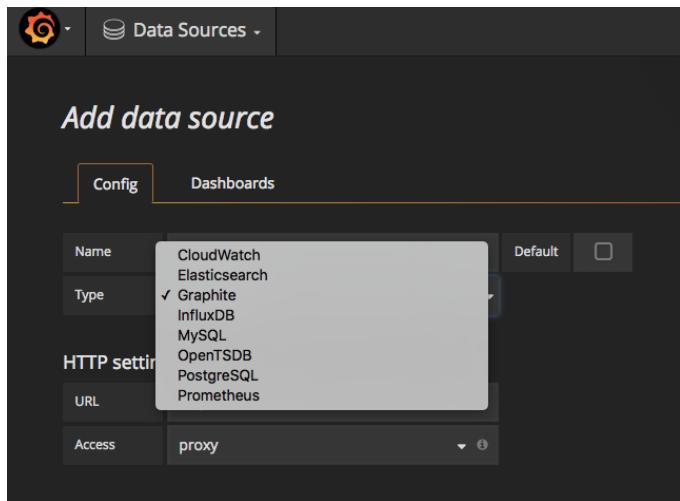
This is the main Dashboard. – Top left: menu. Top right: time controls.



Menu -> Data Sources



Add New Datasource



Menu -> Admin -> Global Users

The screenshot shows the Grafana interface with the 'Admin' menu selected. The 'Global Users' section is active, displaying a table with one row for the user 'admin'. The table columns are 'Login', 'Email', and 'Seen'. The 'Seen' column shows '2m'. To the right of the table are 'Edit' and 'Delete' buttons. A green 'Add new user' button is located in the top right corner of the main content area.

Login	Email	Seen	
admin	admin@localhost	2m	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Users and Orgs

You can secure dashboards with access control.

A User has access to certain Organisations through Roles .

An Organisation has it's own users, datasources and dashboards. You would use organizations when there are multiple tenants requiring access. Most users only have a single Organisation.

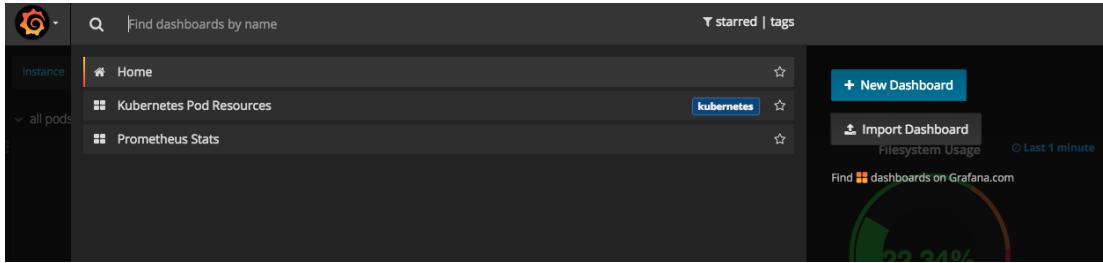
Roles can be:

- **Viewer** : Can only view dashboards, not save / create them.
- **Editor** : Can view, update and create dashboards.
- **Admin** : Everything an Editor can plus edit and add data sources and organization users.

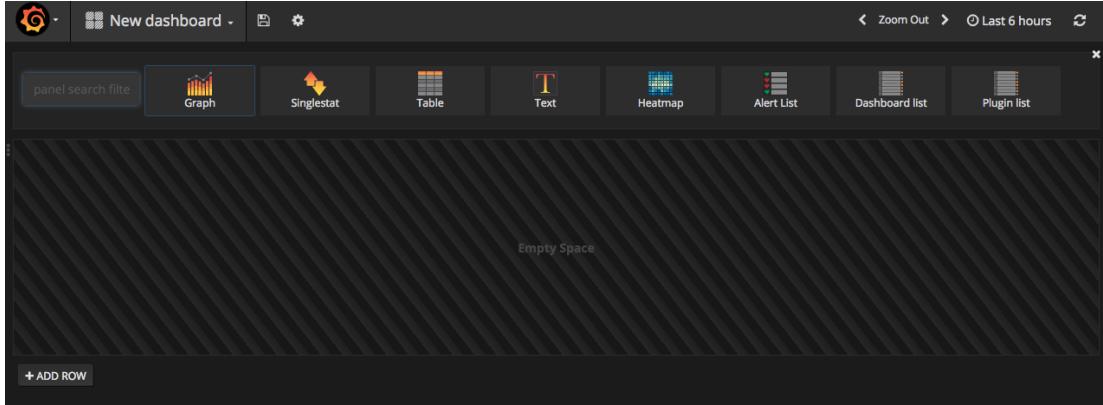
Dashboards

Dashboards are at the heart of Grafana. They display metrics through a range of Rows and Panels.

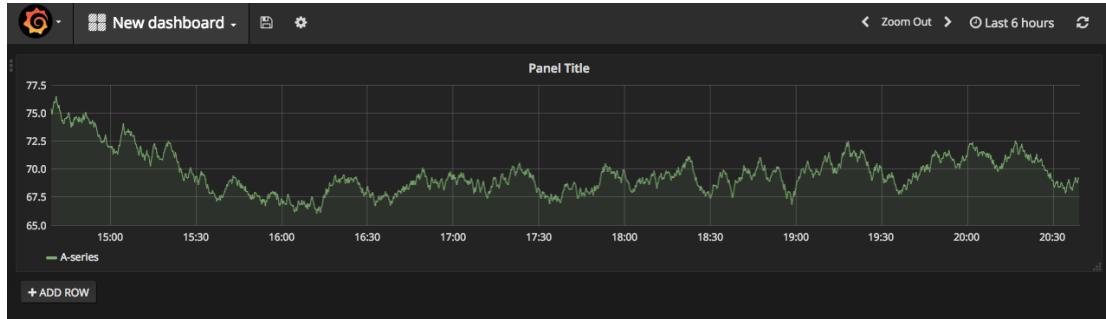
Create a new dashboard by clicking on the dropdown to the right of the Menu and then click New Dashboard.



This is an empty Row. You can add more rows by clicking Add Row.



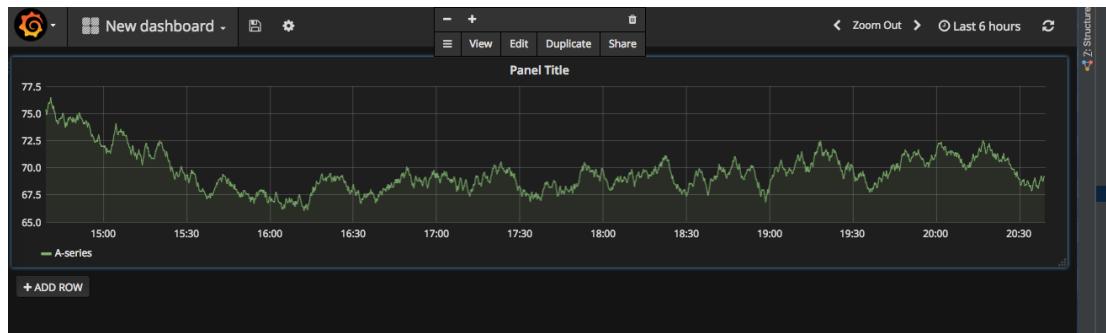
When you select a Panel it will be placed inside the Row.



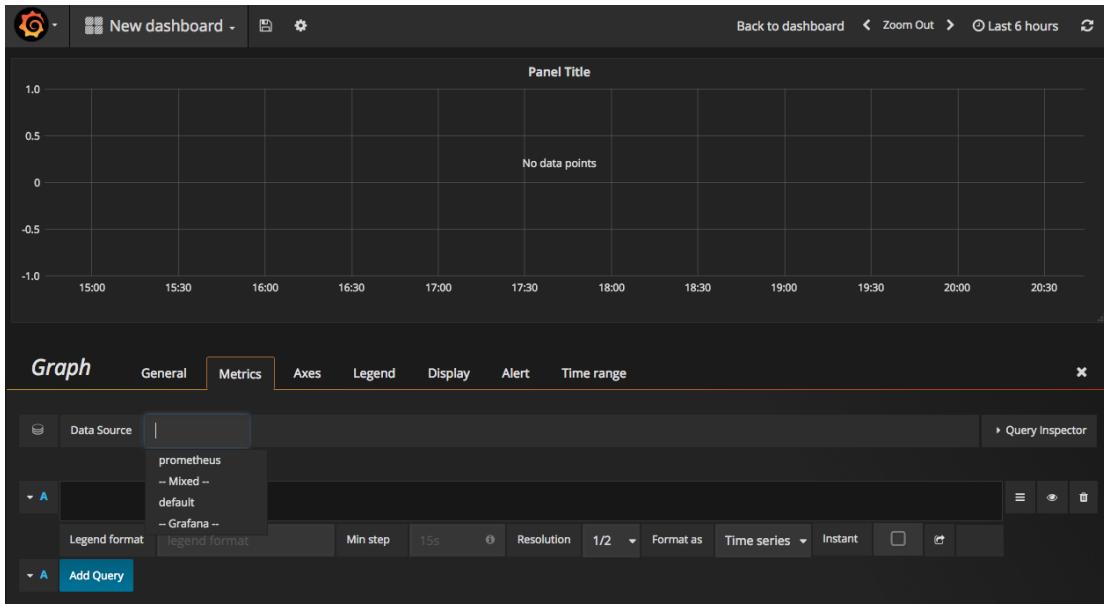
Editing A Panel

If you left click on a panel, a floating window will pop up.

Click on **Edit**.

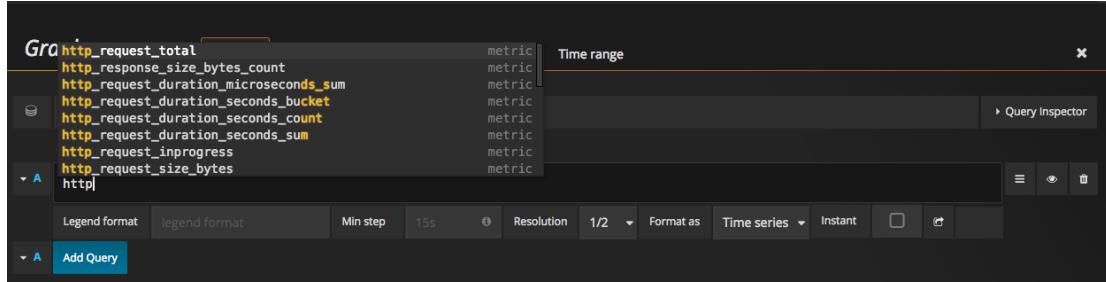


Select a Datasource.

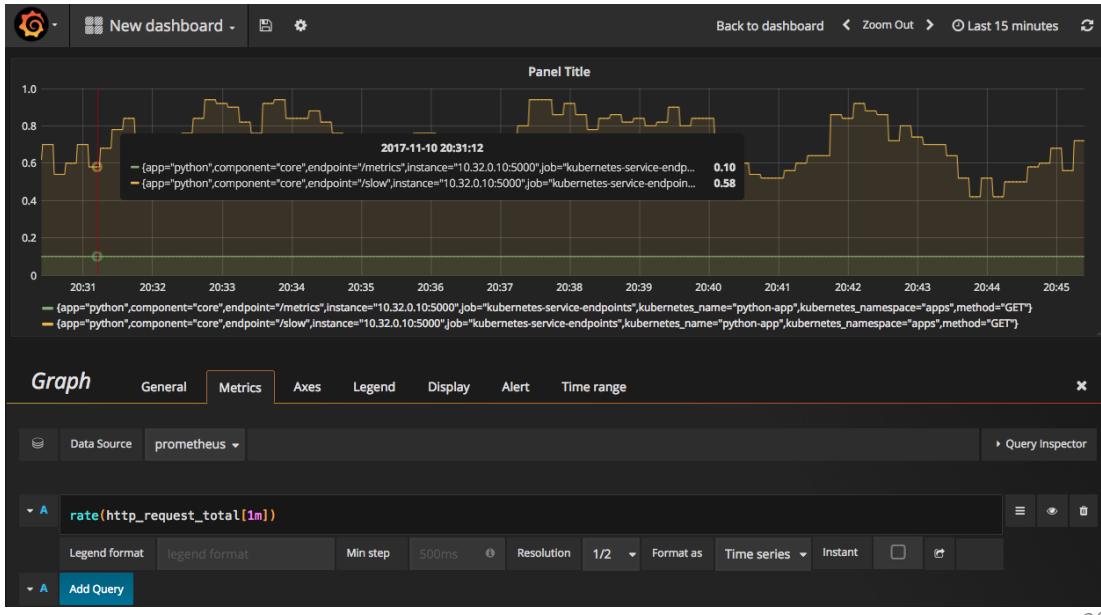


259 / 370

Type your query.



Monitor!

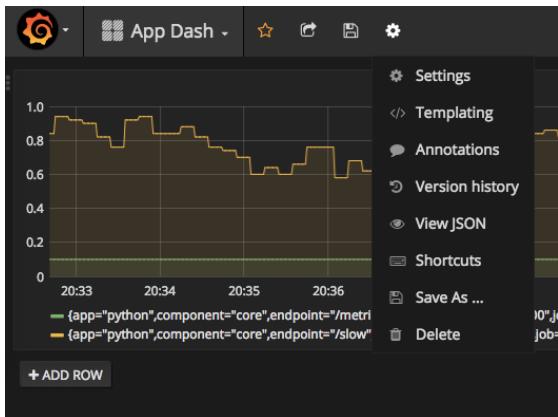


261 / 370

Storing and Saving Dashboards

Clicking the save button will save your dashboard in the internal, in-memory `sqlite3` database. You can alter the configuration to use an external database.

Also, all dashboards and datasources can be saved as JSON objects. You can see the JSON for the dashboard by clicking the `View JSON` button.



So How do we Install it?

A k8s manifest!

```
---  
apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
  name: grafana-core  
  namespace: monitoring  
  labels:  
    app: grafana  
    component: core  
...
```

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: grafana
        component: core
    spec:
      containers:
        - image: grafana/grafana:4.6.1
          name: grafana-core
          imagePullPolicy: IfNotPresent
      env:
        - name: GF_AUTH_BASIC_ENABLED
          value: "true"
        - name: GF_AUTH_ANONYMOUS_ENABLED
          value: "false"
      readinessProbe:
        httpGet:
          path: /login
          port: 3000
      volumeMounts:
        - name: grafana-persistent-storage
          mountPath: /var
      volumes:
        - name: grafana-persistent-storage
          emptyDir: {}
```

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: grafana  
  namespace: monitoring  
  labels:  
    app: grafana  
    component: core  
spec:  
  type: NodePort  
  ports:  
    - port: 3000  
  selector:  
    app: grafana  
    component: core
```

And Load the Dashboards?

A ConfigMap baby!

```
---  
apiVersion: v1  
data:  
  grafana-net-2-dashboard.json: |  
    {  
      "__inputs": [{"  
        "name": "DS_PROMETHEUS",  
        "label": "Prometheus",  
        "description": "",  
        "type": "datasource",  
        "pluginId": "prometheus",  
        "pluginName": "Prometheus"  
      }]  
    }  
  ...
```

There's 2000 lines of this!

```
...
}
prometheus-datasource.json: |
{
    "name": "prometheus",
    "type": "prometheus",
    "url": "http://prometheus:9090",
    "access": "proxy",
}
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: grafana-import-dashboards
  namespace: monitoring
```

And finally, a little job to load the config maps into Grafana using the Grafana HTTP API:

```
- name: grafana-import-dashboards
  image: giantswarm/tiny-tools
  command: ["/bin/sh", "-c"]
  workingDir: /opt/grafana-import-dashboards
  args:
    - >
      for file in *-datasource.json ; do
        if [ -e "$file" ] ; then
          echo "importing $file" &&
          curl --silent --fail --show-error \
            --request POST http://admin:admin@grafana:3000/api/datasources \
            --header "Content-Type: application/json" \
            --data-binary "@$file" ;
          echo "" ;
        fi
      done ;
      for file in *-dashboard.json ; do
        if [ -e "$file" ] ; then
          ...
        fi
      done ;
    fi
```

Hands On!

Enough of the endless manifests!

Visualisation 2: Templating with Prometheus

Remember that I said that Grafana can sometimes be too complicated?

Prometheus's key selling point is it's simplicity and Grafana can feel a bit... corporate.

We can do something else, templating.

What is Templating?

- Server-side rendering of code, usually HTML.

How does this help?

- Because we can load the template into Prometheus and use an endpoint off the Prometheus API to access our rendered HTML.

Why is that simpler?

- Because the code is static! It's statically generated content. There's no users, no organisations, no complex dashboards. Only what you need.

Introduction to Go Templates

- Under the hood, Prometheus is just using [Go Templates](#).

This is a mustache like syntax that is very powerful. It essentially allows you to access all of Go's language possibilities in scripts. E.g. I use it for my websites in the static site generator Hugo.

This is an example of a simple template:

```
simple.html: |  
  {{template "head" .}}  
  <h1>My Python Job!</h1>  
  {{template "tail"}}
```

All internal functions are called using a "mustache" syntax. This one calls an external template (e.g. another html file).

Prometheus Templating Libraries

Prometheus has a range of helper functions:

https://github.com/prometheus/prometheus/blob/master/docs/configuration/template_references.md

We can use these in our templates:

```
a.html: |
{{template "head" .}}
<h1>My Python Job!</h1>
<tr>
  <th>Python-App (# Pods)</th>
  <th>
    {{ template "prom_query_drilldown" (args "sum(up{kubernetes_name='python-app'})") }}
  / {{ template "prom_query_drilldown" (args "count(up{kubernetes_name='python-app'})") }}
  </th>
</tr>
{{template "tail"}}
```

This calls a Prometheus helper function called `prom_query_drilldown`. It allows us to perform a query and format the result.

Using Go Functions

Here's another example with all the HTML trimmings. We're using Go's `range` function ([godocs](#)) to loop through an array. Prometheus's `query` function returns raw results as a Go array.

```
b.html: |
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Title</title>
  </head>
  <body>
    <h1>Jobs that are up:</h1>
    <ul>
      {{ range query "up" }}
        <li>
          {{ .Labels.kubernetes_name }} {{ .Labels.instance }} {{ .Value }}
        </li>
      {{ end }}
    </ul>
    <h1>My Python Job!</h1>
    <tr>
      <th>Python-App (# Pods)</th>
      <th>
        {{ template "prom_query_drilldown" (args "sum(up{kubernetes_name='python-app'})") }} / {{ template "prom_query_drilldown" (args "count(up{kubernetes_name='python-app'})") }} ..
      </th>
    
```

A Plot

Finally, Prometheus provides a little javascript script based upon D3.js to plot the data.

```
c.html: |
  {{template "head" .}}
  <h1>A plot</h1>
  <tr>
    <th>Python-App (# Pods)</th>
    <th>
      {{ template "prom_query_drilldown" (args "sum(up{kubernetes_name='python-app'})") }}
    / {{ template "prom_query_drilldown" (args "count(up{kubernetes_name='python-app'})") }}
    </th>
  </tr>
  <div id="queryGraph"></div>
  <script>
new PromConsole.Graph({
  node: document.querySelector("#queryGraph"),
  expr: "sum(rate(http_request_duration_seconds_count[1m]))"
})
</script>
{{template "tail"}}
```

The head template contains the required javascript libraries. The `PromConsole` class has some more options: [see here](#).

Misc. Templating Notes

- This is rendered on the server-side. You'll have to refresh the browser to load metrics.
- When you reload the template you'll have to restart Prometheus. (So you'll lose the data)
- Make sure your queries work in the Prometheus UI first. It's easier to debug.
- You can use `{{ printf "%#v" . }}` to print the current template context.
- The possibilities are endless!

References

- [Go Templates](#)
- [Prometheus Function Reference](#)
- [More examples](#)

Hands On

Let's try your hand at HTML.

Bonus points for the best templates!

403: How to Build a Dashboard

In the previous chapters, we've learnt how to use Grafana and Templating to build dashboards.

We could go away now and build any dashboard we like.

But what makes a good dashboard? Let's find out.

What Makes a Good Dashboard?

- Simple
- Expresses meaning
- Reveals details as required
- Relevance

Information Overload

We have limited cognitive processing capacity. [1]

We can only process one (maybe a few) thing at a time. [2]

Context switching expends mental energy. [3]

- [1] Speier, Cheri; Valacich, Joseph; Vessey, Iris (1999). "The Influence of Task Interruption on Individual Decision Making: An Information Overload Perspective". *Decision Sciences*. 30 . [doi:10.1111/j.1540-5915.1999.tb01613.x](https://doi.org/10.1111/j.1540-5915.1999.tb01613.x)
- [2] https://en.wikipedia.org/wiki/Human_multitasking
- [3] Levitin, Daniel J. 2014. *The organized mind: thinking straight in the age of information overload*.

The Antidote: Minimalism

- Your dashboard should provide relevant information in less than **5 seconds** (5-second rule) [1]
- Severely restrict what is included. (E.g. a car dashboard) [2]
- No more than 5–9 Visualisations [1]
- Do not add a graph or text simply because you can [3]
- [1] [Dashboard Design Best Practices – 4 Key Principles](#)
- [2] [Theory and Practice of Dashboards](#)
- [3] [Designing and Building Great Dashboards – 6 Golden Rules to Successful Dashboard Design](#)

Meaning

- Place the most important information at the top left

We always start reading at the top left. The most important information should be at the top.

(Think like a journalist. They put the most important content at the top, details at the bottom.)

- Use styling to convey meaning

Red means bad. Green means good.

Big and Bold is important.

Grey and small is less important.

Keep colours and styles simple. Too much style can complicate the data.

Details

Always, **minimalism**.

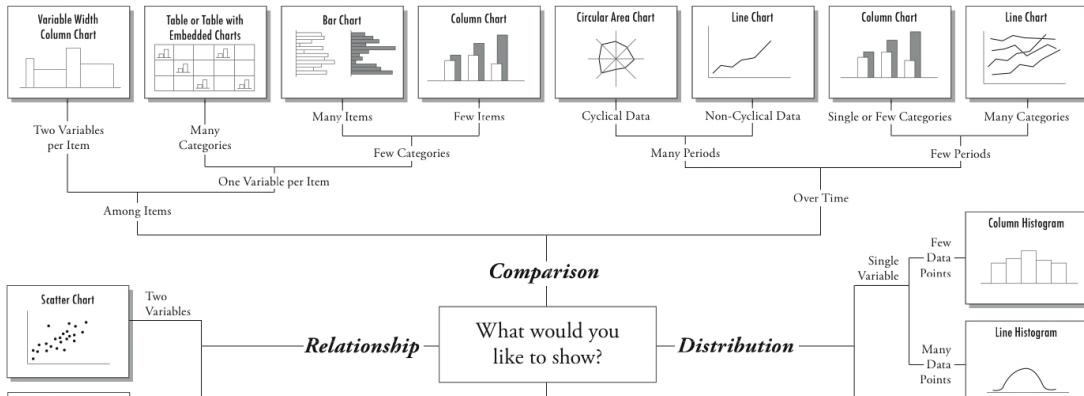
- Use plots and graphics to reduce the number of dimensions in the data.
- Always focus on the **SLI's/KPI's**.
- Select the right plot for the data...

How to Choose the Best Visualisation

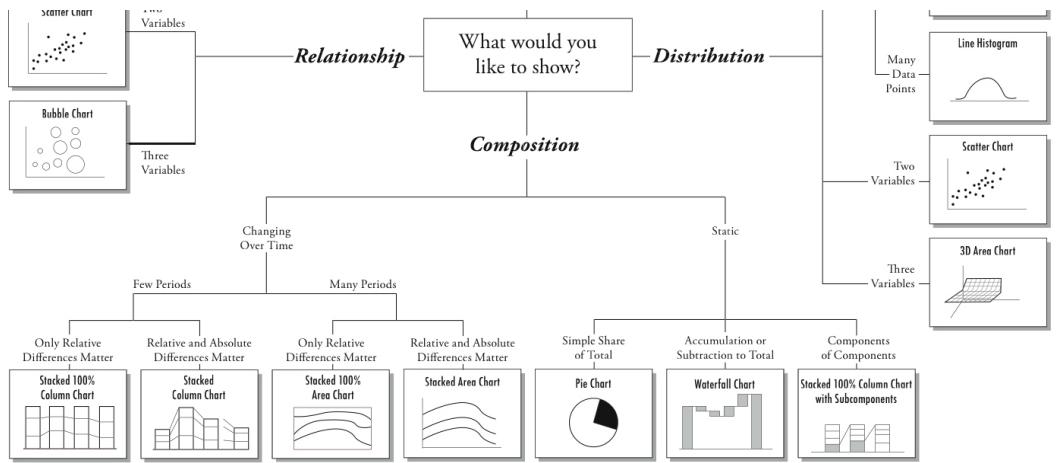
- Comparison
 - Among Items
 - Column
 - Table with embedded plots
 - Over Time
 - Line/Column
- Distribution
 - 1D: Histogram
 - 2D: Scatter

- Composition
 - Over Time
 - Stacked column/area plot
 - Static
 - Pie chart
 - Waterfall
- Relationship
 - Scatter plot

Chart Suggestions—A Thought-Starter



<https://extremepresentation.com/tools/>



www.ExtremePresentation.com
© 2009 A. Abela — a.v.abela@gmail.com

<https://extremepresentation.com/tools/>

Relevance: Who Is This Dashboard For?

Your software has a use case. Your dashboards need one too!

- Design dashboards to a **use case**

Think about the audience. Do they understand the metrics you are presenting? Does it need to be simplified?

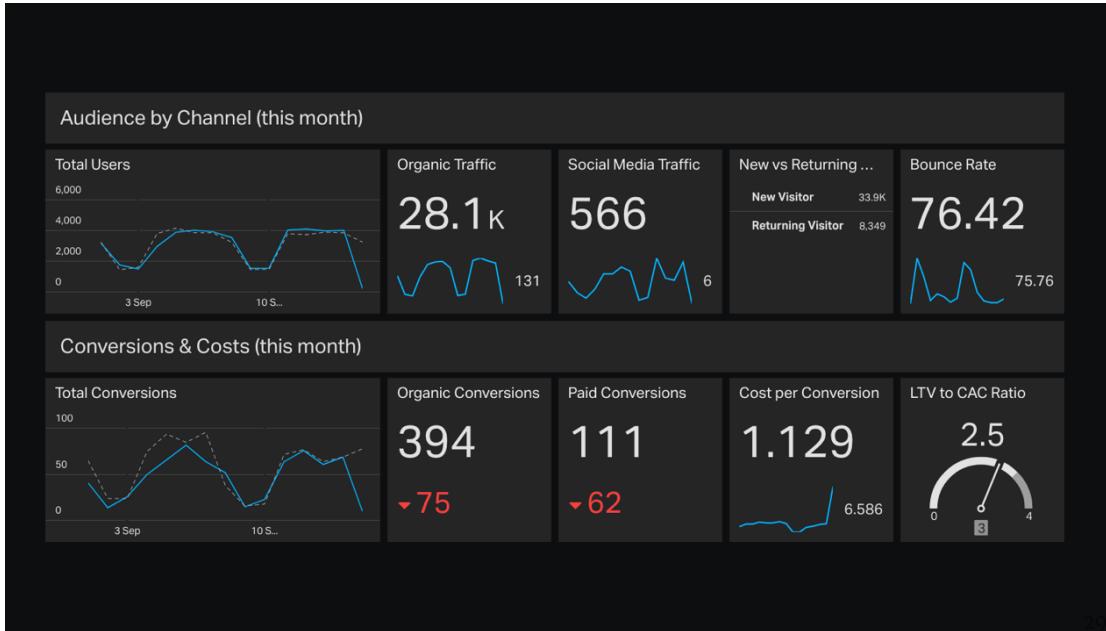
You should not have to explain your dashboard.

- Dashboards are **specific** to an audience. Make sure you know who the audience is.

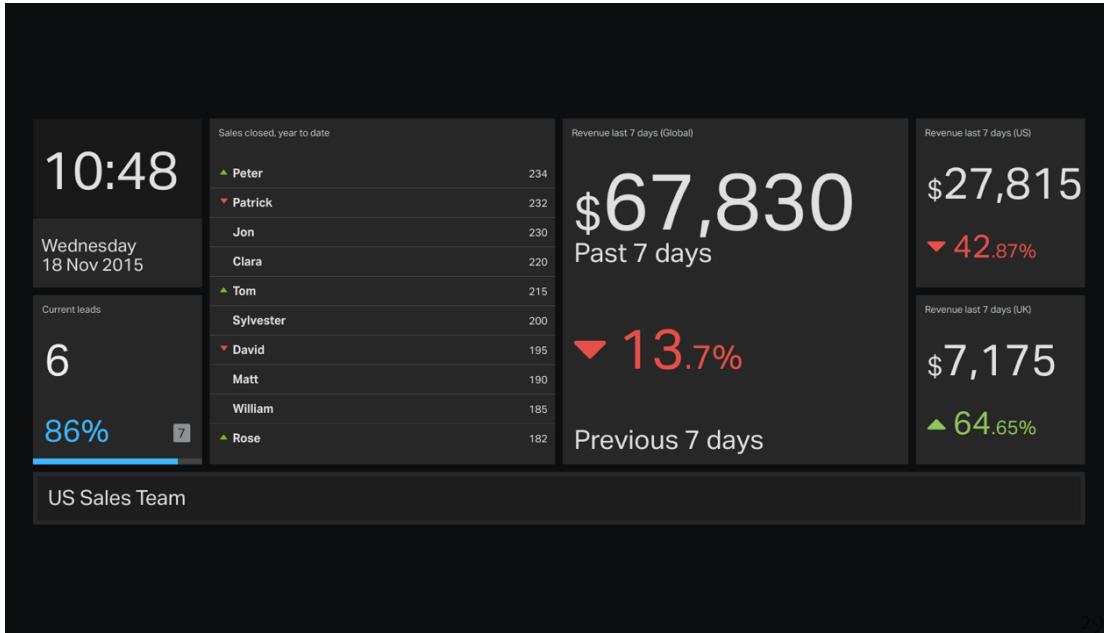
Lets look at some examples...

Examples from geckoboard

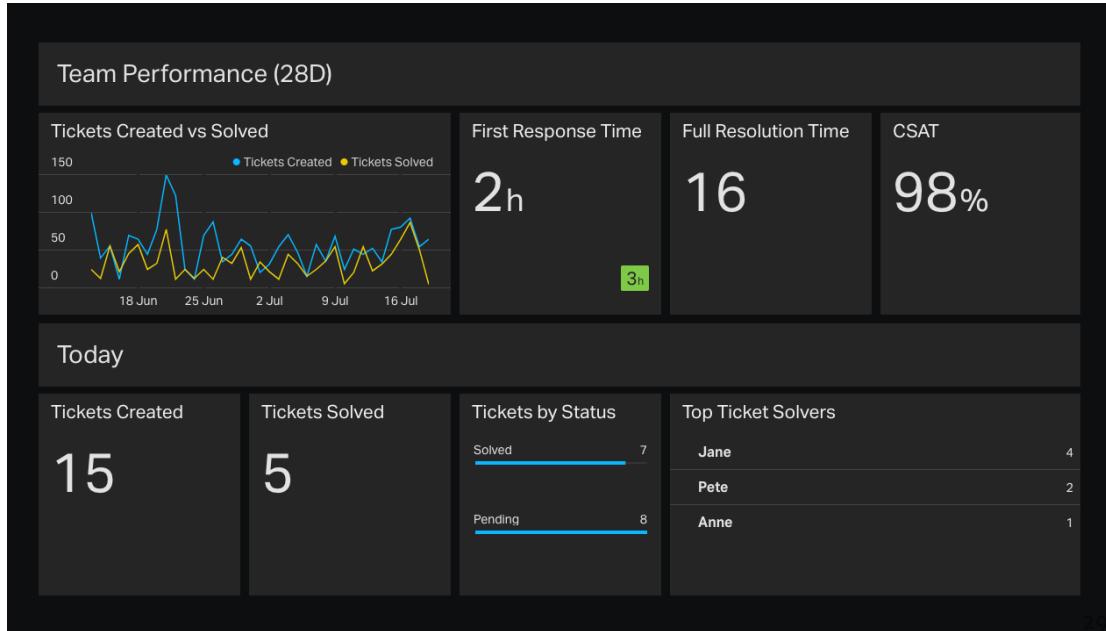
Marketing



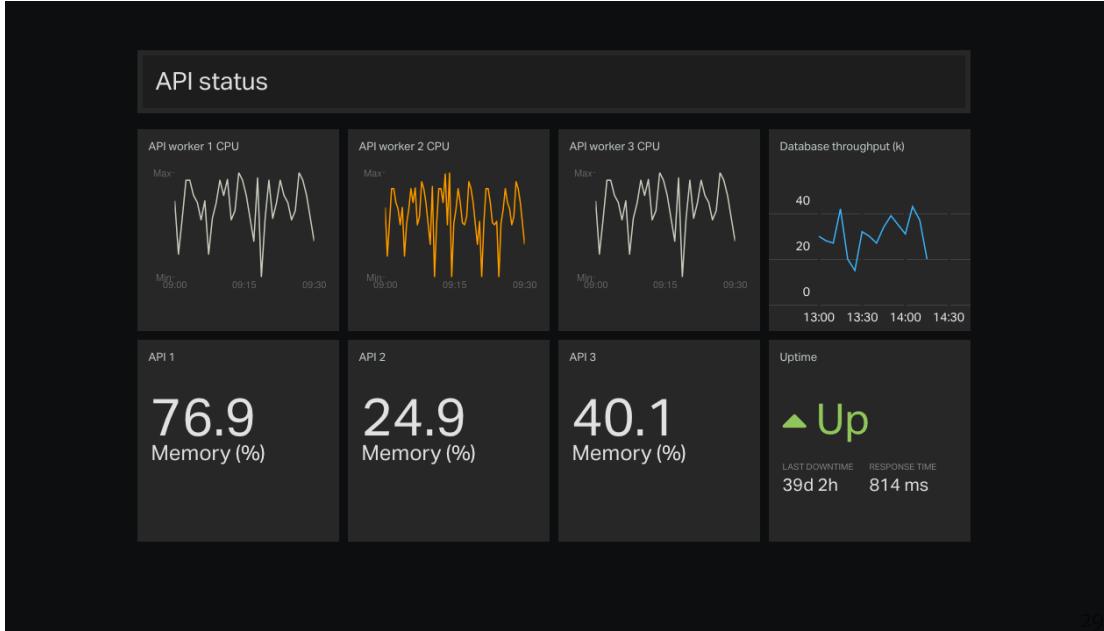
Sales



Support



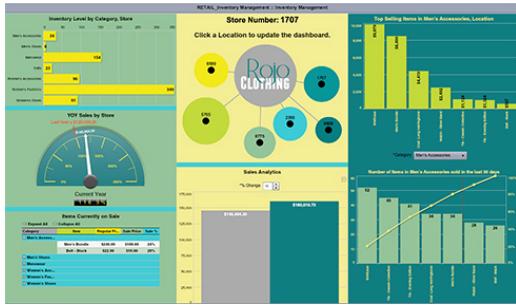
DevOps



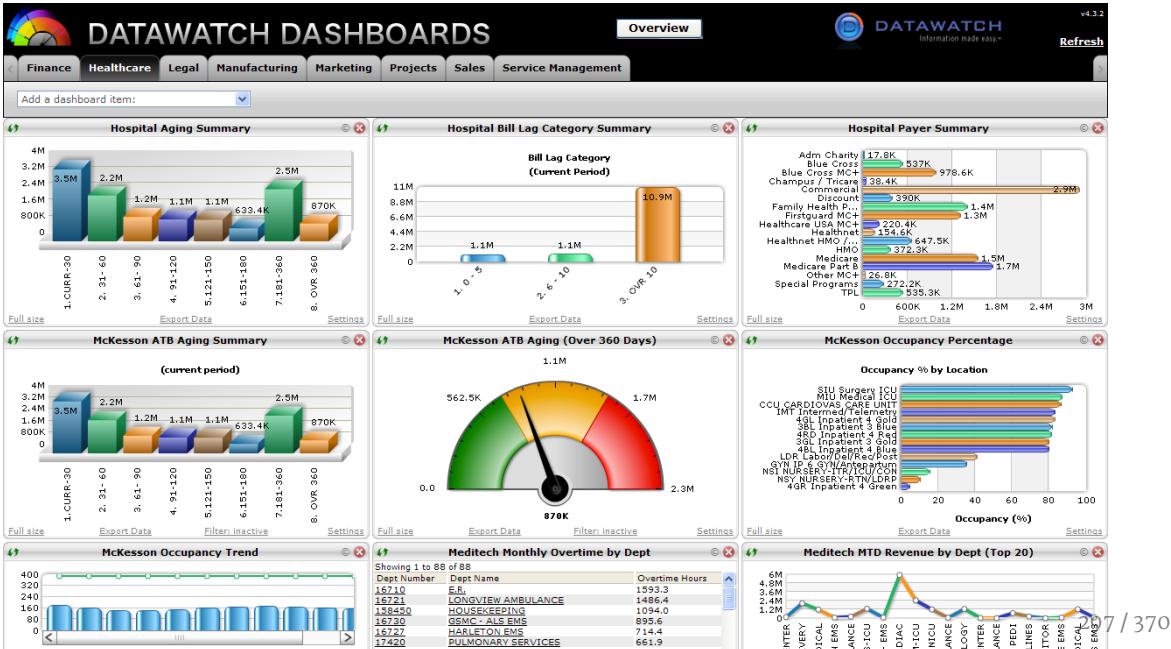
Bad Dashboards

Let's look at some bad dashboards...

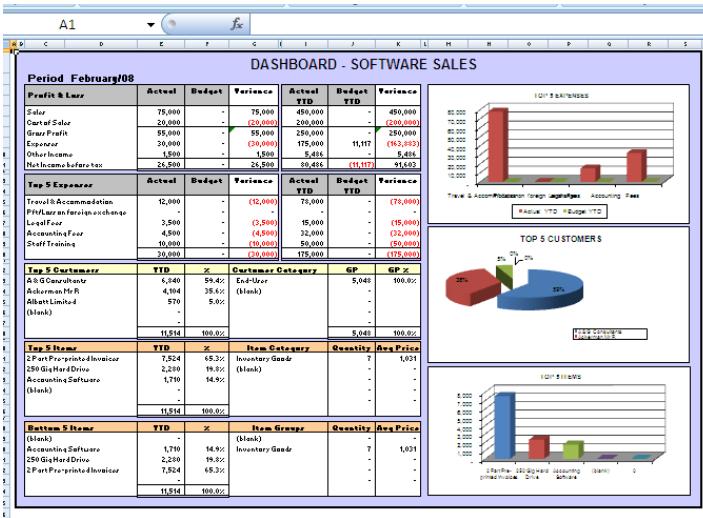
Colours!



TMI



Excel, really? Key info at right bottom.



Summary

- Keep it Simple
- Expresses meaning: Important at the top. Colours.
- Details: Remember the SLI's. Pick the right plot.
- Focus each dashboard on a use case and audience.

501: Alerting

We're now had lots of experience of analysing data, creating queries and visualising results on dashboards.

However, one common use case is to broadcast an alert when certain queries pass a threshold.

In Prometheus, this is achieved by

- alerting rules in Prometheus Core and
- a tool called Alertmanager



Architecture

Prometheus

Prometheus is responsible for:

- Interpreting and implementing alerting rules
- Firing alerts

Alertmanager

The Alertmanager is a small project that has three main responsibilities:

- Storing, aggregating and de-duplicating alerts
- Inhibiting and silencing alerts
- Pushing alerts out to external sources

General Setup

- Create the alerting rules in Prometheus
- Configure the Alertmanager
- Configure Prometheus to talk to Alertmanager

Alerting Rules

Note: There are two ways of defining rules. The pre 2.0 way, which was a text-based format or post 2.0 way which is yaml. I pick yaml for consistency.

```
- alert: <Name>
  expr: <Query that evaluates to a boolean>
  for: <Length of time that this expression must be true>
  labels: # Used for routing
    <key>: <value>
  annotations: # Used for metadata and extra content.
    <key>: <value>
```

for waits for a certain amount of time before firing. Good for suppressing spikes.

labels allows additional identifying labels to be attached to the alert. Can be templated.

annotations allows for metadata to be attached to the alert.

Alert example 1: Downed instances

This example will check if the k8s API server is down.

```
- alert: ApiServerDown
  expr: 'absent(up{job="kubernetes-apiservers"}) or sum(up{job="kubernetes-apiservers"}) <
1'
  for: 1m
  labels:
    severity: warning
  annotations:
    summary: 'Kubernetes API server down for 5m'
    description: 'Our Kubernetes cluster is inoperable. User impact uncertain.'
```

Alert Example 2: Request Latency

This alert will fire if we have a high request duration.

```
- alert: FrontendLatency
  expr: 'histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[1m])) by
  (le,kubernetes_name)) > 2'
  for: 1m
  labels:
    severity: warning
  annotations:
    summary: 'frontend service: high latency'
    description: 'The frontend service has a 99th-quantile latency of {{$value}} s.'
```

Note the use of templates in the description. [documentation](#)

Recording Rules

Note how the query is starting to get quite complex.

Prometheus allows us to specify recording rules.

Recording rules are queries that are evaluated upon each scrape and written back to Prometheus.

They allow you to replace complex queries with names to make them more understandable when reading.

E.g...

```
- record: 'job:request_errors:rate1m'  
  expr: '(sum(rate(http_request_total{status_code=~"5..|error"}[1m])) by  
(kubernetes_name)) / (sum(rate(http_request_total[1m])) by (kubernetes_name))'
```

We can now create an alert like...

```
- alert: FrontendErrorRate
  expr: 'job:request_errors:rate1m{kubernetes_name="python-app"} > 0.1'
  for: 1m
  labels:
    severity: critical
  annotations:
    summary: 'frontend service: high error rate'
    description: 'The frontend service has an error rate (response code >= 500) of
    {{$value}} errors per second.'
```

How Are Rules Passed into Prometheus?

To pass these rules into Prometheus we define a set of rule files.

In K8s we're going to do this in a ConfigMap.

The general format of a rule file is:

```
groups:  
- name: <unique_name_of_group>  
rules:  
- record: ...  
- alert: ...  
- alert: ...
```

E.g...

```
groups:
- name: frontend
  rules:
  - record: 'job:request_errors:rate1m'
    expr: '(sum(rate(http_request_total{status_code=~"5..error"}[1m])) by (kubernetes_name)) / (sum(rate(http_request_total[1m])) by (kubernetes_name))'
    alert: FrontendErrorRate
    expr: 'job:request_errors:rate1m{kubernetes_name="python-app"} > 0.1'
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: 'frontend service: high error rate'
      description: 'The frontend service has an error rate (response code >= 500) of {{$value}} errors per second.'
    - alert: FrontendLatency
      expr: 'histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[1m])) by (le,kubernetes_name)) > 2'
      for: 1m
      labels:
        severity: warning
      annotations:
        summary: 'frontend service: high latency'
        description: 'The frontend service has a 99th-quantile latency of {{$value}} s.'
```

Then we can pop that into a ConfigMap like this...

```
---
apiVersion: v1
data:
  frontend-rules.yml: |
    groups:
    - name: frontend
      rules:
      - ...
  kubernetes-rules.yml: |
    groups:
    - name: kubernetes
      rules:
      - ...
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: prometheus-rules
  namespace: monitoring
```

How is that Passed into Prometheus?

Just a new volume:

```
spec:  
  template:  
    spec:  
      containers:  
        - name: prometheus  
          ...  
          volumeMounts:  
            - name: rules-volume  
              mountPath: /etc/prometheus-rules  
      volumes:  
        - name: rules-volume  
          configMap:  
            name: prometheus-rules
```

And settings in `prometheus.yaml`:

```
prometheus.yaml: |  
  rule_files:  
    - "/etc/prometheus-rules/*.yaml"
```

Then hopefully, when you click on the `Alerts` tab in Prometheus you should see:

A screenshot of the Prometheus interface showing the 'Alerts' tab selected. The top navigation bar includes links for 'Prometheus', 'Alerts', 'Graph', 'Status', and 'Help'. Below the navigation is a section titled 'Alerts' containing three items, each with a colored background and text: 'FrontendLatency (1 active)' with a pink background, 'FrontendErrorRate (1 active)' with a yellow background, and 'ApiServerDown (0 active)' with a green background.

Alert Name	Status
FrontendLatency	1 active
FrontendErrorRate	1 active
ApiServerDown	0 active

Alertmanager Concepts

- **Grouping:** Group alerts so that only one is fired for a large event. E.g. network partition (you don't want a thousand alerts)
- **Inhibition:** Suppress alerts if other alerts are already firing
- **Silences:** Mute alerts for a given time

Alertmanager Configuration

Warning: More yaml ahead!

Some of the key global configuration options. See the [documentation](#) for more info.

```
global:
  # Lots of global SMTP and URL options for receivers.

# Notification receivers.
receivers:
  - [<receiver>](https://prometheus.io/docs/alerting/configuration/#<receiver>) ...

# Routing alerts to notification receivers
route: [<route>](https://prometheus.io/docs/alerting/configuration/#<route>)

# A list of inhibition rules.
inhibit_rules:
  [ - [<inhibit_rule>](https://prometheus.io/docs/alerting/configuration/#<inhibit_rule>)
  ... ]
```

Setting up Alertmanager Receivers

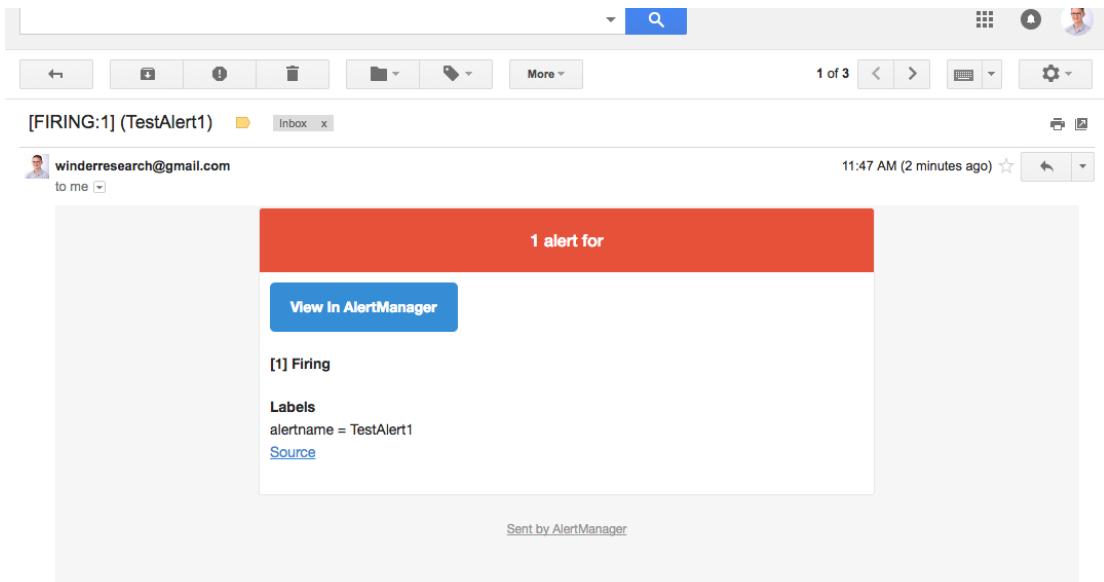
Receivers represent the tools and services that will receive your alerts.

Each has specific setting requirements so check out the [documentation](#).

- Note that no new receivers are being added. They've decided to just allow webhooks. Everyone is free to implement a service based upon that webhook.

```
receivers:  
- name: 'slack_chatbots'  
  slack_configs:  
    - send_resolved: true  
      api_url: 'https://hooks.slack.com/services/xxxxxx'  
      channel: '#chatbots'  
      text: "<!channel> \\nsummary: {{ .CommonAnnotations.summary }}\\ndescription: {{  
.CommonAnnotations.description }}"  
    - name: 'pager_duty'  
      pagerduty_configs:  
        - service_key: xxxxxxxxxxxxxxxxxxxx
```

Notes: Slack: note the text and templates. PagerDuty. Using Gmail and SMTP is cool for personal projects



Today

 AlertManager APP 12:21 PM
[FIRING:1] (TestAlert1)

Alertmanager Routes

Routs allow you to direct alerts to specific receivers based the alert's labels. [docs](#)

E.g. Different teams.

```
routes:  
# This routes performs a regular expression match on alert labels to  
# catch alerts that are related to a list of services.  
- match:  
    service: frontend  
    receiver: slack_chatbots  
    continue: true  
  
- match:  
    service: backend  
    receiver: pager_duty  
    continue: true
```

Alertmanager Inhibitions

Inhibitions allow you to selectively overrule alerts based upon their labels. [docs](#)

E.g. Severity.

```
inhibit_rules:  
- source_match:  
  severity: 'critical'  
  target_match:  
  severity: 'warning'  
  # Apply inhibition if the alertname is the same.  
  equal: ['alertname']
```

Inhibition is a quite advanced technique. I would recommend not providing inhibitors initially to prevent mistakes. Add only when you need to.

Alertmanager Extras

- You can provide external template files for your notifications. E.g. A generic template for slack messages, etc. [Good blog post](#)
- The Alertmanager is the one component that needs to be distributed in a mesh, since duplicate alertmanagers will send out duplicate notifications. We'll talk about this in the HA section.

Alerting Best Practices

- Keep it simple! As few alerts as possible.
- Alert on symptoms that affect your users
- Good consoles and dashboards to find causes. Link to them in the alert.
- Develop Playbooks/Runbooks that describe what errors mean, their likely causes, how to fix and who to ask for help

References:

- [My Philosophy on Alerting – Rob Ewaschuk](#)
- [Prom Docs](#)

Some common alerting use cases...

Alerting for Online Services

- High latencies
- Error rates

Offline Processing

- Duration of job

Batch Jobs

- Duration of jobs
- Failed jobs

Capacity Monitoring

- Close to capacity

Meta-Monitoring - Monitor the monitors!

- Alerts for Prometheus, alertmanagers, other monitoring infra

Hands on

502: Scaling and Availability

Now we're all experts at *using* Prometheus, we're going to use it for everything.

But what happens when you start scraping so many containers that you need to scale out?

And since monitoring is a key part of your infrastructure, how can we make it highly available?

That's what this section is about.



Scaling And Federation

First: It's been shown (depending on the number of metrics) that a single Prometheus server can support in the order of 1000 servers. (This number might be a lot higher with 2.0's new backend optimisation).

So only the largest clusters need to scale for performance reasons.

A more likely scenario is that you have multiple data centres or air-gapped clusters for any number of reasons.

In this case you can run a Prometheus instance per data centre.

Federation

Once you have multiple servers, you probably want to aggregate them into a "global" prometheus server.

This is called hierarchical federation.

Basically the trick is to run a third Prometheus that is scraping from the two primary Prometheus's:

```
- scrape_configs:
  - job_name: dc_prometheus
    honor_labels: true
    metrics_path: /federate          # Default path to get data from other Prom nodes
    params:
      match[]:
        - '{__name__=~"^job:.*"}'   # Request all job-level time series
    static_configs:
      - targets:
          - dc1-prometheus:9090
          - dc2-prometheus:9090
```

Here you can filter out all the non-critical metrics to avoid sending too much data over WANs or the internet.

Splitting

It is also a good idea to empower teams to provide them with their own Prometheus server, where it makes sense.

Each Prometheus would monitor its own stack. E.g. frontend, backend and machines.

Or this could be further split into domains.

Splitting ahead of time can help with isolation and coupling. E.g. Prevent scenarios like "I'm going to ignore that frontend alert because I'm working on the backend at the moment".

You can still federate some metrics to pull out some top level SLI's out of the various Prometheus', for example.

Sharding

Much later you might grow to the point where your scrapes are too slow because the load is too high.

When this happens you can enable *sharding*.

This is only required when a Prometheus instance is monitoring thousands of instances.

In general, I would recommend avoiding this as it adds complication. Imagine trying to generate queries with hundreds of thousands of metrics!

The idea is that you have several "slave" Prometheus servers that only scrape a certain subset of machines.

To do this we take a modulus of an address. If the modulus matches the current slave number, we scrape.

```
global:  
  external_labels:  
    slave: 1 # This is the 2nd slave. This prevents clashes between slaves.  
scrape_configs:  
  - job_name: some_job  
    # Add usual service discovery here, such as static_configs  
    relabel_configs:  
      - source_labels: [__address__]  
        modulus: 4 # 4 slaves  
        target_label: __tmp_hash  
        action: hashmod  
      - source_labels: [__tmp_hash]  
        regex: ^1$ # This is the 2nd slave  
        action: keep
```

Then we use federation to aggregate all the slaves:

```
- scrape_config:
  - job_name: slaves
    honor_labels: true
    metrics_path: /federate
    params:
      match[]:
        - '{__name__=~"^slave:.*"}' # Request all slave-level time series
    static_configs:
      - targets:
          - slave0:9090
          - slave1:9090
          - slave3:9090
          - slave4:9090
```

But again, I don't advocate this. Most people will never need to do it and it adds unnecessary complication.

High Availability

Prometheus can also be made highly available.

Note: High Availability (HA) is a distributed setup which allows for the failure of one or more services.

And thanks to Prometheus' simplicity, a HA mode is also very simple!

Remember that all Prom does is scrape targets and store them in a DB?

To make them HA, simply **run two Prometheus instances!**

Two scrape cycles, two databases. If one goes down, the other is still free to scrape!

Not Entirely True

It's not *just* about running two instances. Not quite, anyway.

The first problem is with dashboarding. The dashboards need to be able to connect to *either* Prometheus instance. If one goes down, the dashboard needs to use the other.

This requires some sort of load balancing.

Since we're using Kubernetes, we can simply put multiple Prometheus pods behind the same service.

If you're using something else then you might need to enable sticky sessions or something.

However, the slightly more tricky part is the Alertmanager.

Two Prometheuses will send two alerts to the Alertmanager.

That's fine, it will de-duplicate.

But we also need to make the Alertmanager HA too.

HA Alertmanager

We can make the Alertmanager HA by running multiple Alertmanagers. [docs](#)

But to make sure we don't send multiple alerts for the same problem, Alertmanager needs to communicate with the other.

This is unfortunate, because it requires the two Alertmanagers to be synchronised, which usually means complexity.

Prometheus Talks to Both

The first addition is that each Prometheus needs to talk to *both* Alertmanagers.

I.e. if one Alertmanager goes down, both Prometheuses can talk to the one remaining.

This is easy to achieve in practice because we're using k8s to discover pods. We can use that.

Secondly, the Alertmanagers need to communicate with each other, so we need to connect them.

Alertmanager has a (few) command line options to specify that:

```
-mesh.peer=alertmanager-2:9093
```

Which is actually quite annoying. We're using K8s for service discovery in Prometheus, but for alertmanager we're going to have to rely on DNS.

It uses the `Gossip` protocol to find other peers. So that means you only need to provide one address and it will find all the other instances through the protocol.

However, this is actually quite difficult in Kubernetes because `kube-dns` doesn't expose name-based dns entries (why!?).

So instead, we have to use two different separate k8s service.

But all of this is a moot point. If you really wanted a HA setup then they would be running in separate clusters.

So imagine two copies of deployments and services.

Then one has:

```
args:  
  - '-config.file=/etc/alertmanager/config.yml'  
  - '-mesh.peer=alertmanager-2.monitoring.svc.cluster.local:6783'
```

and the other has:

```
args:  
  - '-config.file=/etc/alertmanager/config.yml'  
  - '-mesh.peer=alertmanager-1.monitoring.svc.cluster.local:6783'
```

Hands On

503: Advanced Prometheus

In this final section I will try and cover some of the more advanced features of Prometheus.

However, I'm not going to cover them in depth, because they aren't a core part of the Prometheus use case.

Storage

Prometheus includes an on-disk time series database, but also integrates with other remote storage systems.

Prometheus isn't designed as a long-term data store. It defaults to only storing 15 days worth of data.

Prometheus should be viewed as an ephemeral sliding window of recent data.

If you need better persistency or longer durations, think carefully about what you are doing.

[Storage presentation - Pre 2.0](#)

Local Storage

- A custom format
- Blocks of two-hours
- All stored in one `/data/` directory: `--storage.tsdb.path` setting
- Two hour blocks are eventually compacted into longer blocks in the background.
- Defaults to 15d storage: `--storage.tsdb.retention` setting

It uses around 2 bytes per sample so you can calculate the total disk space required:

```
needed_disk_space = retention_time_seconds * ingested_samples_per_second *  
bytes_per_sample
```

[More information on 2.0 storage layer improvements here](#)

Remote Storage

Prometheus is not designed to store data. You should be able to wipe the data directory and carry on without any issues.

If you want a longer term storage solution, use one of the [remote storage integrations](#).

Notable data stores:

- Graphite
- InfluxDB
- OpenTSDB
- PostgreSQL

Security

As a rule of thumb, anyone with any access to any part of Prometheus has access to all the functionality and data within.

So all instances should be locked down on locked networks.

If you need external access then they should be placed behind API gateways or reverse proxies that implement authentication and authorisation.

Prometheus does not offer any encryption, authentication or authorisation. This is part of its simplicity.

Wire Formats

- Text format (we've used this the whole time)
 - Human-readable
 - Easy to assemble, especially for minimalistic cases (no nesting required)
 - Readable line by line (with the exception of type hints and docstrings)
 - Verbose
 - Types and docstrings not integral part of the syntax, meaning little-to-nonexistent metric contract validation
 - Parsing cost

- Protocol Buffer Format
 - Cross-platform
 - Size
 - Encoding and decoding costs
 - Strict schema
 - Supports concatenation and theoretically streaming (only server-side behavior would need to change)
 - Not human-readable

The text format is nice, human readable, but verbose. Consider moving to ProtoBuf if you have lots to scrape.

Kubernetes Operator

Even though we haven't tried it today (we concentrated on core prometheus) CoreOS have developed a [Prometheus Operator for Kubernetes](#).

This simplifies the initial setup of Prometheus on Kubernetes.

Definitely check it out if you're using this on Kubernetes.

Performance

Since the 2.0 upgrade, much of the "tweaking" that was performed to optimise memory and disk usage has gone.

CoreOS reports significant improvements in memory usage and memory consistency (good for container setups where we have to set limits!).

- Mem usage reduced by a factor of 3
- CPU usage reduced by a factor of 5
- Disk I/O reduced by factor of 80
- Storage space reduced by a factor of 5

Java Client benchmarks show that it takes around 1-20ms to write to a metric, depending on the type. source. So it is highly unlikely to cause you any issues on the client side.

Thats It!

- If you're interested in the Data Science, check out <https://TrainingDataScience.com>, an attempt to bring data science to Software Engineers by Phil Winder.
- If there is anything that you think should belong in this training, please let us know.
- If some of it was too hard, or too difficult, please let us know.
- I've also provided an appendix which has links to loads more documentation, talks and blog posts.

Thanks!

601: Further Reading

Taken from: <https://gist.github.com/mose/f392345e936a3953813ef74b05d83a7b>

Main links

- <https://prometheus.io/>
- <https://prometheus.io/docs/introduction/overview/>
- <https://github.com/prometheus>

Exporters

- Xen exporter https://github.com/lovoo/xenstats_exporter
- NFS exporter https://github.com/arnarg/nfs_exporter
- ZFS exporter <https://github.com/ncabatoff/zfs-exporter>
- Dovecot exporter: https://github.com/kumina/dovecot_exporter
- PowerDNS exporter: https://github.com/janeczku/powerdns_exporter
- Process exporter (/proc based): <https://github.com/ncabatoff/process-exporter>
- Generic script exporter: https://github.com/adhocteam/script_exporter
- Rabbitmq exporter: https://github.com/kbudde/rabbitmq_exporter
- System info exporter: https://github.com/prometheus/node_exporter
- Collectd exporter: https://github.com/prometheus/collectd_exporter
- Blackbox exporter: https://github.com/prometheus/blackbox_exporter
- Varnish: https://github.com/jonnenuauha/prometheus_varnish_exporter
- Grok (to extract data from logs) https://github.com/fstab/grok_exporter
- IPMI exporter https://github.com/lovoo/ipmi_exporter

Other tools

- https://github.com/prometheus/nagios_plugins a nagios plugin to query prometheus
- <https://github.com/cloudflare/unsee> alerts dashboard
- <https://github.com/ncabatoff/prombench> for generating a lot of scrape to benchmark volume
- <https://github.com/qvl/promplot> for generatic static plot graphs (for mail reports?)
- <https://coreos.com/operators/prometheus/docs/latest/high-availability.html> about high availability
- <https://gitlab.com/gitlab-org/gitlab-monitor> ruby code for a web exporter for application custom monitoring

- <https://github.com/prometheus/alertmanager#high-availability> if we need a deathstar
- <https://github.com/UnderGreen/ansible-prometheus> if we use ansible
- <https://github.com/weaveworks/grafanalib> grafanalib - python lib to create grafana templates
- <https://prometheus.io/webtools/alerting/routing-tree-editor/> alertmanager routes visualization and testing
- <https://github.com/weaveworks/cortex> prometheus as a service (could make sense for customers?) uses generic read+write remtoe
- <http://chromix.io> - <https://github.com/ChronixDB/chronix.ingester> - read-only for analysis
- <https://github.com/google/mtail> transform logs into metrics

Reading list

- <https://gitlab.com/gitlab-com/runbooks/blob/master/howto/monitoring-overview.md> prometheus architecture at gitlab
- <https://gitlab.com/gitlab-com/runbooks/tree/master/alerts> gitlab config for alerts
- <https://gitlab.com/gitlab-com/runbooks/blob/master/.gitlab-ci.yml> rules checker for prometheus in Gitlab CI
- <https://www.robustperception.io/scaling-and-federating-prometheus/> about scaling up
- <https://www.robustperception.io/federation-what-is-it-good-for/> about federation and aggregation
- <https://prometheus.io/blog/2017/04/10/prometheus-2.0-sneak-peak/> the next version of prometheus
- <https://fabxc.org/blog/2017-04-10-writing-a-tsdb/> details on what new storage in v2 will be based on

- <http://blog.alexellis.io/prometheus-monitoring/> general overview of a prometheus setup
- <https://www.digitalocean.com/community/tutorials/how-to-query-prometheus-on-ubuntu-14-04-part-1> PromQL missing doc
- <https://www.youtube.com/watch?v=MuHkckZg5Lo&list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz&index=7> remote storage
- <https://www.influxdata.com/prometheus-influxdb-thoughts/>
- <https://blog.acolyer.org/2017/03/10/chronix-long-term-storage-and-retrieval-technology-for-anomaly-detection-in-operational-data/> for long-term retention

Videos

- https://www.youtube.com/playlist?list=PL0z-W_CUquUlCq-Qohy53TolAhaED9vmU
promcon august 2016
- <https://www.youtube.com/playlist?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
cloudnativecon prometheus track april 2017

- https://www.youtube.com/watch?v=likpVWB5Lvo&index=4&list=PL0z-W_CUquUlCq-Oohy53TolAhaED9vmU digitalocean scaling prometheus on 2 millions of servers
 - one single prometheus server is ok until 1k or 2k nodes -> need sharding + prometheus proxy for grafana
 - more than 8G ram tuning is mandatory
 - alertmanager to replace nagios. but it has high availability lack.
 - sharding issue: add a shard, don't import old data. they thought using a kafka storage, now uses casandra store
 - give data to customers on per-vm basis
 - vulcan: fork of prometheus (on github) in advance on prometheus. designed to store on cassandra
 - downsampling done before cassandra, because need for 8 month data retention (10 cassandra nodes cluster for 40 metrics on a million machines)

- https://www.youtube.com/watch?v=Cvbc6oT1uUY&index=14&list=PLoz-W_CUquUlCq-Oohy53TolAhaED9vmU debian support for prometheus
 - prometheus, alertmanager and pushgateway are packaged. node-exporter, mysql exporter also in package. in unstable, and pass to testing in 5 days. testing packages work fine on stable.
- https://www.youtube.com/watch?v=XvqaYbiTOMg&list=PLoz-W_CUquUlCq-Oohy53TolAhaED9vmU&index=15 highly available alertmanager
 - work in progress (as of august 2016)
- https://www.youtube.com/watch?v=r6N5-1Jyifk&index=16&list=PLoz-W_CUquUlCq-Oohy53TolAhaED9vmU vulcan
 - the digitalocean api-compatible alternative
 - long-term storage
 - requires kafka, zookeeper, cassandra, elasticsearch, it's a lot

- https://www.youtube.com/watch?v=KoU_DquChS8&index=21&list=PLoz-W_CUquUlCq-Oohy53TolAhaED9vmU grafana master class
 - mostly basic information about grafana. good primer.
 - use annotations from search or state change
 - on templates, if using multiselect , need to use the =~ matcher
 - panel repeater using template variables
 - simple json datasource for pushing arbitrary events, may be of some use for mep process

- https://www.youtube.com/watch?v=yrK6z3fpu1E&index=22&list=PL0z-W_CUquUlCq-Oohy53TolAhaED9vmU alerting in prometheus
 - 4 golden: latency, traffic, errors, cause-based warnings (capacity/saturation)
 - avoid static thresholds: make threshold relative to context (ex. errors relative to traffic)
 - predict_linear for predictive alerts
 - alerts grouping (includes summary) and alert inhibition (alerts dependencies) for less alerts traffic
 - anomaly detection with holt_winters

- https://www.youtube.com/watch?v=b5-SvvZ7AwI&list=PLoz-W_CUquUlCq-Oohy53TolAhaED9vmU&index=5 labels
 - regexp on labels, relabelling
 - metric relabel are another type, after scrape, just how data is stored

- <https://www.youtube.com/watch?v=KXq5ibSj2qA> write an exporter
 - when app doesn't have a /metrics in prometheus format
 - ipmi, snmp and blackbox
 - metrics naming:
 - need to include the unit. use seconds instead of milliseconds
 - need to have explicit prefix
 - need suffix with type of data (_sum or _count)

- `_ratio` 0 to 1
 - `_total` is a counter, gauge should not have suffix
 - process *and* scrape prefix are reserved
 - user label partitionning when there are various values (like disk free) except latency
 - min, max, stddev are useless
 - return a 500, scrape_up will be 0

- <https://www.youtube.com/watch?v=gNmWzkGViAY> talk about borgmon from google which is totally same as prometheus
- <https://www.youtube.com/watch?v=NFPGtbQfL1A> what gitlab does with prometheus (up to customer facing usage)
- <https://www.youtube.com/watch?v=MuHkckZg5Lo> about the remote write/read possibilities
 - this is experimental as the video was shot
 - there is sample bridge for influx, graphite, opentsdb
 - but those backends suck a bit:
 - influxdb open source version don't do clustering. But maybe we don't need clustering for db? remoting make dual-prometheus already that's something
 - opentsdb is backed by hadoop, hbase, java heavy machinery, hard to maintain
 - graphite brings yet another backend like cassandra or cyanite
- <https://www.youtube.com/watch?v=XQdEVY2l2eo> about high availability of the alert manager
 - UUID is generated
 - gossip protocol to share data about aggregation, alerting etc

- <https://youtu.be/67Ulrq6DxwA?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
counting with prometheus
 - why aliases make graph reload is not the same graph
 - when counter resets to 0, new value is added to old value because counters never decrease so if it decreases it's interpreted as a reset
 - irate stress the average on the last 2 values on the timexpan given: more accurate and spiky
 - increase() is just rate() multiplied by the number of seconds of the timestamp
 - histograms are expensive, as quantiles are
 - in rate() the range (timestamp) should at least be 2 or 3 scrape steps

- <https://youtu.be/XOdEVY2l2eo?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Alertmanager on Its Way to High Availability
 - using gossip for having alertmanagers communicate silences and notifications between other instances
 - using mesh from weaveworks
 - ensures at least one notif is sent out, sometimes gossip is hindered by network and then there is duplication
 - no master but a different treatment delay on each alertmanager is used to let gossip communicate its shit
 - memory usage of alertmanager is low because retention of alerts is low
 - when new alertmanager is popped up, it gets updated by gossip, plus gossip do consistency communication checks from time to time
- <https://youtu.be/jpb6fLQOgn4?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Understanding and Extending Prometheus AlertManager
 - good general overview of Alertmanager
 - alertmanager has its own api for third party notifications
 - there is a visual editor and tester for alertmanager routes

- <https://youtu.be/bfSMDERvkZY?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Grafana is Not Enough: DIY User Interfaces for Prometheus
 - grafana templates as code (in python) = grafanalib
<https://github.com/weaveworks/grafanalib>
 - interesting example for postmortems, custom dashboards with editable queries
- <https://youtu.be/MuHkckZg5Lo?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Integrating Long-Term Storage with Prometheus
 - remote storage is an old story (issue #10)
 - local storage can work in combination optoinaly
 - http + protobuf protocol -> bridge -> remote storage
 - influxdb, opentsdb and graphite are just example bridges
 - `remote_write` config directive just takes a destination url, or `remote_read` for read
 - read is centralized promql evaluation
 - future: federation of shards for reads
 - cortex is the flagship of remote storage
 - still experimental - remote will be only in next release 1.7

- <https://youtu.be/looY1OyGhok?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Prometheus: The Unsung Heroes
 - usage of mtail to scrape logs from brocade syslog (smaller footprint than grok_exporter)
 - usage of snmp_exporter, featuring a generator
- <https://youtu.be/hPC6oldCGm8?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz>
Configuring Prometheus for High Performance
 - at start prometheus increase memory usage, but at some point it plateau. finding where the plateau will be is hard to predict
 - storage.local.memory-chunks : chunks are current unpersisted chunks plus persisted and cached ones (total memory / 6)
 - max-chunks-to-persist (memory-chunks / 2)

- see <https://www.robustperception.io/how-much-ram-does-my-prometheus-need-for-ingestion/>
- prometheus 2.0 is totally rewritten and will have better memory management (to be released in 2017)

Saving the best until last...

!!!!Phil's Talk!!!!

- https://youtu.be/U_nQObcAxSk?list=PLqm7NmbgjUExeDZU8xb2nxz-ysnjuC2Mz
Monitor My Socks: Using Prometheus in a Polyglot Open Source Microservices
- about integrating metrics endpoints in microservices framework

