Monitoring Microservices with Prometheus - Workshops

102: Monitoring Docker

In this workshop you will log into your VMs (or locally if you've chosen to develop locally) and run some containers. You will apply some load and use docker's tools to inspect resource usage.

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
```

See What's Running

```
$ docker ps
```

(If on Linux) Look at `cgroup` Aggregates

Take a look at the files:

- `/sys/fs/cgroup/memory/docker/<longid>/cpuacct.stat`
- `/sys/fs/cgroup/memory/docker/<longid>/memory.stat`
- What resource metrics can you see?
- How do these numbers map to physical resources?

`docker stats`

Now try `docker stats`

```
$ docker stats --no-stream
```

Note: If the load test container has finished you might need to do a `docker-compose down` and rerun the command to start the container.

- What resource metrics can you see?

- How do these numbers map to physical resources?
- Do these map to the aggregates show above?

Scale It!

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

`docker top`

Find one of your containers and use `docker top` to inspect the processes.

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

103: cAdvisor

In this workshop we're going to start looking at some metrics that can be observed from our docker instances. For this we're using a tool called cAdvisor. It's a really nice, simple tool to expose various measurements about our system.

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Run cAdvisor

Linux:

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

View the Web UI

Browse to `http://<public ip address>:8080`.

- Investigate the various metrics. What do they mean? Ask questions. Read the documentation

Run the Load Test again

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

- What happens to the measurements now?

Take a look at the Prometheus API

Browse to `http://<public ip address>:8080/metrics`.

- Take a look through that text. Familiarise yourself. # 202: Using Prometheus

In this workshop we're finally going to get our hands on Prometheus.

You'll be starting the Prometheus container, getting used to the super-simple UI and writing some configuration files to scrape data from services.

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Start the Prometheus Container

Let's start by instantiating the Prometheus container and having a look around the UI.

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

- Browse to `http://<public ip>:9090`. Look around the UI.
- What can you see at `http://<public ip>:9090/metrics`?
- Can you plot some of that data?

Your First Prometheus Configuration

Write your own Prometheus configuration to scrape itself. Use the following yaml or crate one yourself:

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

Mount that configuration file in a Docker volume and run Prometheus again:

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

- What has changed? Right, nothing. The default config scrapes itself! But at least you got it working.

Your Second Prometheus Configuration

Let's add cAdvisor so we can see what's going on with the containers. Run the following command to start cAdvisor again.

```
$ 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
```

- Now edit your Prometheus configuration and add cAdvisor as a job.
- Plot some of the results. Notice anything interesting?

Add Some Load!

First, bring up a plot of some CPU-related metric. Then run the load generator we used earlier.

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

- What happens to the CPU usage?! # 204: Instrumenting Applications

In this workshop you will be creating your own instrumented application!

I'm not going to provide much guidance here. Please feel free to try new things.

Example Repositories

To get you started, I've created some examples.

- **Python**: https://github.com/philwinder/prometheus-python
- **Java Spring Boot**: https://github.com/philwinder/prometheus-java-spring-boot

Please feel free to use/borrow/steal anything you like from there.

Client Libraries

If you need to check the client library documentation, you can find links to all the implementations here: https://prometheus.io/docs/instrumenting/clientlibs/

Your Turn!

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!

Some more ideas:

- Spin up Prometheus, plot the results of your instrumentation
- Add a:
  - Counter
  - Gauge
  - Histogram
  - Summary
- Try changing the default bin allocations for the histogram (important for long running jobs!)
- Try altering the Summary settings
- Add some random delays in there to make it more interesting
- Write a docker-compose file to start both your container and prometheus
- Push your code to Github to prove your awesomeness! :-D

Figure 1: monitor-all-the-things

301: Prometheus Queries

Queries are at the heart of monitoring. Only you, the Engineer that is responsible for maintaining the availability of your system is able to create the best queries.

This is not something that will happen only once. You will be tweaking queries over a long time.

You might have incidents which highlight that you weren't able to *see the right data*.

In this workshop you will practice writing some queries.

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Start The Containers

I have provided a handy docker-compose setup for you to practice your queries.

Please feel free to alter this to include your own applications.

Included in this setup are two load generators for the java and python webapps. I'm hitting the slow python endpoint, which will result in request durations that average around 0.5 s. The java one is just returning a string straight away so that should be able to handle thousands of requests per second.

```
$ cd <workshop directory>/301-queries
$ docker-compose up -d
```

- Make sure that all of the containers are running as expected with `docker-compose ps`. If they are not, troubleshoot.
- Make sure you can bring up the prometheus UI.
- Look at the targets and make sure they are all available
- Add your own containers to the setup, if you so wish.

Write Some Queries

I'd like you to try and do this by yourself, just using the documentation.

This is what you will be doing for real, so lets get some examples of this now.

Please feel free to ask for help if you need it.

At the very least, create two plots, showing:

- The Average Request Rate per second (**R**ED)
- The 95th percentile HTTP duration (RE**D**)

Note that you have a couple of jobs here, so you might want to pick one first for testing.

Challenges

If you manage to do that, then have a go at these:

- Average Number of Request Errors per second (R**E**D)
- Add cAdvisor and monitor CPU and RAM performance
- What would you like to see if this was your service? # 302: Data Science

In this workshop we're just going to have a little play with visualising the difference between the average and the median.

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Start The Containers

Let's reuse the previous example. The python app has a very special random generator in the `/slow` API and we're going to use that to perform an experiment.

```
$ cd <workshop directory>/301-queries
$ docker-compose up -d
```

The Mean

First, generate a plot in the prometheus UI of the average HTTP request duration. Pick a suitable time frame.

The Median

Next, generate another plot in the prometheus UI of the **median** HTTP request duration.

- What do you see?
- Is there a difference?
- Imagine you were creating an anomaly detector for an alert. Which would you use?
- Where would you set the threshold of your anomaly detector? Why? (Think about the difference between the population and what you are observing - a sample). # 303: Prometheus Exporters

Prometheus Exporters are *sidecars* that probe a technology for metrics and expose them in a Prometheus-friendly format.

In this workshop you will be using an exporter to monitor a Redis instance.

Figure 2: redis

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Start The Containers

Located in another handy docker-compose file is a Redis instance, the Redis exporter and a Redis load test container (and Prometheus, of course!).

```
$ cd <workshop directory>/303-prometheus-exporters
$ docker-compose up -d
```

- Make sure that all of the containers are running as expected with `docker-compose ps`. If they are not, troubleshoot.
- Make sure you can bring up the prometheus UI.
- Look at the targets and make sure they are all available

Metrics

- Take a look at the metrics exposed by Redis.
- Try plotting some interesting metrics.
- Find a metric that corresponds to the number of requests per second and plot the rate. # 304: Monitoring Kubernetes

This is what we've been working towards. Monitoring on Kubernetes.

But I think you'd agree after seeing those configuration files that if we did this straight away you would have been rather confused!

As of writing these materials, you will create a single node k8s cluster on your VM with a special script. This might change in the future, so check with your instructor.

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

Setup

This workshop requires a k8s cluster, so we need to set one up.

Create a Single Node K8s Cluster

Run the following script:

```
$ /tmp/installk8s.sh
```

Note, we're running that without `sudo`. Not as root. This is so the script can set the k8s configs for your user.

Wait for around 5 minutes

Then double check that the "cluster" is operational.

```
$ kubectl get nodes
NAME               STATUS    ROLES     AGE       VERSION
workshop-kskpaqi   Ready     master    2h        v1.8.3

$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                       READY     STATUS    RESTARTS   AGE
kube-system   etcd-workshop-kskpaqi                      1/1       Running   0          2h
kube-system   kube-apiserver-workshop-kskpaqi            1/1       Running   0          2h
kube-system   kube-controller-manager-workshop-kskpaqi   1/1       Running   0          2h
...
```

Everything should be Ready, running, green, etc. If it's not, troubleshoot.

Note that I've removed all requests and limits to allow all the containers to fit on this measly 1 CPU machine. :-)

Launch the Prometheus Stack

This is a concatenation of the manifest that you saw in the slides. It *should just work*. (Famous last words).

```
$ cd <workshop directory>/304-kubernetes
$ kubectl apply -f manifest.yml
```

Wait for around 5 minutes

It will take a while to download the containers.

Again, we need to make sure that everything has started ok:

```
$ kubectl --namespace monitoring get deployments
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-state-metrics   2         2         2            2           42m
prometheus-core      1         1         1            1           42m

$ kubectl --namespace monitoring get pods
NAME                                  READY     STATUS    RESTARTS   AGE
kube-state-metrics-694fdcf55f-xqb9p   1/1       Running   0          42m
kube-state-metrics-694fdcf55f-zv8xq   1/1       Running   0          42m
node-directory-size-metrics-4bml7     2/2       Running   0          42m
prometheus-core-7666789646-grpv9      1/1       Running   0          42m
prometheus-node-exporter-48vp4        1/1       Running   0          42m
```

Everything should be Ready, running, green, etc. If it's not, troubleshoot.

Find the NodePort

```
$ kubectl --namespace monitoring get svc
```

And copy the nodeport.

Browse to Prometheus

Visit `<public_ip>:<NodePort>` in your browser. You should be able to see Prometheus.

The public IP is the same as the one that you used to ssh into the machine.

Let's Go!

Finally, we have a running system.

Because you're Prometheus Pro's now, I'm not going to go through everything step by step.

Instead, try and achieve the following things:

- Add the scrape annotation to pods
- Plot the CPU usage of all the pods
- Monitor the disk usage of the nodes
- Add your own service that you created (or the python example)
  - Create a deployment
  - Add the annotation
  - Plot the metrics
- Add an ingress and see if the blackbox exporter is working
- Anything else that you are particularly interested in. Feel free to go off-piste.

Note: The trainer probably doesn't know about every metric listed! :-D

401: Visualisation 1 - Grafana

We're finally getting to the pretty stuff!

Grafana is a very comprehensive dashboarding tool. Possibly even too complicated for simple jobs.

In this workshop we're going to load up an instance of Grafana and include our Python app from earlier.

Note that we've got a lot of containers running now, so your system may start to creak!

Setup

You *should* be ok just overwriting the previous configuration. But if you have made any changes, you might need to start from scratch.

If so, run:

```
kubectl delete ns monitoring
```

This should remove all the old stuff.

Install Grafana

```
$ cd <workshop directory>/401-grafana
$ kubectl apply -f manifest.yml
```

Again, *hopefully* it should just work. If not, troubleshoot.

Make sure everything is up and running and healthy before you try and access the web UIs.

Browse to Grafana

Get the nodeport for Grafana and browse to the UI.

```
$ kubectl --namespace monitoring get svc
```

Login

The default `username:password` is `admin:admin`.

Tasks

- Get yourself used to the UI
- Browse around
- View the provided dashboards
- Change the timescale
  - Go to the top right and click the little clock icon.
  - Select the `Last 15 minutes`
  - At the bottom left, select `Refreshing Every: 10s`.

Advanced Tasks

- Create a Dashboard
- Create a Plot showing the average request rate of the `python-app` app.
- Create a plot showing the 95% percentile request duration of the `python-app`
- Add Some statistics about uptime and availability.
- Visit https://grafana.com/dashboards and see if there are any dashboards you'd like to download. Anything to do with K8s?
- Visit https://grafana.com/plugins and see if there are any interesting plugins.

Bonus points for the coolest dashboards!

402: Templating

You can use Prometheus to render Go-based HTML templates on the server side. All templates are available at: `http://<public_ip:9090/consoles/<template_filename>`.

Again we use Kubernetes ConfigMap's to store our template definition. This allows us to version control the template. If you did this for real, you wouldn't put it all in one giant file!

References

You'll probably need take a look at the documentation to do anything cool. Here are some references:

- Go Templates
- Prometheus Function Reference
- More examples

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

```
$ cd <workshop directory>/401-grafana
$ kubectl delete -f manifest.yml
```

Install Prometheus With Example Templates

```
$ cd <workshop directory>/402-templating
$ kubectl apply -f manifest.yml
```

Browse to:

- `http://<public_ip:9090/consoles/simple.html`
- `http://<public_ip:9090/consoles/a.html`
- `http://<public_ip:9090/consoles/b.html`
- `http://<public_ip:9090/consoles/c.html`

To see the examples I showed in the slides.

You can see these defined in the manifest: `402-templating/manifest.yml#L429`.

Tasks

- Add a new template. Start simple.
  - Add your template to the manifest
  - Apply the new template
  - Restart the prometheus pod to force a configuration refresh

_Note: you can also refresh the config on a live system with: `curl -X POST http://<public_ip>:9090/-/reload`

- Make the template more complex. Again, try and recreate something that would be useful to you in production.
- Add your app and add a template for that.

You're only limited by your imagination here. Go wild.

A Little Crazy

- Ajax requests to the k8s API server to scale up your pod?
- Dropdown box to allow you to select different K8s namespaces to monitor?
- Monitoring Roulette: A button to present a random metric… ## 501: Alerting

In this fun workshop you will be creating an alerting system!

There are two key components:

- Alerting rules inside prometheus
- Alertmanager tool to publish alerts.

First we'll create some alerting rules using our Kubernetes configs, then we'll setup alert manager and finally your tasks will be to integrate alertmanager with the k8s configs and produce an end-to-end monitoring system!

Clean up

Remove all previous containers/k8s pods/docker-compose's etc.

```
$ kubectl delete ns monitoring apps
```

Prometheus Alerting Rules

Spin Up My Example

In this example I provide three alert rules then add that ConfigMap as a volume.

We're still using the Python app as an example. As always, feel free to add your own.

```
$ cd <workshop directory>/501-alerting
$ kubectl apply -f manifest.yml
```

- Browse to Prometheus and click on the Alerting tab. Check that the alerts are there.
- Are the queries working for those alerts? Click around and double check.
- Do any of those alerts fire?

Adding Your Own Alerting Rules

- Add a new alert rule that alerts when the python-app is down
  - Delete the python-app deployment to test that it is working
- Add a new recording rule that alerts when the number of received requests is low
  - Call the recording rule in the query browser to check it is working
- Add a new alert that uses the previous recording rule
  - Delete the python-load deployment to test that it is working

Alert Manager

First we're going to make sure the alert manager configuration works by running it as a simple docker container. Once it's working you can port it to Kubernetes.

There are two tasks:

- Alerting via your personal gmail account
- Alerting via Slack

If you can't do one of them for whatever reason, don't worry too much. But you should try and find some way of making at least one receiver work.

E.g. Setup a new personal slack channel, it's free and quick to do.

Using Gmail To Send Alerts to Yourself

To setup alerts using gmail's smtp servers you'll need to create an `App Password` https://myaccount.google.com/apppasswords.

If you try and use your root account password, google will flag it as unauthorised activity.

To create an `App Password` you **must have 2-factor authentication turned on**.

Create the alertmanager settings file inside your VM:

```
GMAIL_ACCOUNT=XXXXXXX@gmail.com # Add your email address
GMAIL_AUTH_TOKEN=XXXXXXX        # Substitute in your app password

cat <<EOF > alertmanager.yml
route:
  group_by: [Alertname]
  # Send all notifications to me.
  receiver: email-me

receivers:
- name: email-me
  email_configs:
  - to: $GMAIL_ACCOUNT
    from: $GMAIL_ACCOUNT
    smarthost: smtp.gmail.com:587
    auth_username: "$GMAIL_ACCOUNT"
    auth_identity: "$GMAIL_ACCOUNT"
    auth_password: "$GMAIL_AUTH_TOKEN"
EOF
```

Run the alertmanager and turn on debugging.

```
docker run -d --name alertmanager -p 9093:9093 -v $PWD/alertmanager.yml:/etc/alertmanager/co
```

Make sure the container is running ok:

```
docker logs -f alertmanager
```

Now generate an alert by hitting the API

```
curl -H "Content-Type: application/json" -d '[{"labels":{"alertname":"TestAlert1"}}]' localh
```

Note: It takes a minute for the email. Prometheus has a `group_wait` option in the routes to de-duplicate multiple alerts before sending.

Using Slack to Send Slack Messages to Your Team

We need to install the `Incoming Webhooks` Slack app.

1. Login to the Slack web UI
2. Browse to https://slack.com/apps/A0F7XDUAZ-incoming-webhooks
3. Make sure you have the right slack team selected.
4. Click "Install" if you don't have it installed. If you do you'll see"Add configuration".
5. In the `Post to Channel` card, select the default channel you want to post to. Then click on the confirmation button.
6. Copy the Webhook URL

Now use the folling configuration. Paste this on your VM. Change the webhook url. You also might want to change the channel to `@<yourname>` to prevent spamming everyone.

```
WEBHOOK_URL=XXXXXXX # Your webhook URL

cat <<EOF > alertmanager.yml
route:
 group_by: [cluster]
 # If an alert isn't caught by a route, send it slack.
 receiver: slack_general
 routes:
  # Send severity=slack alerts to slack.
  - match:
      severity: slack
    receiver: slack_general

receivers:
- name: slack_general
  slack_configs:
  - api_url: $WEBHOOK_URL
    channel: '#general'
EOF
```

Run the alertmanager and turn on debugging.

```
docker run -d --name alertmanager -p 9093:9093 -v $PWD/alertmanager.yml:/etc/alertmanager/co
```

Make sure the container is running ok:

```
docker logs -f alertmanager
```

Now generate an alert by hitting the API

```
curl -H "Content-Type: application/json" -d '[{"labels":{"alertname":"TestAlert1"}}]' localh
```

Note: It takes a minute for the message. Prometheus has a `group_wait` option in the routes to de-duplicate multiple alerts before sending.

Ingegrating Alertmanager with Kubernetes

Now, integrate this configuration with your Kubernetes manifest.

Service Discovery

There's only one tricky bit and that's setting the Alertmanager discovery options in the Prometheus configuration. docs

Note that the docs appear to be wrong here (as of 12/11/17). It is not possible to specify `alertmanager.url` as of 2.0.

This is what we need to add:

```
alerting:
  alertmanagers:
  - kubernetes_sd_configs:
      - role: pod
    relabel_configs:
    # Drop all pods except the pods not in the monitoring namespace
    - source_labels: [__meta_kubernetes_namespace]
      regex: monitoring
      action: keep
    # Drop all pods except the pod named: alertmanager
    - source_labels: [__meta_kubernetes_pod_name]
      regex: 'alertmanager.*'
      action: keep
    # With a non-empty port
    - source_labels: [__meta_kubernetes_pod_container_port_number]
      regex:
      action: drop
```

This adds an alertmanager that's using a Kubernetes service discovery mechanism (lots of others exist too - docs) and filters for the pods that are in a namespace of `monitoring`, the pods are named `alertmanager.*` (the regex is because k8s appends a UID to the pod names) and it must have a non-empty port.

We don't need to specify k8s certificates here because we are actually running inside the cluster.

You can check to see if Prometheus has found your alertmanager by looking at the bottom of `http://<public_ip>:31503/status`. An endpoint should be listed.

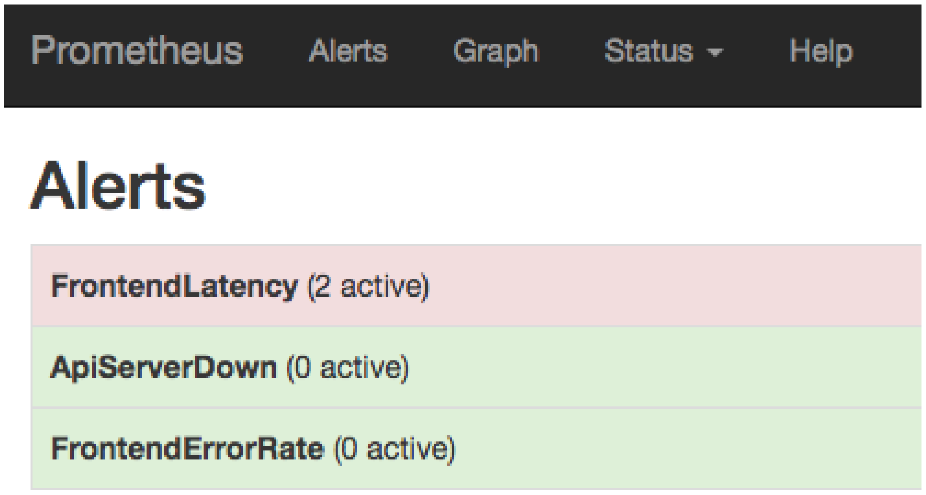
Over to You!

You should be able to figure everything else out.

Tasks:

1. Integrate the Alertmanager with Kubernetes.
    1. Test that your alertmanager config works with the standalone test (see gmail or slack examples)
    2. Add the alertmanager configuration
    3. Add the ConfigMap as a volume.
    4. Alter the Prometheus configuration to point to your alertmonitor
    5. Make sure Prometheus can communicate with Alertmanager docs - see above
2. Add alert rules to the prometheus configuration
    1. Make sure your queries work first
    2. Add them to the config
    3. Check that they are read by Prometheus by browsing to `http://<public_ip>:31503/alerts`
    4. Check that they are firing (if you expect them to)
3. Sit back and watch the alerts roll in!

Your setup should show something like the following images:



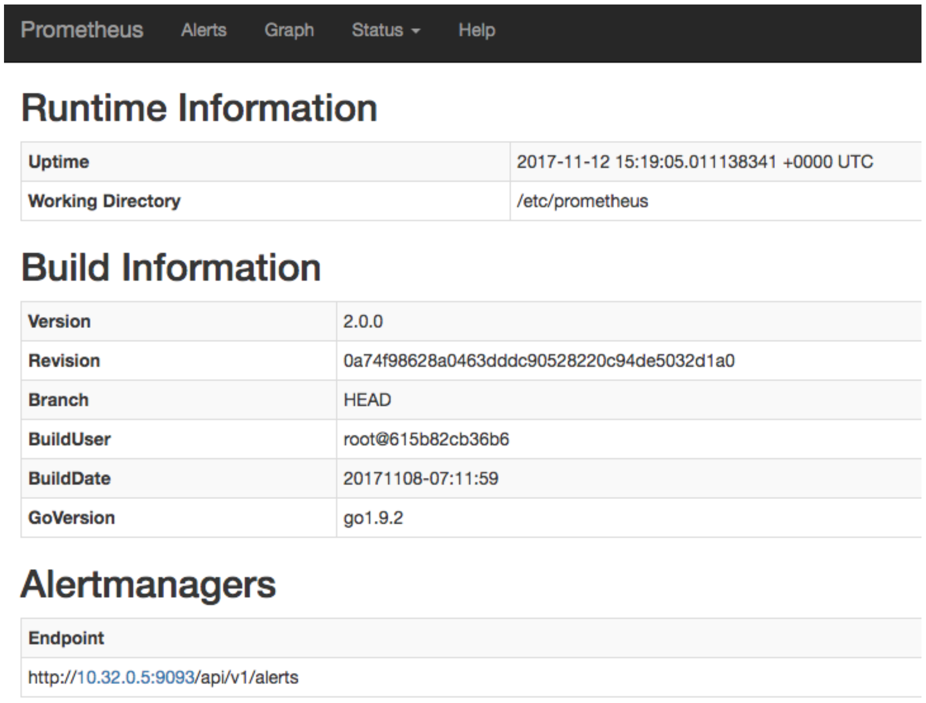
Alerts

FrontendLatency (2 active)

ApiServerDown (0 active)

FrontendErrorRate (0 active)

Figure 3: alertmanager-working-alerts



Runtime Information

Uptime	2017-11-12 15:19:05.011138341 +0000 UTC
Working Directory	/etc/prometheus

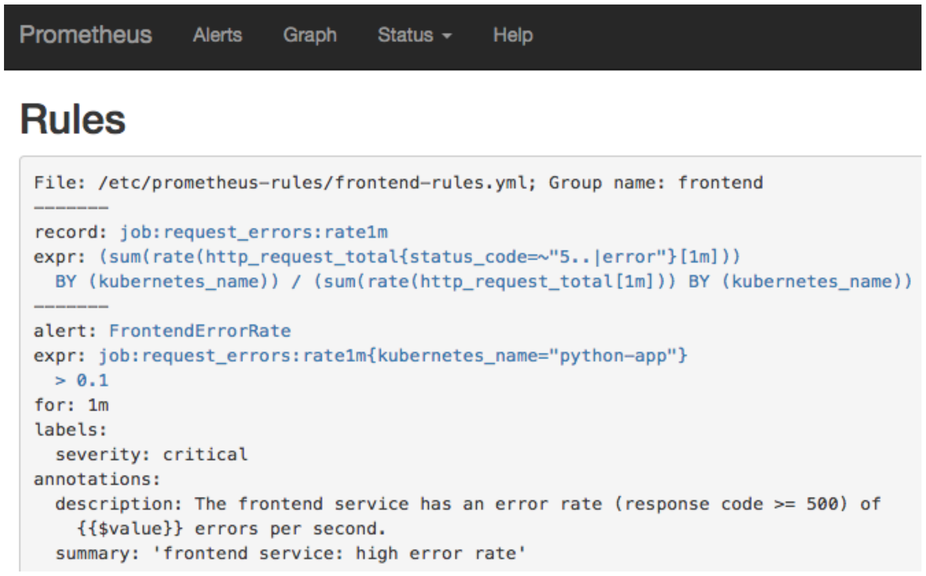
Build Information

Version	2.0.0
Revision	0a74f98628a0463dddc90528220c94de5032d1a0
Branch	HEAD
BuildUser	root@615b82cb36b6
BuildDate	20171108-07:11:59
GoVersion	go1.9.2

Alertmanagers

Endpoint
http://10.32.0.5:9093/api/v1/alerts

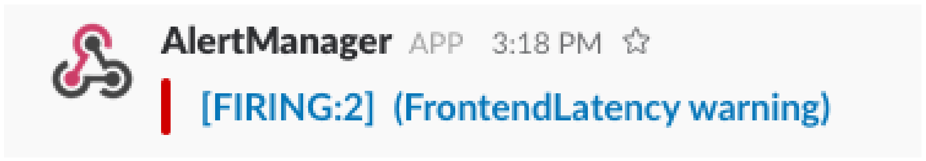
Figure 4: alertmanager-working-config



Rules

```
File: /etc/prometheus-rules/frontend-rules.yml; Group name: frontend
--------
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:
  description: The frontend service has an error rate (response code >= 500) of
    {{$value}} errors per second.
  summary: 'frontend service: high error rate'
```

Figure 5: alertmanager-working-rules


AlertManager APP 3:18 PM

[FIRING:2] (FrontendLatency warning)

Figure 6: alertmanager-slack-alert-2

Help

If you get stuck, please feel free to ask for help.

If you get really stuck, then there is a working manifest in `501-alerting/manifest-alerting.yml` (except I have put some dummy credentials in there). # 502: Scaling and High Availability

In the slides we talked about two things.

- When, how and why to scale out Prometheus
- How to make Prometheus and Alertmanager highly available

In this workshop we're going to pretend that we have two teams and we want to setup federation.

Also we're going to look at a HA mode. This is a little contrived, as you wouldn't normally be running a HA setup in a single cluster. I.e. K8s is already pretty resilient; it will restart your alertmanager pod if it fails and send the alert when it starts.

In other words, if you do this for real, make sure your second alertmanager instance is separated from your cluster. Otherwise if your cluster went down, you'd lose both instances.

Federation

I'm not going to provide too much help on this one, because you should find it relatively easy. There's only a few new lines of configuration. The rest is copy/pasting from previous exercises.

Task:

- Create two instances of Prometheus (it's up to you how you do this. Separate namespaces, just another deployment and service, whatever.
- Create a third instance and federate the the first two.

So in other words create two Prometheus instances just like you have been doing before. Then when creating the federating Prometheus, you'll want to alter the settings slightly to federate the first two (and not do any other scraping):

```
- scrape_config:
  - job_name: federated_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
```

For the targets you could either use the service DNS addresses or you can use the Kubernetes service discovery definition.

High Availability

This one is a bit more tricky because of the DNS requirements of the Alertmanager.

The best way I thought of doing it in this environment was to create two separate Alertmanager deployment and services, then use the service name DNS entry.

Tasks: - Take the manifest from `501-alerting/manifest-alerting.yml` and replicate the alertmanager deployment specification and service specification. - Add - Add the `mesh.peer` arguments to each of the deployments (see below) - Add port `6783` to each of the services and deployments (this is the Gossip protocol port)

```
args:
  - '-log.level=debug'
  - '-config.file=/etc/alertmanager/config.yml'
  - '-mesh.peer=alertmanager-1.monitoring.svc.cluster.local:6783'
  - '-mesh.peer=alertmanager-2.monitoring.svc.cluster.local:6783'
```

If you get stuck, ask for help. Or check out the version I implemented in `502-ha/manifest.yml`.