Observability and Autoscaling

Cloud Native - Week 5

Contents of this lecture

- What is observability and why does it matter?
- Observability in the Cloud
- Observability in Kubernetes
- Measuring stuff
- Graphing stuff
- Scaling your software based on this "stuff"

What is observability?

Where we started: an app on localhost

When the app ran on our machine:

- java -jar ...
- **Logs** in our terminal
- **Stacktraces** right there when things go wrong
- http://localhost:8080 in the browser
- Debuggers in our IDE!
- Can watch its performance with top, etc..

BUT

- No one else can access our server so what's the point?!
- No release management at all
- No scalability at all beyond the bounds of our PC

Not really an accesible application..

So we moved it into "The Cloud" ™

So Fancy

Where we started: an app in a VM (with a sprinkling of Docker)

Now:

- Other people can access our application!
- Code released and deployed via Docker
- Docker keeps everything running
- Scaled vertically to as many CPUs as we can get our hands on

BUT

- We have **ABSOLUTELY** no idea what it is doing.

  Best guess we can make is by acting as a user and seeing what breaks..

- Have to get everything else via SSH..
  - Logs
  - top
  - uptime
- Not highly available
- Constant hardware use

So we moved it into Kubernetes

~~Because we saw it on HackerNews~~

Where we started: an app in Kubernetes

Now:

- It. Scales.
  - Up *and* down
- Highly Available (database and app)
- Access to tools in Kubernetes
  - kubectl logs (some buffered logs)
  - Events, resource quotas, replicas

This is complexity.

But critical to note

As we've moved the app between these compute locations we've lost our intimate view of what the app is doing at any point in time.

- We still can't really tell what it's doing
- We still can't really tell what it's done in the past
- We can't really draw any conclusions on **what it will do in the future**
- These problems just grow as we run more and more apps.

This brings us the area of *observability*:

> *"Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs"*

Observability covers a huge range of topics

- Logging
- Monitoring & metrics
- Diagnostics
- Health checks
- Debugging
- Fault injection
- Etc etc..

We'll look at a few of them in this session.

Lets look at **logging** first

```
2018-01-01T00:00:00 INFO com.blah.service.myclass 1231242 12 Processed GET request to / in 28.12312 seconds.
```

Old school : open and write to file

Using either the shell or your own file IO, open a file and continue appending bytes to it as your app runs.

- Very fragile
- Will happily eat your entire hard drive
- *Very* large log files for large apps, even when compressed
- You lose the logs if you lose the machine

Have to use esoteric combinations of grep head cut zgrep to find the things you're looking for.

Slightly better : logrotate

Logrotate is a tool traditionally run on a cron job to copy and truncate the log file your application is logging to. Your app doesn't need to do anything special itself.

- No longer eats your entire harddrive!
- Still creates very large files!
- Still keeps the logs on the same machine

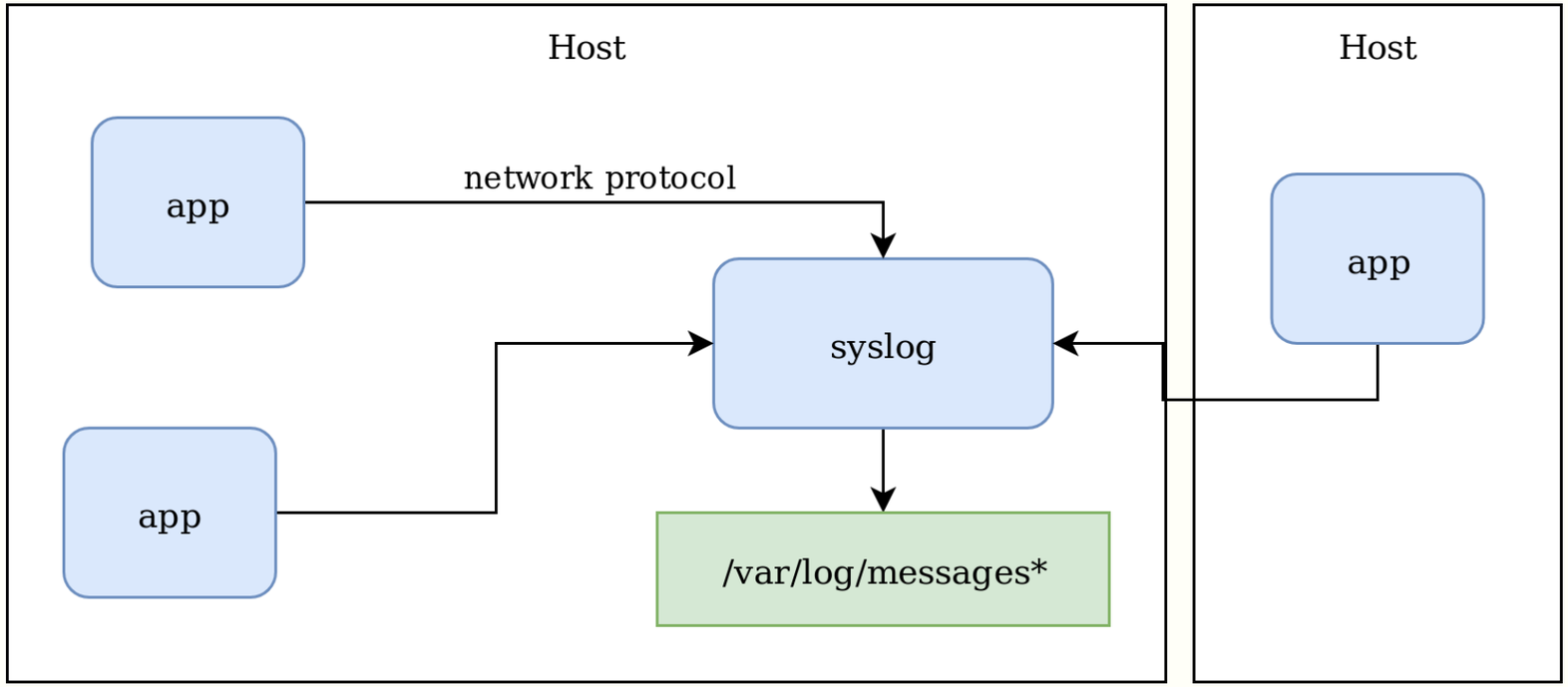
```
$ ls -1 /var/logs/blah
blah-20180101T000000.gz
blah-20180101T010000.gz
blah-20180101T020000.gz
blah-20180101T030000.gz
blah-20180101T040000.gz
blah-20180101T050000.gz
blah-20180101T060000.gz
...
```

Still hard to deal with or analyse - we need to get these logs shipped of to a central location.

Aggregating logs

In distributed or complex systems log lines are useless in isolation. We need to be able to deal with them in one place in order to really diagnose things.

Many Linux components integrate with a system called syslog. A service that can receive logs from many services on the same machine (or other remote machines) and aggregates them after applying transformations, filtering, or forwarding. It also handles rotation.



Modern and self-hosted : E.L.K

- Very common and popular now (for good reason)
- **ElasticSearch** - a structured document store that handles rich **queries and indexing** (very powerful and not specialised for logs)
- **Logstash** - reads your logs files, parses them into structured data, forwards them to ElasticSearch
- **Kibana** - web interface for ElasticSearch (also not specialised for logs)

Very nice stack to work with, however you usually have to deploy and run it yourself. *Which just adds to your concerns and responsibility.*

You can hosted ELK solutions! But you have to pay for them.

Here's a picture of an ELK

Using ELK in Kubernetes

We can either do this in the application layer or in the platform layer (imprecise defintions..).

Application layer

- Run Logstash in a side car container next to our app
- Forward to a copy of ElasticSearch that we run
- Run our own Kibana UI

Platform layer

- Configure Kubernetes to send *all* logs from *all* pods to a single large ElasticSearch pod.
- Very useful in the long run as we run more and more things.
- Gathers logs for our Kubernetes internals as well
- Consistent log access across all of your applications and pods
- Only works when we are admins on the cluster (eg: OKE)

We're going to go with the platform layer in this case

Default Kubernetes logging

- Kubernetes use the Docker container engine (normally)
- Docker uses json logging driver for stdout and stderr
- json files written to /var/log/...

When you use kubectl logs it directs the kubelet service on the pod's node to read and stream the lines back to the user.

This means that by deafult you have the same issues as you usually get with file-based logging!

- logs only stored in one location
- log rotation must be configured manually

Lets throw some ELKs at this

Or more specifically, E.*F*.K.s?

Sidebar : Where do Kubernetes logs go?

Kubernetes doesn't try to do anything special with logs by default.

All of the containers (including the Kubernetes internals) use the Docker json logging provider which writes lines to files on the host. The cluster admin has to hook up logrotate to keep these controlled.

```
$ cat /var/log/containers/kube-proxy-cbg9m_kube-system_kube-proxy-
6cddcb03ee65ce1710487926d7a2db522f70743de4dcc72ec26be707d2db47cc.log | tail -1 | python -m json.tool

{
    "log": "E1105 22:12:27.664064       1 reflector.go:205]
k8s.io/kubernetes/pkg/client/informers/informers_generated/internalversion/factory.go:129: Failed to list *core.Endpoints: Get
https://138.1.19.87:6443/api/v1/endpoints?limit=500&resourceVersion=0: net/http: TLS handshake timeout\n",
    "stream": "stderr",
    "time": "2018-11-05T22:12:27.664221684Z"
}
```

We're going to use Fluentd to read, parse and ship these lines.

Fluentd

Basically the same as LogStash

- Slightly better resource use
- More supported and used by the Kubernetes community

Let's install it!

```
$ export KUBECONFIG=~/.kube/uob.admin
$ git clone --branch=master https://github.com/kubernetes/kubernetes.git
$ cd kubernetes/cluster/addons/fluentd-elasticsearch
$ git reset 1cdc9059ba0c05bc6aae81f70bac44f269c28396

$ kubectl apply -f fluentd-es-configmap.yaml
$ # -> (modify ds file for OKE-specific volume paths)
$ kubectl apply -f fluentd-es-ds.yaml

$ kubectl label node --all beta.kubernetes.io/fluentd-ds-ready=true
```

Note: this is an important aspect of the Kubernetes community: common and understood manifests and Helm charts that can be deployed by anyone anywhere. You often need to fork and modify them for your specific configuration, but its better than nothing.

Now we have fluentd running on all our nodes and shipping our logs!

```
$ kubectl -n kube-system get pods | grep fluentd
```

```
fluentd-es-v2.2.0-9gtkv              1/1     Running   0          9m    10.244.6.11   132.145.38.234
fluentd-es-v2.2.0-cvzb8              1/1     Running   0          9m    10.244.2.13   132.145.41.147
fluentd-es-v2.2.0-fx6rn              1/1     Running   0          9m    10.244.1.16   132.145.53.105
fluentd-es-v2.2.0-g92vv              1/1     Running   0          9m    10.244.0.20   132.145.57.154
fluentd-es-v2.2.0-htlwz              1/1     Running   0          9m    10.244.7.12   132.145.24.27
fluentd-es-v2.2.0-p2qsg              1/1     Running   0          9m    10.244.8.9    132.145.19.242
fluentd-es-v2.2.0-qqxgc              1/1     Running   0          9m    10.244.4.12   132.145.32.222
fluentd-es-v2.2.0-smp7b              1/1     Running   0          9m    10.244.3.16   132.145.54.187
fluentd-es-v2.2.0-x4f4l              1/1     Running   0          9m    10.244.5.12   132.145.19.31
```

```
NAME                 DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR                          AGE
fluentd-es-v2.2.0    9         9         9       9            9           beta.kubernetes.io/fluentd-ds-ready=true   16m
```

ElasticSearch

The previous Fluentd deployment is configured to push logs to ElasticSearch at elasticsearch-logging:9200 so lets get that set up.

```
$ cd kubernetes/cluster/addons/fluentd-elasticsearch

$ kubectl apply -f es-statefulset.yaml
$ kubectl apply -f es-service.yaml
```

- **2 ElasticSearch Pods** - Active-Active cluster
- **Ephemeral storage** - only suitable for testing and learning - you should really replace it with real PersistentVolume storage!

```
elasticsearch-logging-0          1/1     Running   0          11m
elasticsearch-logging-1          1/1     Running   0          11m
```

```
$ kubectl -n kube-system port-forward svc/elasticsearch-logging 9200:9200
```

And we can watch our data begin to appear..

```
$ curl http://localhost:9200/_cat/indices?v

health status index               uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   logstash-2018.10.09 faJYUH77RGK8ynSM7wrkvw   5   1      19034            0     17.4mb        13.6mb
green  open   logstash-2018.11.02 NZopqdcWQ0eUfEW1Smc-aA   5   1        963            0       1mb       568.3kb
green  open   logstash-2018.10.01 2IXNhlrPRtyhs-aEr4FKjw   5   1         30            0     742kb         371kb
...
```

This can take some time, and can be seen as eventually consistent. Logs appear slowly as fluentd ingests the older log files and eventually catches up with everything.

Note that funnily enough fluentd uses the same index names as log stash in order to work better with existing tools.

And lastly, Kibana

More stateless = easier to deploy:

```
$ (edit deployment to remove BASEPATH envvar)
$ kubectl apply -f kibana-deployment.yaml
$ kubectl apply -f kibana-service.yaml
```

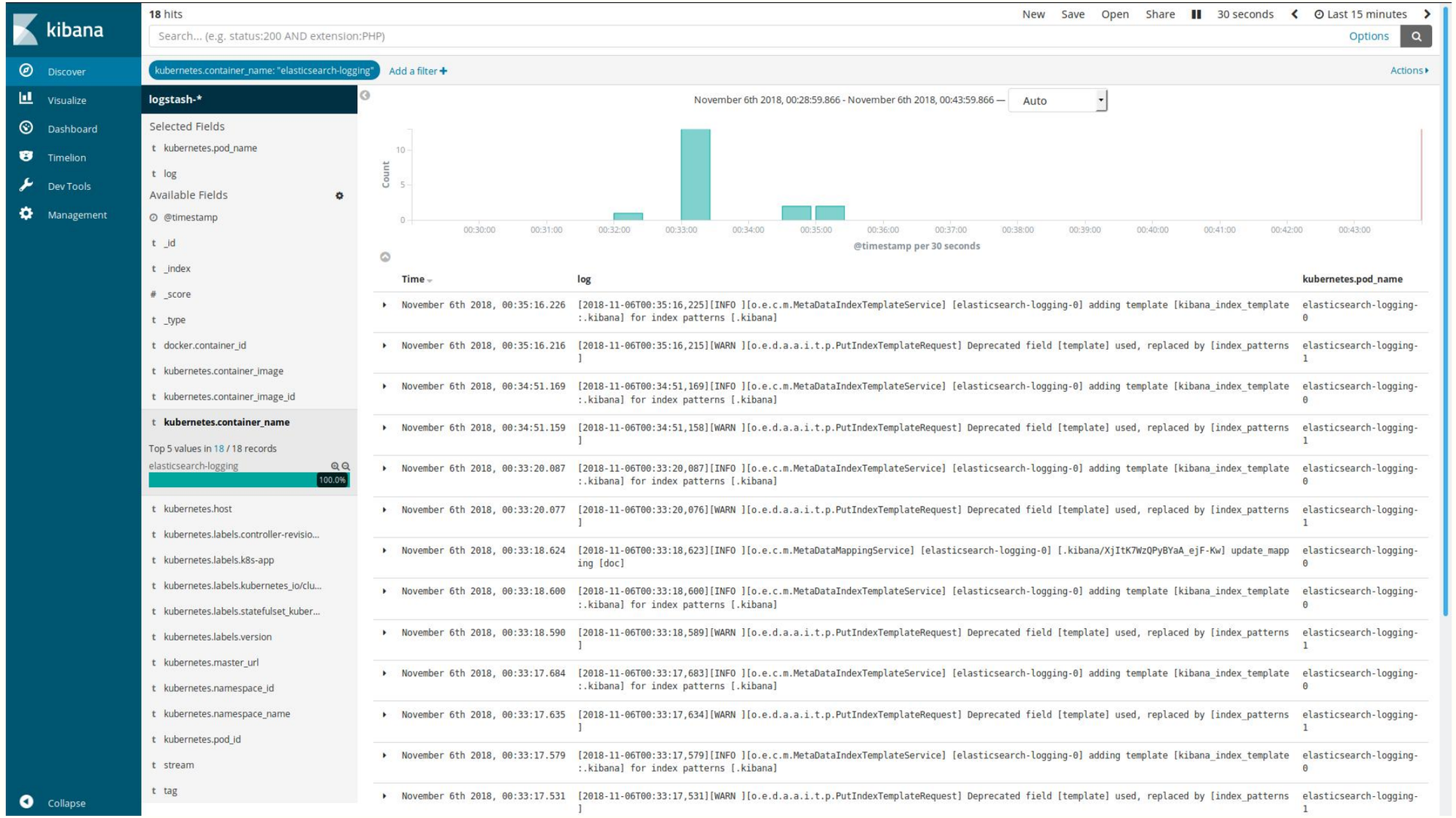
```
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kibana-logging   1         1         1            1           1m
```

```
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kibana-logging   ClusterIP   10.96.137.118   <none>        5601/TCP   1m
```

Now lets make this available via an Ingress route:

```
$ kubectl apply -f - << EOF
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kibana-logging-ingress
  namespace: kube-system
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.org/ssl-services: "kibana-logging"
spec:
  rules:
  - host: kibana.uob.example.local
    http:
      paths:
      - path: /
        backend:
          serviceName: kibana-logging
          servicePort: 5601
EOF
```

And after setting up kibana.uob.example.local in /etc/hosts, we can get our logs at http://kibana.uob.example.local/app/kibana.



Querying for logs from kubernetes.container_name=elasticsearch-logging in the last 15m.

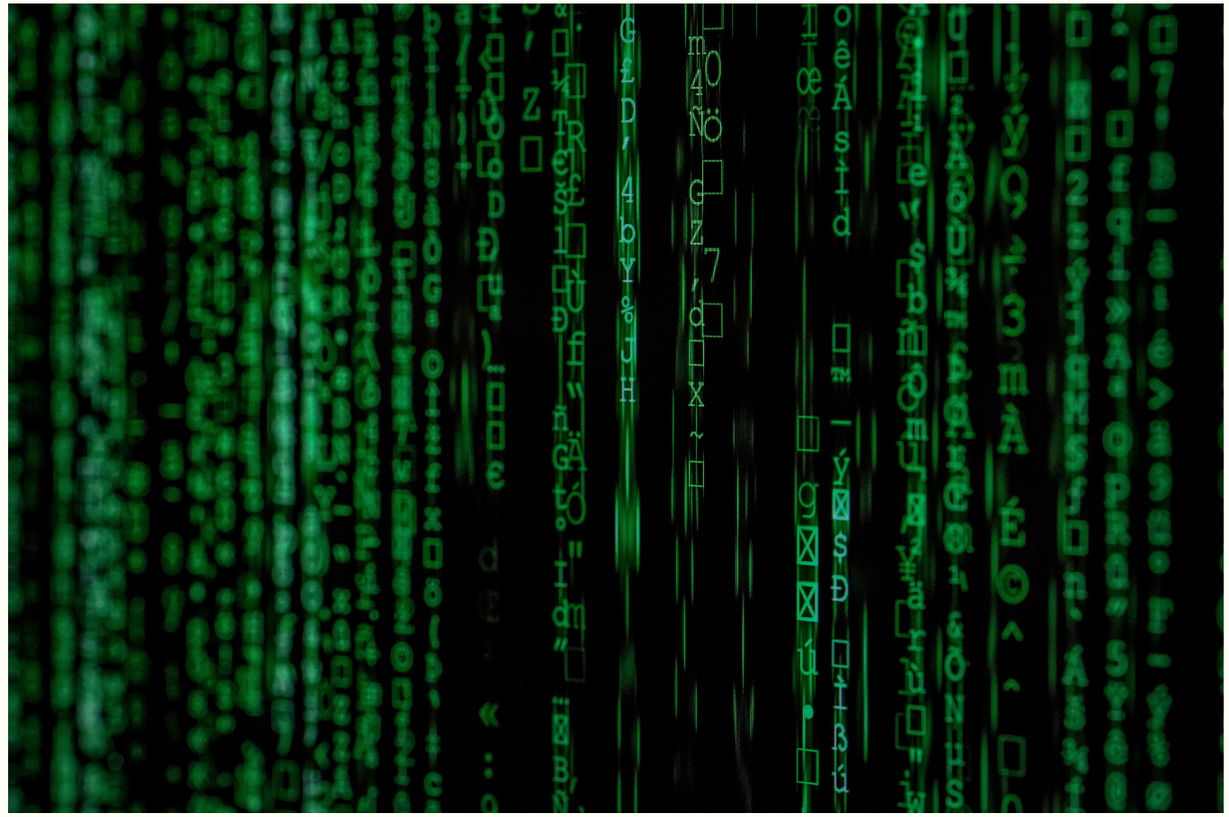
So what have we achieved?

- Logs from all containers and pods (and Kubernetes internals) are available in one place
- Lifecycle of logs is not tied to the containers themselves
- Rich queries and graphs can be made based on logging data and elements found therein

Observability++

NOTE: the way I deployed the Elastic Search storage here is bad in production! It uses RAM as its storage! So you'll quickly find yourself using hundreds of gigabytes of RAM if you leave it running for long.

But remember: this isn't the matrix, you can't stare at logs all day.



*Really, you shouldn't need to look at logs unless things are **really** bad*

Next: metrics

```
2018-01-01T00:00:00 com.blah.service.http.200 123123
2018-01-01T00:00:10 com.blah.service.http.200 235234
2018-01-01T00:00:20 com.blah.service.http.200 873287
```

Intro

Log lines are helpful to step through the state of an application over time and see a descriptive story of what events were processed and the order in which things took place.

But there are many questions that you can't naively answer through logs:

- How much memory is my process using throughout the day? Am I going to run out? *(utilization)*
- How many 500's did my webserver handle in the last hour? *(errors)*
- How long does it take to load our web page? *(latency)*
- How many backups are left in my queue *(saturation)*

What can we get from our logs?

You can get many of these things from your logs using **post-processing**.

1. Write Python script
2. Parse log files with crazy regular expressions
3. Use some statistics
4. ???
5. Profit

You can do this all directly in Kibana as well! It supports a **very rich query language**.

This can work fairly well and is probably a good place to start if you don't feel like deploying another group of complicated components.

But, this is fairly fragile due to your log format and only gets you so far.

What do we *not* get from our logs?

- Node statistics
  - CPU, load average, interrupts
  - Free memory
  - Network packets received, transmitted, dropped
  - Disk space
  - ...
- Trends of your min, max, median, quartiles, percentiles over time
- Uptimes, ie: Was my process even running at this point?
- Accurate aggregation across all sources

You could write specialised programs to read and log these values, but that's not really the point here!

Time Series Database (TSDB)

A common concept within any metrics stack is an efficient Time Series storage engine.

This is a method of efficiently storing, indexing, and querying measurements.

Generally indexed over time, into a large array of aligned time buckets.

Time	Value
12:00:00	43
12:00:05	64
12:00:10	7
12:00:15	
12:00:20	12

Can be stored and indexed very efficiently.

Compressible. Easy to garbage collect or flush.

Common class of metrics and measurements

Counters

Can only increase or be reset to 0. Eg: how many requests have been served.

Gauges

A value that can arbitrarily increase or decrease or be set to any other value. Eg: what is the temperature?

Histograms

Fixed counting buckets. Eg: how many requests took between 1 and 2 seconds to fulfill?

Summaries (complex)

Streaming estimated quantiles over sliding windows. Eg: good for arbitrary and unpredictable statistical summaries (usually you can get away with a histogram)

Common metrics ecosystems

- PUSH

  Each process opens a connection to the metrics endpoint, and sends data whenever it's available or generated (eg: on each request)

- PULL

  Processes are responsible for doing local bucket aggregations or histgram/summary calculations. A central server requests metrics from each process on a specified interval.

There is no correct or best framework, **the important thing is which one works in your architecture and whether you can get the data you need.**

- Traditional Push framework (UDP and TCP)
- Subsystems: Carbon, Whisper, Graphite, Diamond(?), Graphite-Web( )
- Arbitrary retention and aggregation (10s for 24h, 1m for 6 months)
- Light weight client
- Very scalable (Capable of global scale)
- Clients specify their metrics
- Clients can cause Denial Of Service

- Modern Pull framework (HTTP GET /metrics)
- Scrapes each 'target' every interval (10 seconds usually)
- Complex client code
- Simple protocol and storage
- Server has to know who the clients are!
- High granularity data on a fixed retention window
- Doesn't generate graphs - just data

Metrics in Kubernetes

Most metrics in Kubernetes are collected by an optional component called metrics-server (previously heapster).

- Runs as a pod
- Connects to the kubelet to retrieve pod and node cgroup information.
- Adds resource use information to pods
- Exposes /metrics endpoint to be scraped by Prometheus

This keeps responsibilities isolated.

- Allowed kubectl top node and kubectl top pod without any additional components.

Deploying metrics server

Note: this is something a cluster administrator would do!

```
$ git clone https://github.com/kubernetes-incubator/metrics-server.git

$ kubectl apply -f kubernetes-incubator/metrics-server/deploy/1.8+/
```

Done.

```
$ kubectl -n todoapp-demo top pods

NAME                     CPU(cores)   MEMORY(bytes)
mysql-55ffff8667-f9t6r   3m           260Mi
todoapp-7b9fd9b857-2kj6n   2m           3529Mi
todoapp-7b9fd9b857-5jdfz   2m           2881Mi
```

Lovely!

Option A - manual Prometheus setup

- Deploy Prometheus pod
- Persistent storage
- Service name so that other components can access it
- Manually configure it through ConfigMap and the Prometheus UI

This is a perfectly appropriate solution and would probably work fine.

However, it's not very Kube-y, as it means you need to **keep reconfiguring Prometheus as you deploy or scale your applications**. Your user and namespace for your app should probably not have write access to your Prometheus configuration.

There's a better solution using the Prometheus Operator.

Option B - the Prometheus Operator

Operator's perform work on custom resource definitions defined in Kubernetes.

The Prometheus Operator by CoreOS exposes custom resources for:

- Prometheus - Namespaced prometheus pods
- ServiceMonitor - Declaratiely specify which services your Prometheus objects target.
- PrometheusRule - Additional Prometheus Rule's regarding metrics aggregation, allow and denylists, etc..
- Alertmanager - Companion pods for sending alerts based on Prometheus queries

All of these can be defined declaratively in your manifest files alongside your App. *You don't have to reconfigure Prometheus at all!*


NOICE!
quickmeme.com

Deploying Prometheus Operator

We're going to deploy the manifests from https://github.com/coreos/prometheus-operator. The repo has many manifets in the /contrib/kube-prometheus/manifests/ directory which we'll deploy in stages in the rest of this presentation.

First, the namespace and operator:

```
$ cd /prometheus-operator/contrib/kube-prometheus/manifests

$ kubectl apply -f 00namespace-namespace.yaml
namespace/monitoring created

$ find . -type f -name '0prometheus-operator*' -exec kubectl apply -f {} \;
clusterrolebinding.rbac.authorization.k8s.io/prometheus-operator created
serviceaccount/prometheus-operator created
customresourcedefinition.apiextensions.k8s.io/servicemonitors.monitoring.coreos.com created
clusterrole.rbac.authorization.k8s.io/prometheus-operator created
servicemonitor.monitoring.coreos.com/prometheus-operator created
customresourcedefinition.apiextensions.k8s.io/prometheusrules.monitoring.coreos.com created
deployment.apps/prometheus-operator created
customresourcedefinition.apiextensions.k8s.io/alertmanagers.monitoring.coreos.com created
service/prometheus-operator created
customresourcedefinition.apiextensions.k8s.io/prometheuses.monitoring.coreos.com created
```

Lets look at our operator:

```
$ kubectl get pods -n monitoring

NAME                                   READY   STATUS    RESTARTS   AGE
prometheus-operator-c4b75f7cd-7x9sl   1/1     Running   0          1m
```

and custom resources:

```
$ kubectl get customresourcedefinition
NAME                                    CREATED AT
alertmanagers.monitoring.coreos.com     2018-10-21T20:40:13Z
prometheuses.monitoring.coreos.com      2018-10-21T20:40:13Z
prometheusrules.monitoring.coreos.com   2018-10-21T20:40:13Z
servicemonitors.monitoring.coreos.com   2018-10-21T20:40:13Z
```

```
$ kubectl get prometheus
No resources found.
```

Remember that this is not Prometheus itself, this is a Kubernetes component that will deploy and configure Prometheus for us in response to custom resource definitions.

Using the operator

The Prometheus Operator comes with some useful configuration for a Prometheus deployment that gathers all of the metrics for Kubernetes. We're going to slowly deploy some parts of it in the next few slides.

First the Prometheus server, its permissions, and it's rules for monitoring the Kube components, pods, and nodes.

```
$ find . -type f -name 'prometheus-*' -exec kubectl apply -f {} \;

servicemonitor.monitoring.coreos.com/prometheus created
rolebinding.rbac.authorization.k8s.io/prometheus-k8s-config created
rolebinding.rbac.authorization.k8s.io/prometheus-k8s created
rolebinding.rbac.authorization.k8s.io/prometheus-k8s created
rolebinding.rbac.authorization.k8s.io/prometheus-k8s created
servicemonitor.monitoring.coreos.com/kube-controller-manager created
servicemonitor.monitoring.coreos.com/kube-scheduler created
clusterrole.rbac.authorization.k8s.io/prometheus-k8s created
role.rbac.authorization.k8s.io/prometheus-k8s created
role.rbac.authorization.k8s.io/prometheus-k8s created
role.rbac.authorization.k8s.io/prometheus-k8s created
servicemonitor.monitoring.coreos.com/kubelet created
serviceaccount/prometheus-k8s created
prometheus.monitoring.coreos.com/k8s created
role.rbac.authorization.k8s.io/prometheus-k8s-config created
prometheusrule.monitoring.coreos.com/prometheus-k8s-rules created
servicemonitor.monitoring.coreos.com/coredns created
clusterrolebinding.rbac.authorization.k8s.io/prometheus-k8s created
servicemonitor.monitoring.coreos.com/kube-apiserver created
```

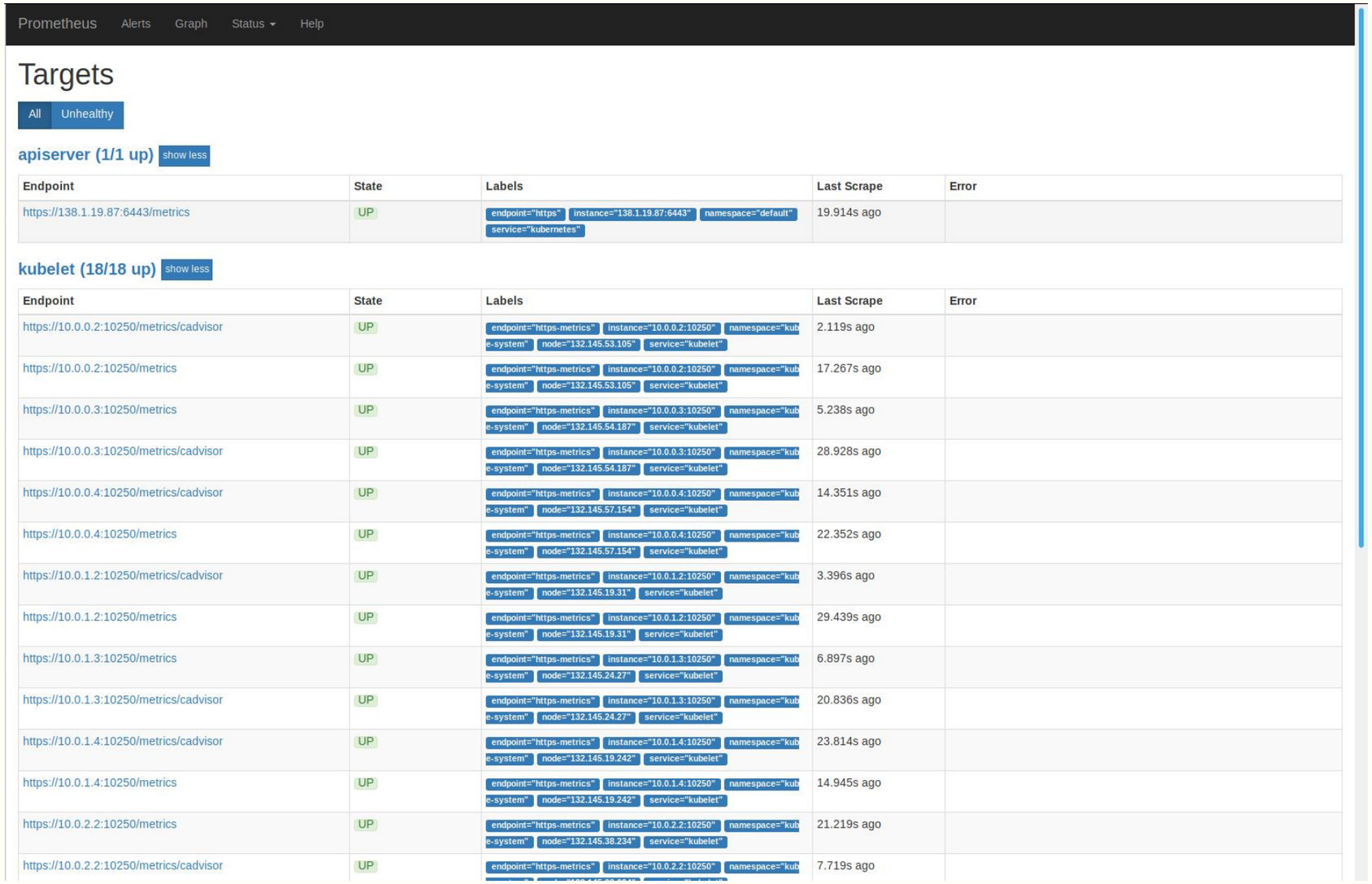
Lets wait for the pod to be ready..

```
$ kubectl -n monitoring get pods
NAME                                   READY   STATUS    RESTARTS   AGE
prometheus-k8s-0                       3/3     Running   1          1m
prometheus-k8s-1                       3/3     Running   1          1m
prometheus-operator-c4b75f7cd-7x9sl    1/1     Running   0          17m
```

And now we can use our port forward trick to look at the UI:

```
$ kubectl -n monitoring port-forward svc/prometheus-k8s 9090
```

```
http://localhost:9090/targets
```



Lots of lovely scrape targets...

Using Grafana for some graphs

Before we go further, lets deploy a Grafana UI in front of this so that we can demonstrate some graphs.

Again, the Prometheus Operator provides a useful setup:

```
$ find . -type f -name 'grafana-*' -exec kubectl apply -f {} \;
service/grafana created
secret/grafana-datasources created
serviceaccount/grafana created
configmap/grafana-dashboards created
deployment.apps/grafana created
configmap/grafana-dashboard-k8s-cluster-rsrc-use created
configmap/grafana-dashboard-k8s-node-rsrc-use created
configmap/grafana-dashboard-k8s-resources-cluster created
configmap/grafana-dashboard-k8s-resources-namespace created
configmap/grafana-dashboard-k8s-resources-pod created
configmap/grafana-dashboard-nodes created
configmap/grafana-dashboard-pods created
configmap/grafana-dashboard-statefulset created
```

See if you can work out whats going on based on the list of objects above..

Connecting to and viewing Grafana

```
$ kubectl -n monitoring get pods
NAME                          READY   STATUS    RESTARTS   AGE
grafana-566fcc7956-smr54      1/1     Running   0          1m
```

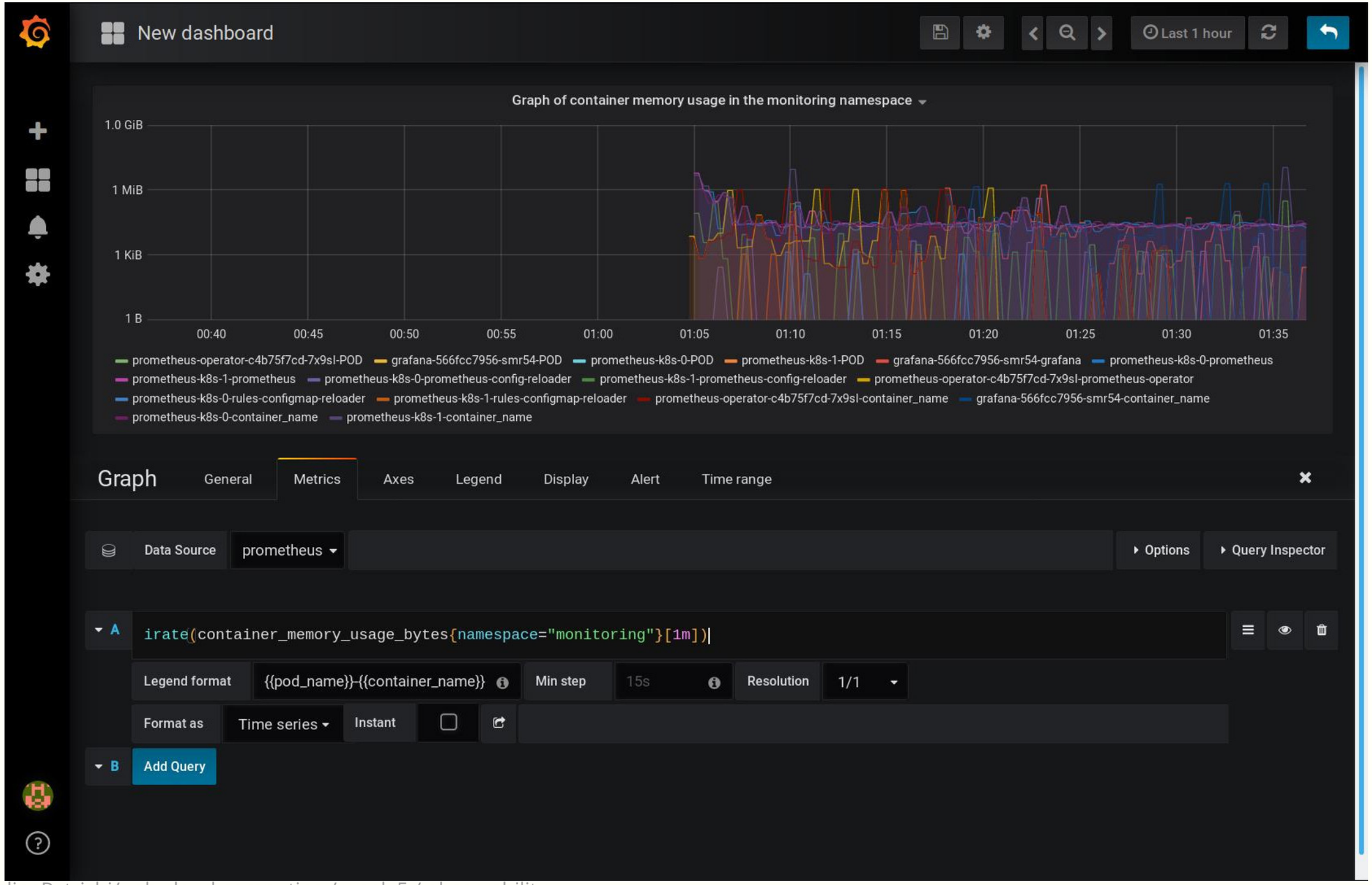
Again, lets use port forward:

```
$ kubectl -n monitoring port-forward svc/grafana 3000
```

```
http://localhost:3000
```

Login with admin / admin.

NOTE: if this is your first time using Grafana, take some time to explore it. It's a very complex interface with complex configuration and queries. It's very powerful and being able to use it is a useful skill.


New dashboard
Last 1 hour
Graph of container memory usage in the monitoring namespace
1.0 GiB
1 MiB
1 KiB
1 B
00:40
00:45
00:50
00:55
01:00
01:05
01:10
01:15
01:20
01:25
01:30
01:35
prometheus-operator-c4b75f7cd-7x9sl-POD
grafana-566fcc7956-smr54-POD
prometheus-k8s-0-POD
prometheus-k8s-1-POD
grafana-566fcc7956-smr54-grafana
prometheus-k8s-0-prometheus
prometheus-k8s-1-prometheus
prometheus-k8s-0-prometheus-config-reloader
prometheus-k8s-1-prometheus-config-reloader
prometheus-operator-c4b75f7cd-7x9sl-prometheus-operator
prometheus-k8s-0-rules-configmap-reloader
prometheus-k8s-1-rules-configmap-reloader
prometheus-operator-c4b75f7cd-7x9sl-container_name
grafana-566fcc7956-smr54-container_name
prometheus-k8s-0-container_name
prometheus-k8s-1-container_name
Graph
General
Metrics
Axes
Legend
Display
Alert
Time range
Data Source
prometheus
Options
Query Inspector
A
irate(container_memory_usage_bytes{namespace="monitoring"}[1m])
Legend format
{{pod_name}}-{{container_name}}
Min step
15s
Resolution
1/1
Format as
Time series
Instant
B
Add Query

rate(container_memory_usage_bytes{namespace="monitoring"}[1m])

container_memory_usage_bytes - bytes of memory in use by the container

{namespace="monitoring"} - filter by the namespace

[1m] - each data point is calculated from a sliding window of 1 minute (range vector)

rate() - per second average rate (convert range vector to instant vector)

Prometheus Node exporter

The node_exporter is a Prometheus (not Kube) component that gathers metrics from the host on which it runs and exposes them as a scrape target for Prometheus.

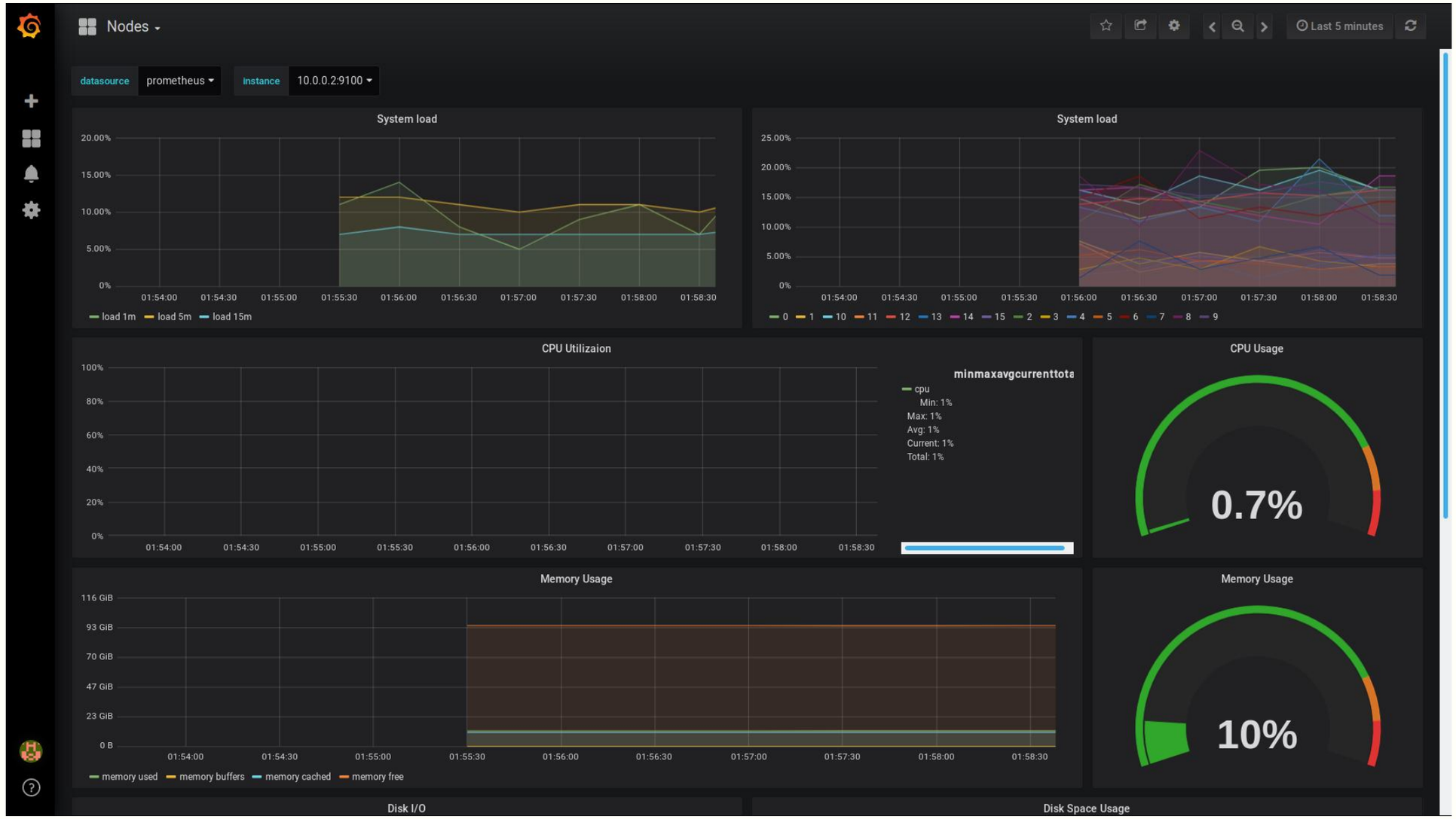
We can expect things like CPU, memory, network, disk usage, etc..

This should help to complete our metrics picture.

```
$ find . -type f -name 'node-exporter-*' -exec kubectl apply -f {} \;

servicemonitor.monitoring.coreos.com/node-exporter created
service/node-exporter created
clusterrole.rbac.authorization.k8s.io/node-exporter created
daemonset.apps/node-exporter created
serviceaccount/node-exporter created
clusterrolebinding.rbac.authorization.k8s.io/node-exporter created
```

```
node-exporter-2nfw5              2/2     Running   0          2m    10.0.0.2     132.145.53.105
node-exporter-644tz              2/2     Running   0          2m    10.0.2.4     132.145.32.222
node-exporter-6sd9s              2/2     Running   0          2m    10.0.0.3     132.145.54.187
node-exporter-75g7w              2/2     Running   0          2m    10.0.1.3     132.145.24.27
node-exporter-mt85k              2/2     Running   0          2m    10.0.1.2     132.145.19.31
```



Now we have our node stats in the preprepared dashboards that were deployed for us.

Application metrics

There are many Prometheus client libraries that help you turn your app into a scrape target itself.

This allows you to expose metrics from the application level rather than the platform:

- HTTP request latencies and codes
- Function call times and rates
- Database call frequencies and latencies
- Almost anything you want!

Traditionally exposed over a /metrics path on an HTTP server.

So what do we DO with these metrics and logs now that we have them?

Things to do with metrics

- Deploy Prometheus Operator objects to **monitor your own applications**
- Identify **bottlenecks** in your application (but don't optimise too early!)
- Monitor the **utilisation** of your Kubernetes resources
- Identify hardware **failures** before they happen
- Provide support for your customers by **diagnosing** problems in production
- Scale your app in **response** to its resource utilization and saturation
- **Autoscale your app** in response to its resource usage

Autoscaling

Why scale?

- Network cards can only handle a certain number of packets per second
- Applications can only handle a certain number of concurrent connections
- Requests may contain "blocking" processing like database calls, requests to other services, file uploads, etc.
- SSL is vital! But has serious CPU costs on each connection handshake.
- Malicious users can cause denial of service to other users

Most systems fall somewhere in between the following scenarios:

Worst case

You own physical space in a datacenter and have to physically plug in machines and boot them by pressing the power button. Scaling is slow and manual.

Best case

Your app runs entirely in FaaS (function as a service), CDNs, edge computing, and other zero-scale platforms. Scaling is easy and you don't even need to think about it.

What is autoscaling?

- Adding more physical VM's, pods, deployments, copies of your app, load balancers, etc in response to some signal. (Horizantal scaling)
- What kind of signals:
  - Customer traffic
  - Anticipation of customer traffic
  - Time of day/week/year
  - CPU usage
  - Optimisations or changes in your business logic
  - *A person clicking a button*

The important thing here is that your management code automatically provisions and connects new infrastructure and installs your application.

Sources of signals

- ElasticSearch!
- Grafana!
- Monitoring built into your platform or IaaS

Autoscaling in Kubernetes

You may have already seen Deployments and StatefulSets in Kubernetes.

These allow you to deploy replicas of your pods with the same image and configuration.

Kubernetes also supports a Horizantal Pod Autoscaler (HPA).

> *The Horizontal Pod Autoscaler is implemented as a Kubernetes API* ***resource and a controller****. The resource determines the behavior of the controller. The controller* ***periodically adjusts the number of replicas*** *in a replication controller or deployment to* ***match the observed average CPU utilization*** *to the target specified by user.*

Basically:

desiredReplicas = ceil(replicas * (currentMetricValue / desiredMetricValue))

= ceil(2 * (200m / 100m))

= 4

Example time!

- Simple Python 3 Server doing Scrypt on each POST request

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scrypt-script
data:
  server.py: |
    import hashlib, http.server, socketserver, base64, os
    class ScryptHandler(http.server.BaseHTTPRequestHandler):
        cost = int(os.environ.get('COST', '14'))
        def do_POST(self):
            self.send_response(200)
            self.end_headers()
            self.wfile.write(
                base64.b64encode(
                    hashlib.scrypt(b"blah", salt=b"salt", n=(1 << self.cost), r=5, p=10)
                )
            )

    try:
        server = http.server.HTTPServer(('', 8000), ScryptHandler)
        server.serve_forever()
    except KeyboardInterrupt:
        server.socket.close()
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scrypt-server
spec:
  replicas: 1                                # <- replicas
  selector:
    matchLabels:
      app: scrypt-server
  template:
    metadata:
      labels:
        app: scrypt-server
    spec:
      containers:
      - name: python
        image: python:3.6
        args: ["python", "/app/server.py"]         # <- run the script
        resources:
          limits:
            cpu: 200m                          # <- resource constraint
        ports:
        - containerPort: 8000
        volumeMounts:
        - name: script-vol
          mountPath: /app                          # <- mount the script
      volumes:
      - name: script-vol
        configMap:
          name: scrypt-script
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: scrypt-svc
spec:
  selector:
    app: scrypt-server
  ports:
  - protocol: TCP
    port: 8000
    targetPort: 8000
```

A service to expose this pod.

An http ingress based on our loadbalanced Nginx ingress:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: scrypt-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: scrypt.uob.example.local
    http:
      paths:
      - path: /
        backend:
          serviceName: scrypt-svc
          servicePort: 8000
```

And some fake local DNS to access it:

```
$ cat /etc/hosts
...
132.145.15.135 scrypt.uob.example.local
```

Lets see what it does without autoscaling

In one terminal lets watch the pod metrics:

```
$ while true; do kubectl top pod -l app=scrypt-server && sleep 2; done
NAME                             CPU(cores)   MEMORY(bytes)
scrypt-server-5cd5f48d5b-sjjhr   1m           12Mi
...
```

And in another, lets make requests in a loop:

```
$ while true; do time curl -X POST http://scrypt.uob.example.local; done
XNLScy1SfT5zqH+I6kJXlAmcX0IkE9Z8GbyJJQZ9oZv4EejJUyHOSjP4BjQNrGM5V+czfulMTxQqDctL5xWkpQ==
real    0m2.724s
user    0m0.016s
sys     0m0.011s
...
```

We should see our pod reach its CPU limit (this can take 30s - 1m):

```
NAME                             CPU(cores)   MEMORY(bytes)
scrypt-server-7cf4d7f9b5-kml5b   194m         23Mi
```

Now with an HPA

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: scrypt-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: scrypt-server
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
```

> *"Increase the replica count up to 10 replicas for the `scrypt-server" deployment while the cpu utilization is above 50%"*

Rerun the load test

After a minute or two we should see the HPA begin reacting:

```
NAME                          CPU(cores)   MEMORY(bytes)
scrypt-server-7cf4d7f9b5-kml5b   15m          23Mi
scrypt-server-7cf4d7f9b5-xpzgm   115m         22Mi
```

A new replica!

```
NAME        REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
scrypt-hpa   Deployment/scrypt-server   32%/50%   1         10        2          3m
```

NOTE: annoyingly the HPA runs on a polling loop *and* only takes some time to ensure that the target utilization has been exceeded. This means that the metrics and replica counts can take a few minutes to reconcile. (This is usually fine!)

Other notes

- Why is the autoscaling somewhat delayed?

  To prevent *Thrashing* the autoscaler has a purposeful delay in scaling up or down. We don't want to scale our app too fast in response to what may only be a very small spike in resource utilization.

- Scaling based on other resources?

  You can also scale on memory usage and even your own 3rd party metrics if they support the "metrics API"

- Could you implement this as a ELK/Grafana monitoring hook with a Kube API call?

  Yes you totally could! But it would lack the declarative configuration that Kubernetes manifests give you.

Fin!

Key takeaway points

- Observability is vital in Cloud Native applications!
- Metrics and logs are both important when running a production application
- This is a huge space with many options and approaches and no real "best" approach
- **Important thing is: can you tell how your app is performing and what it is doing?**

How does this apply to your project work?

- If you're writing long running applications, especially those that are web or Kubernetes based, think about using metrics and proper logging to make your system more observable.
- Think about deploying a proper ELK or Prometheus stack to aggregate all of your observability items. Allow this to help drive your development in the long term.

Thanks!