{mthree}

Prometheus & Grafana

Site Reliability Engineering

Overview

In this module, we will look at a monitoring system called Prometheus, the dashboard called Grafana, and how they interact.

Learning Objectives

- Explain what Prometheus is and what it does
- Examine how Prometheus gets data
- Configure Prometheus
- Query Prometheus with PromQL
- Use Grafana to create visualisations

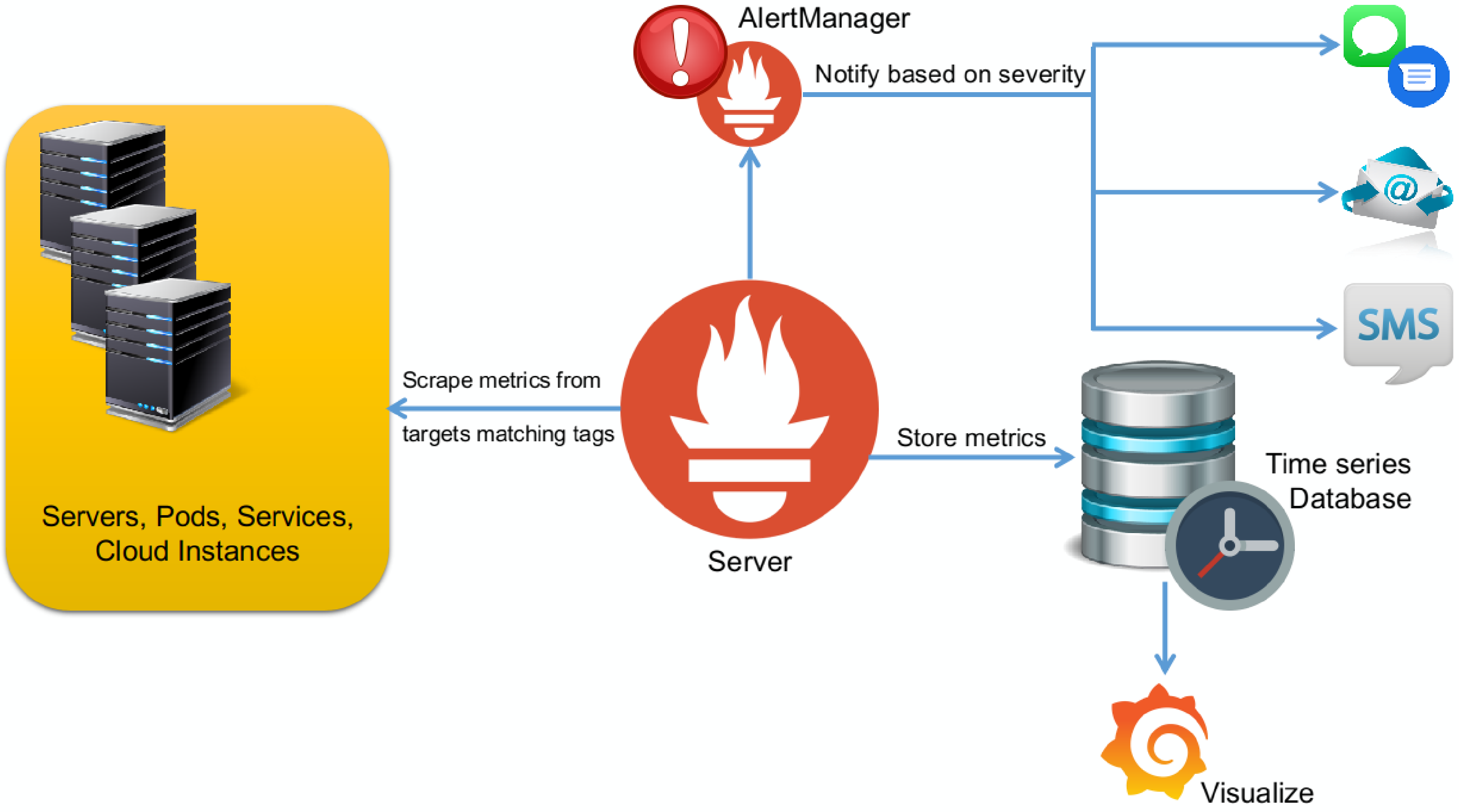
{mthree}

What is Prometheus?

- Monitoring tool
- From SoundCloud
  - Created to monitor Highly dynamic environments
    - Very useful for monitoring Kubernetes, Docker swarm
  - Can be used in traditional non-container environments
- Has become the go to in Container and Microservice world
  - Modern DevOps complex needs more and more automation
  - Challenging and complex infrastructures need to be managed

{mthree}

How Prometheus Works



How Does Prometheus Work?

- Prometheus server
  - Time series database
    - Stores metrics data current CPU usage
  - Data Retrieval worker
    - Responsible for pulling metrics
  - Web server
    - Accepts queries and display data in UI or dashboard or visualization such as Grafana
- Monitors targets
  - Linux server, Windows server, web servers, application servers, databases
- Measures units such as
  - CPU status, request counts and durations, exception counts, memory/disk usage – Metrics

{mthree}

Monitoring Applications

- Collects relevant metrics such as:
  - How many requests
  - How many exceptions
  - How many server resources
- Client libraries expose endpoints via HTTP URL called /metrics
  - Requires using the library within your code
  - https://prometheus.io/docs/instrumenting/clientlibs/
- Metrics can be
  - Counter
  - Guage
  - Historgram

{mthree}

Exposing Metrics Steps

- Exporters
  - These are agents that extract data
  - Run on the system to monitor
  - Prometheus configured to point at the exporter agent port
- Many 3rd party exporters
  - https://prometheus.io/docs/instrumenting/exporters/
  - Pre-built - https://prometheus.io/download/
- Download and run exporter
  - For Linux, server download a node exporter, untar and execute
  - Converts metrics of the server
  - Exposes endpoints
  - Configure Prometheus to scrape this endpoint
- Windows is also supported
- There may already be a Docker container
  - Search the web for the app name and exporter
  - Or check through Prometheus exporters and see if they mention Docker containers

What do metrics look like?
- https://jenkins.computerlab.online/prometheus/

{mthree}

Configuring Metrics

- Exporters expose attributes
- Customize the exporters to define attributes to gather
  - >>> Generally passed as CLI options
  - >>> May have a configuration file
- Example MySQL metrics
  - >>> https://github.com/prometheus/mysqld_exporter

```
./mysqld_exporter -collect.auto_increment.columns \
-collect.info_schema.innodb_metrics \
-collect.perf_schema.tablelocks
```

{mthree}

Configuring Metrics for Java Spring Boot: Maven

- Maven pom.xml requires the following dependencies

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-prometheus</artifactId>
        <scope>runtime</scope>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

{mthree}

Configuring Metrics for Java Spring Boot: application.properties

- application.properties requires

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.prometheus=/metrics
management.endpoints.web.path-mapping.health=status
management.endpoints.web.exposure.include=health,info,prometheus
management.metrics.distribution.percentiles-
histogram.http.server.requests=true
```

{mthree}

Configuration

- Prometheus.yaml – specify targets
  - >>> How often
  - >>> Rules for aggregation
  - >>> Creating alerts
  - >>> What resources to monitor
  - >>> Define other endpoints
- Example shows static configuration

```
global:
  scrape_interval:     15s  evaluation_interval: 15s
…….
scrape_configs:
  - job_name: 'prometheus-metrics'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'database'
    static_configs:
      - targets: ['mydb.courselab.online:9090']
```

{mthree}

P A G E

Our Lab Configuration (1)

- Automated/Dynamic
- Prometheus scrapes Kubernetes pods and services
  - >>> Done through Custom Resource Definitions (CRD)
- Uses Prometheus Operator
  - >>> https://prometheus-operator.dev/
- Pods and Services expose scraper metrics ports
- New containers automatically detected
- PodMonitor or ServiceMonitor
  - >>> Depending on whether you are scaling

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: helm-controller
  namespace: flux-system
spec:
  selector:
    matchLabels:
      app: helm-controller
  podMetricsEndpoints:
    - port: http-prom
```

{mthree}

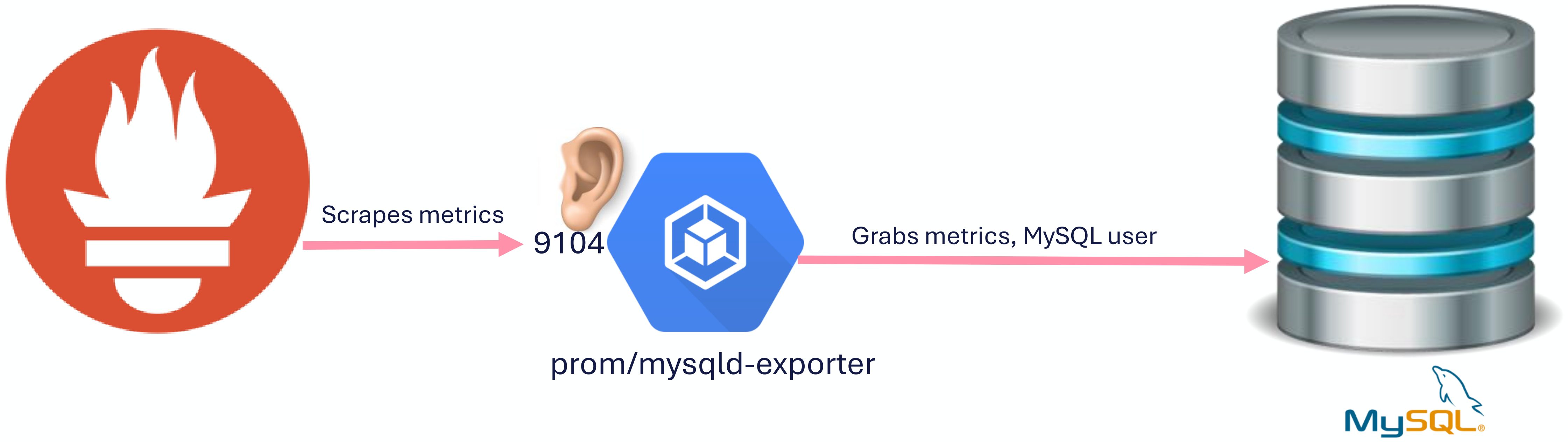
PAGE

Our Lab Configuration (2)

- Managed via Kubernetes manifests
  - Within your sre-course-infra environment/namespace directory
- Requires
  - podMonitor.yaml or serviceMonitor.yaml or probe.yaml (if exists)
    - To tell Kubernetes about your monitoring ports to scrape
  - Containers
    - Either a separate monitoring container, e.g. mysqld-exporter
    - Or a port in your application container
      - That is not the application
      - Exposes Prometheus metrics

Example podMonitor

- Monitoring MySQL server remotely
- Reduce load on the Database server
- Dedicated pod to monitor



{mthree}

The Deployment Of Exporter

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/component: mysqld-exporter
    app.kubernetes.io/name: orderbook
  name: mysqld-exporter
  namespace: sre-example-dev
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/component: mysqld-exporter
      app.kubernetes.io/name: orderbook
  template:
    metadata:
      labels:
        app.kubernetes.io/component: mysqld-exporter
        app.kubernetes.io/name: orderbook
```

```yaml
spec:
      containers:
        - env:
            - name: DATA_SOURCE_NAME
              value:
"root:secret123@(orderbookdb:3306)/"
          image: prom/mysqld-exporter
          name: mysqld-exporter
          imagePullPolicy: Always
          ports:
            - name: http-prom
              containerPort: 9104
      restartPolicy: Always
```

{mthree}

The Prometheus Capture

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: mysqld-exporter
  namespace: sre-example-dev
spec:
  selector:
    matchLabels:
      app.kubernetes.io/component: mysqld-exporter
      app.kubernetes.io/name: orderbook
  podMetricsEndpoints:
    - port: http-prom
```

The port name for your pod port exposing /metrics

{mthree}

Monitoring the application using blackbox exporter

```
apiVersion: monitoring.coreos.com/v1
kind: Probe
metadata:
  name: orderbook
  namespace: sre-example-dev
spec:
  jobName: blackbox
  interval: 60s
  scrapeTimeout: 30s
  module: http_2xx
  prober:
    url: prometheus-blackbox-exporter.prometheus:9115
    scheme: http
    path: /probe
  targets:
    staticConfig:
      static:
        - http://orderbookdev.computerlab.online
```

Scraper URL

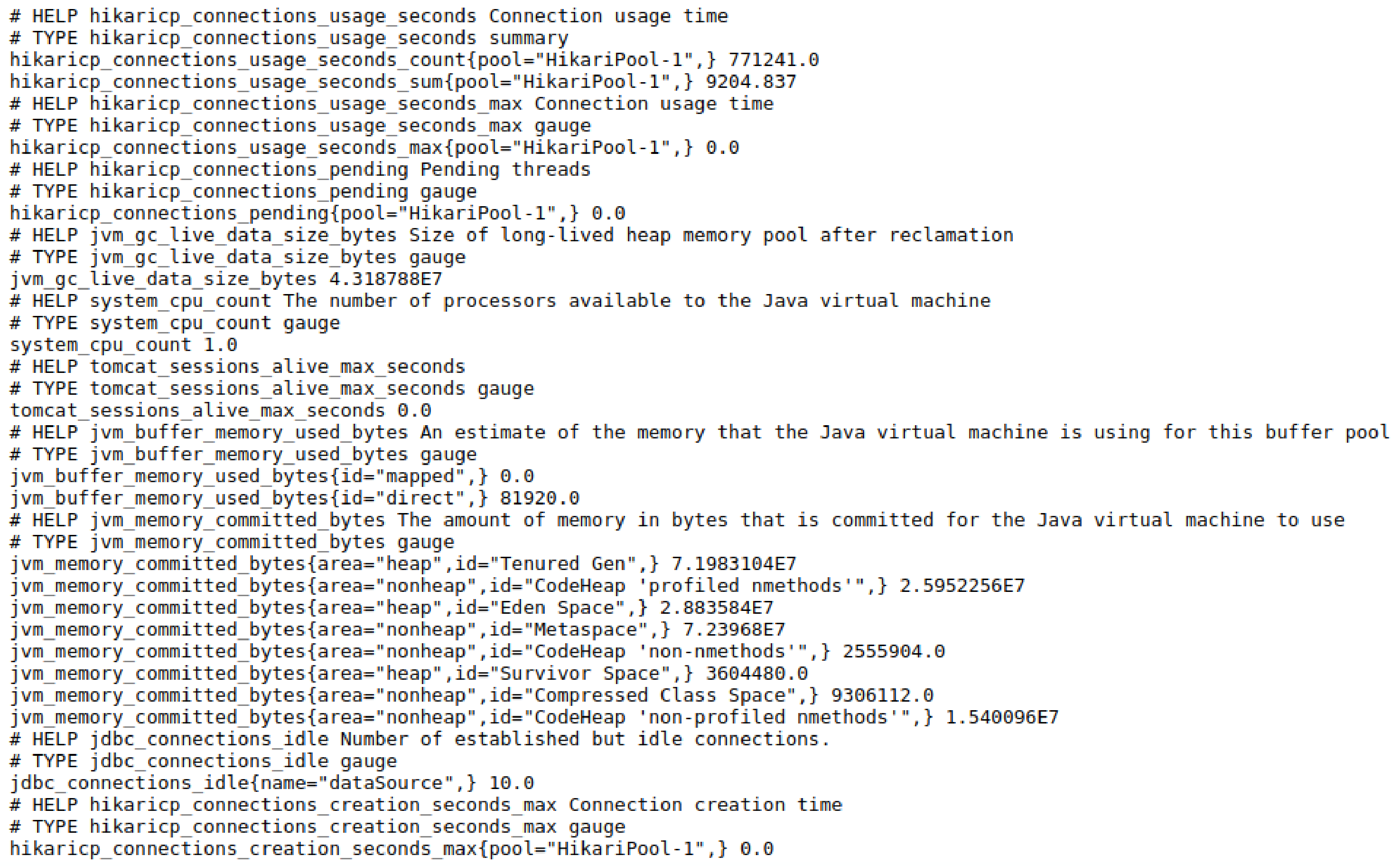
URL to monitor

{mthree}

https://github.com/prometheus/blackbox_exporter

Orderbook has /metrics

https://cXXteam??dev.computerlab.online/metrics



```
# HELP hikaricp_connections_usage_seconds Connection usage time
# TYPE hikaricp_connections_usage_seconds summary
hikaricp_connections_usage_seconds_count{pool="HikariPool-1",} 771241.0
hikaricp_connections_usage_seconds_sum{pool="HikariPool-1",} 9204.837
# HELP hikaricp_connections_usage_seconds_max Connection usage time
# TYPE hikaricp_connections_usage_seconds_max gauge
hikaricp_connections_usage_seconds_max{pool="HikariPool-1",} 0.0
# HELP hikaricp_connections_pending Pending threads
# TYPE hikaricp_connections_pending gauge
hikaricp_connections_pending{pool="HikariPool-1",} 0.0
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes 4.318788E7
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count 1.0
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
tomcat_sessions_alive_max_seconds 0.0
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that the Java virtual machine is using for this buffer pool
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{id="mapped",} 0.0
jvm_buffer_memory_used_bytes{id="direct",} 81920.0
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="heap",id="Tenured Gen",} 7.1983104E7
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'",} 2.5952256E7
jvm_memory_committed_bytes{area="heap",id="Eden Space",} 2.883584E7
jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 7.23968E7
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",} 2555904.0
jvm_memory_committed_bytes{area="heap",id="Survivor Space",} 3604480.0
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",} 9306112.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'",} 1.540096E7
# HELP jdbc_connections_idle Number of established but idle connections.
# TYPE jdbc_connections_idle gauge
jdbc_connections_idle{name="dataSource",} 10.0
# HELP hikaricp_connections_creation_seconds_max Connection creation time
# TYPE hikaricp_connections_creation_seconds_max gauge
hikaricp_connections_creation_seconds_max{pool="HikariPool-1",} 0.0
```

Grafana

↘ Grafana allows you to:
  >>> Query
  >>> Visualize
  >>> Alert on
  >>> Understand metrics

↘ Central visualization of all your Prometheus monitoring
  >>> Graphs, meters and more

↘ Dashboards to provide ease of understanding and speed
  >>> Useful and colourful representation
  >>> Thresholds and colours to determine issues or perfect conditions

Activity: Viewing Prometheus

Let's get a feel for Prometheus and PromQL

- Go to https://prometheus.computerlab.online
  - Our Prometheus is authenticated through OAuth and your GitHub login
- Find the following:
  - Jenkins disk usage
    - Start by typing in disk into the search box
    - Select the default_jenkins_disk_usage_bytes
    - Click the Execute button
    - You will now see various lines relating to bytes for various directories
  - Typing any letter will bring up potential metrics that have been scraped
    - e.g., probe_http
  - Find the following:
    - Kubernetes build info?
    - What plugins do we have with Grafana?
    - What namespaces exist in Kubernetes?

{mthree}

Activity: Viewing Prometheus

- View the pod container info for the orderbook container

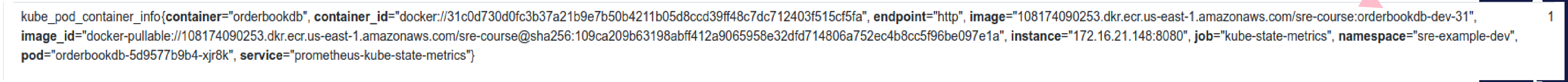
```
kube_pod_container_info{container=~'orderbook.*'}
```

- Now look for your container orderbook in your namespace

Exact match

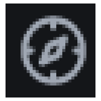
```
kube_pod_container_info{container=~'orderbook.*',namespace='sre-example-dev'}
```

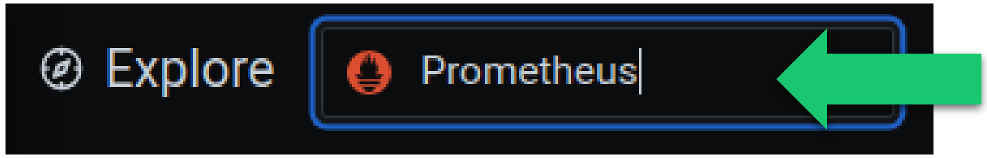
RegEx match



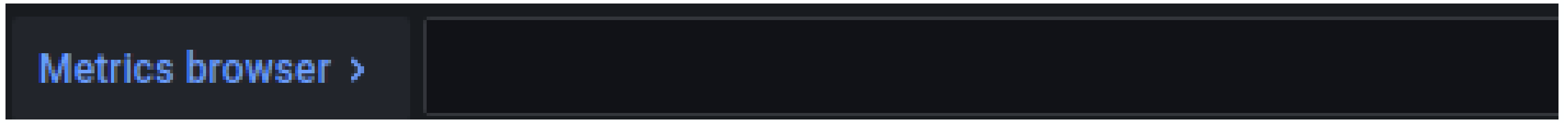
```
kube_pod_container_info{container="orderbookdb", container_id="docker://31c0d730d0fc3b37a21b9e7b50b4211b05d8ccd39ff48c7dc712403f515cf5fa", endpoint="http", image="108174090253.dkr.ecr.us-east-1.amazonaws.com/sre-course:orderbookdb-dev-31",   1
image_id="docker-pullable://108174090253.dkr.ecr.us-east-1.amazonaws.com/sre-course@sha256:109ca209b63198abff412a9065958e32dfd714806a752ec4b8cc5f96be097e1a", instance="172.16.21.148:8080", job="kube-state-metrics", namespace="sre-example-dev",
pod="orderbookdb-5d9577b9b4-xjr8k", service="prometheus-kube-state-metrics"}
```

Activity: PromQL in Grafana

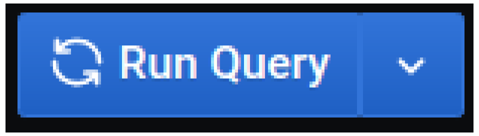
- Head over to https://grafana.computerlab.online
- Click the Explore icon 
- Select Prometheus in the pull down, top left of screen



- In the metrics text field type your PromQL



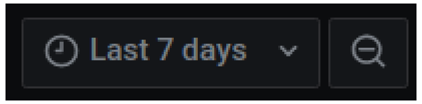
- Enter kube_namespace_created; note IntelliSense as you type

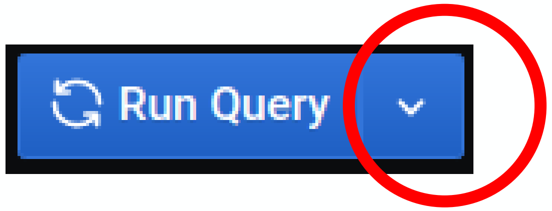


{mthree}

Activity: PromQL in Grafana

- No results?
- Check the date range top right, left of Run Query
  - >>> Change to Last 7 days
- Have Grafana update
  - >>> Click the down arrow on the Run Query button
  - >>> Select a refresh interval
- Now query the Jenkins disk usage
- Change Range under Metrics to Instant
  - >>> What do you notice?
  - >>> Change back to Range
- Change the Step to 2d for every 2 days





{mthree}

Activity: Grafana Ui

- Take a look at the following;
  - >>> Click the Inspector - what does it tell you?
  - >>> Click the Query History - re-run the kube_namespace_created from here

Activity: Dashboards

- Dashboards provide your teams useful view of the site
- Creating through Grafana:
  - Select the Dashboards icon
  - Select Manage from the pop-up menu
  - From here you can create a new dashboard
    - Create one with your full name for practice
    - We can delete it later

{mthree}

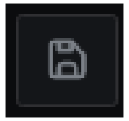
P
A
G
E

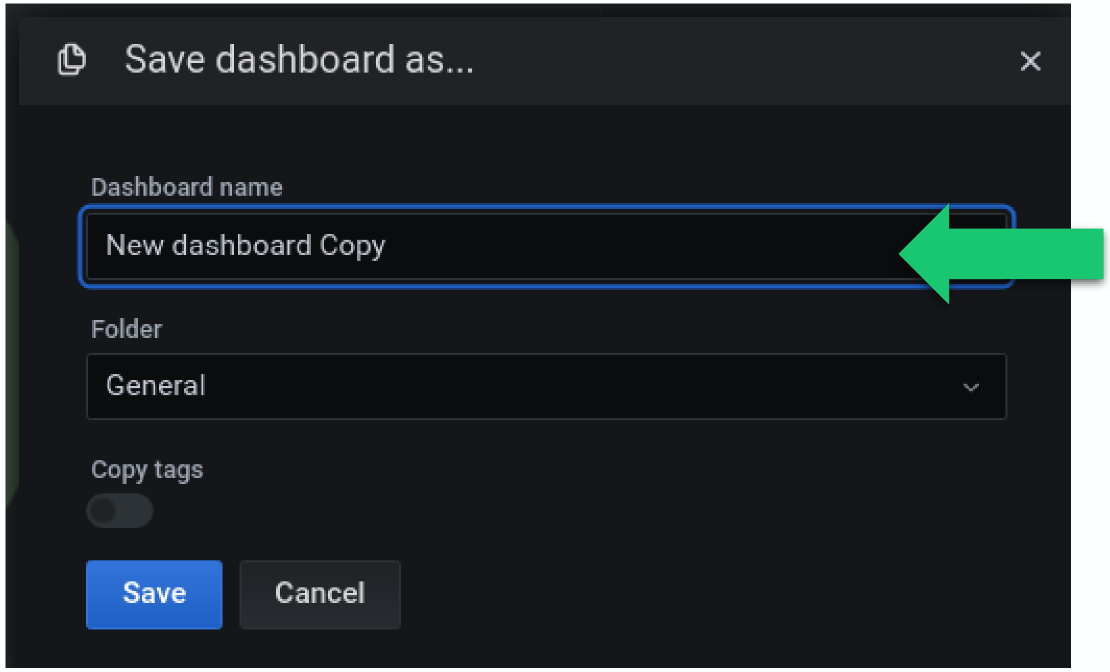
Activity: Dashboards

- Click the New Dashboard button
- You will be presented with a blank dashboard
- Click the Add new panel button to start adding charts
  - Or use the icon
- To the right of the screen are the settings for the panel
  - Title your panel Jenkins Disk Space
  - Expand the Visualization section
  - Select a Pie Chart
  - Under options
    - Remove the Legend
- Add your metric to the Prometheus query in the lower main screen
- Select Instant
- Select Last hour as the time range
- Click Apply button to see your panel on the dashboard

{mthree}

Activity: Save the Dashboard

- Click the floppy disk icon to save 
- Fill in the pop up box by naming your dashboard
- If folders exist, you may wish to save it in a project folder


Save dashboard as...

Dashboard name: New dashboard Copy

Folder: General

Copy tags

{mthree}

Activity: Folder

- If you wish to separate your work
- Click the + icon
- Select Folder
- Type in the name of your folder
- You can move existing Dashboards to the folder
  - >>> Go to Dashboards Home
  - >>> Select your Dashboard
  - >>> Click the settings icon
  - >>> Select your folder from the pull down menu
  - >>> Save dashboard

Activity: Viewing Logs

- Our Grafana also trawls logs using Loki
  - >>> https://grafana.com/oss/loki/
  - >>> Alternatives include ELK, Splunk, Datadog
- Go to Explore
- Set the Explore data from Prometheus to Loki
- Where Metrics Browser used to be you'll now see Log browser
  - >>> You can click through to find a log
  - >>> You can start to type a log query

Activity: Show deployment logs

- Click Log labels
- Select namespace
- Select your Dev namespace
  - If your namespace does not show in the list increase the time range
  - Grafana only shows data in valid time range
- You will now see log lines from the files of all your running containers

{mthree}

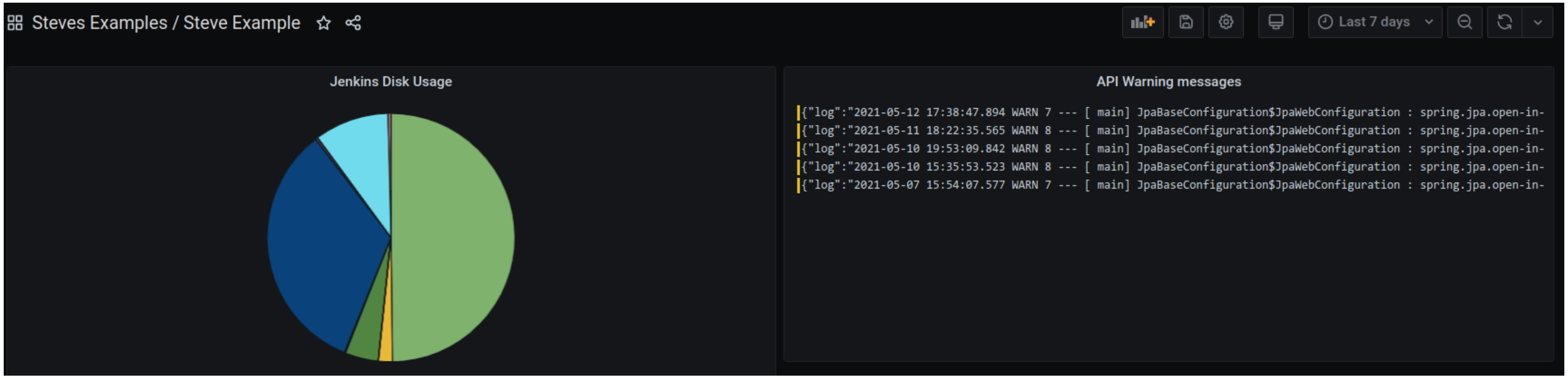
The Loki Query Syntax

- Similar to Prometheus
- Key = value statement contained in { }
- Exact match of value use =
- RegEx match of value use =~
  - >>> Note if you wish to ”contain” then use .* either side of your text
    - e.g. {namespace=~”.*example.*”}
- Multiple key searches are comma separated
  - >>> {namespace=”sre-example-dev”, app=”orderbookapi”}
- Values must be in double quotes
- Looking for the word error in the log line
  - >>> {namespace=”sre-example-dev”, app=”orderbookapi”} |= ”error”

{mthree}

Activity: Create A Log Entry

- Add to your dashboard a log entry for warnings in your orderbookapi
- Note you can drag widget positions around using the title bar
- The title bar also has a pull down menu next to the panel name



```
{"log":"2021-05-12 17:38:47.894 WARN 7 --- [ main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-
{"log":"2021-05-11 18:22:35.565 WARN 8 --- [ main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-
{"log":"2021-05-10 19:53:09.842 WARN 8 --- [ main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-
{"log":"2021-05-10 15:35:53.523 WARN 8 --- [ main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-
{"log":"2021-05-07 15:54:07.577 WARN 7 --- [ main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-
```

{mthree}

Activity: Coding the Dashboard 1

- Click the New Dashboard button
- Click the Add new panel button
  - Set the panel properties:
    - Panel title: Application Memory Usage
    - Visualization: Gauge
    - Fields:
      - Min: 0
      - Max: 100
  - Expand Thresholds and click Add threshold
    - 80 red
    - 70 yellow
- Set the query of Prometheus
  - sum(container_memory_usage_bytes{namespace="<yournamespace>", container="orderbookapi"}) / sum (container_memory_usage_bytes{namespace="<yournamespace>"}) * 100
  - Click the Apply button

{mthree}

Activity: Coding the Dashboard 2

- Click the Add new panel button
  - Set the panel properties:
    - Panel title: Application CPU Usage
    - Visualization: Gauge
    - Fields:
      - Min: 0
      - Max: 100
  - Expand Thresholds and click Add threshold
    - 80 red
    - 70 yellow
- Set the query of Prometheus
  - sum (container_cpu_usage_seconds_total {namespace="<yournamespace>", container="orderbookapi"}) / sum(container_cpu_usage_seconds_total{namespace="<yournamespace>"}) * 100
  - Click the Apply button

{mthree}

Activity: Export the Dashboard

- Click the ⚙ button
- Click JSON Model
- Click into the edit box and copy all of the JSON
- Save this to a file or other safe place for later use

Activity: Add the Dashboard Definition

- Switch to your local sre-course-infra repository
  - git checkout main
  - git pull
  - git checkout <your branch>
  - git merge main
- Copy an existing dashboard definition file to your project folder
  - cd <project folder>
    - flux/apps/eks-sre-course/cXXX-team??-dev
  - Edit the file status.yaml
- Edit stats.yaml and change:
  - name: to cXXXteam??dev
  - data: to cXXXteam??dev.json: |

```
{
  <paste in the saved json definition>
}
```

  - At the bottom of the JSON code change
    - Change the title value to “cXXXteam??dev Dashboard”
    - Change uid value to “cXXXteam??dev”
- Save the new definition

{mthree}

Activity: Deploy the Dashboard Definition

- Push the changes to the shared repo
  - >>> git add --all (two dashes)
  - >>> git commit –m "Adding orderbook stats dashboard"
  - >>> git commit
  - >>> git push
- Log in to the shared repository
  - >>> Change to your branch
  - >>> Create a pull request
  - >>> Have the pull request approved
- The dashboard definition will be deployed

{mthree}

References & Further learning

- How Prometheus monitoring works
  >>> https://www.youtube.com/watch?v=h4Sl21AKiDg
- How To Setup A Grafana Dashboard Step By Step
  >>> https://www.youtube.com/watch?v=4qpI4T6_bUw
- How to create Grafana Dashboards: The Easy way
  >>> https://www.youtube.com/watch?v=Mqt_bBsejKQ
- Grafanalib: Dashboards as Code
  >>> https://www.youtube.com/watch?v=OOyEGG98B7w
  >>> https://www.weave.works/blog/grafana-dashboards-as-code/

{mthree}

Summary
Q&A

{mthree}

P A G E

Useful Prometheus Selectors

- probe_http_*
  - HTTP request information for the application
  - Provided you've added a probe to your namespace
- default_jenkins_*
  - default_jenkins_builds_duration_milliseconds_summary_sum
  - default_jenkins_builds_health_score
  - default_jenkins_builds_failed_build_count
  - default_jenkins_builds_success_build_count
  - default_jenkins_builds_last_build_result

- container_*
- kube_pod_*
- kube_deployment_*
- kube_endpoing_*
- kube_namepsace_*
- mysql_*
- nginx_ingress_controller_*
- probe_http_*
- probe_success

{mthree}

And also https://cXXXteam??dev.computerlab.online/metrics

Useful Loki LogQL

- {app=”kustomize-controller”}
  - >>> To see if your deployment is there, failing, or changed
  - >>> add |= “namespace”
    - ~ To view only your namespace
- {app=”ingress-nginx”} | json | host=”orderbookdev.computerlab.online”
  - >>> To view request/response details to a particular applications
    - ~ orderbookdev.computerlab.online is the application
- {app=”ingress-nginx”} | json | host=~”orderbookdev.*” | request_time > 0.005
  - >>> Checking if request time is > 5ms
- sum(count_over_time(({container=”orderbookac”} |= ”Steve” |= ”buy”)[1h]))
  - >>> The number of buy’s by Steve over the last 1 hour period
- sum(rate(({container=”orderbookac”} |= ”Steve” |= ”sell”)[1m]))
  - >>> The per second rate of all sell’s within the last minute
- {namespace=”orderbook-dev”}
  - >>> Get logs from containers running in your namespace

{mthree}