Prometheus 101

1- Install docker-compose on the worker node:

```
> sudo apt-get install docker-compose
```

To test for a successful installation, check the version using:

```
> docker-compose -v
```

2- Run prometheus and grafana in docker to see the final result:

You can run the whole setup of a simple ping server written in golang, Prometheus and Grafana using docker. Clone the following repo:
https://github.com/fc4it-k8s/prometheus-101
Then just run the following command:

```
> docker-compose up
```

http://localhost:9090 - prometheus UI
http://localhost:8090 - ping/pong service
http://localhost:3000/d/WdZVAykGk/ping-service?orgId=1 - grafana dashboard (admin:admin)

3- What are metrics and why is it important?

Metrics in layman terms is a standard for measurement. What we want to measure depends from application to application. For a web server it can be request times, for a database it can be CPU usage or number of active connections etc.

Metrics play an important role in understanding why your application is working in a certain way. If you run a web application and someone comes up to you and says that the application is slow. You will need some information to find out what is happening with your application. For example the application can become slow when the number of requests are high. If you have the request count metric you can spot the reason and increase the number of servers to handle the heavy load. Whenever you are defining the metrics for your application you must put on your detective hat and ask this question what all information will be important for me to debug if any issue occurs in my application?

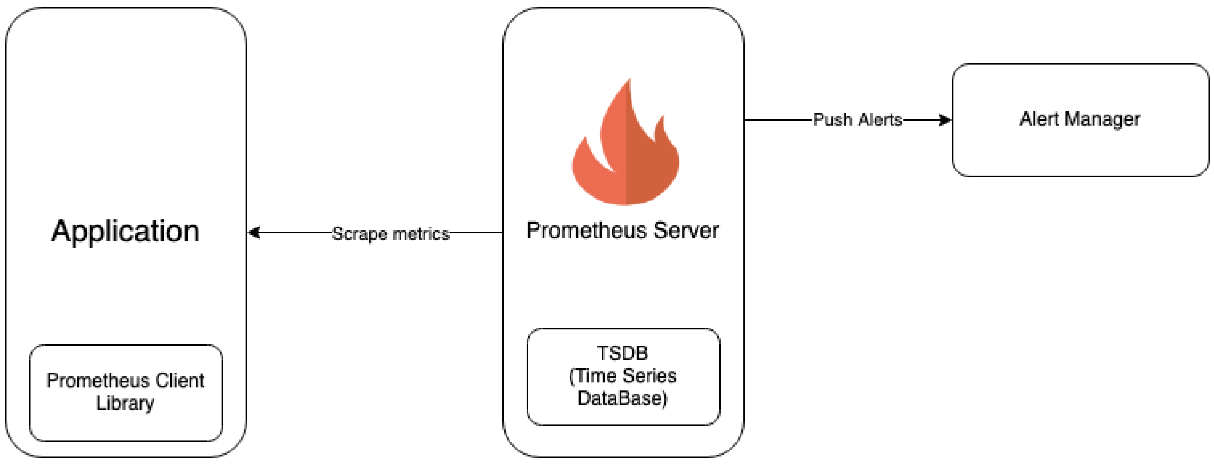
Analogy: I always use this analogy to simply understand what a monitoring system does. When I was young I wanted to grow tall and to measure it I used height as a metric. I asked my dad to measure my height everyday and keep a table of my height on each day. So in this case my dad is my monitoring system and the metric was my height.

- **Basic Architecture of Prometheus**

The basic components of prometheus are:

- Prometheus Server (The server which scrapes and stores the metrics data).
- Client Library which is used to calculate and expose the metrics.
- Alert manager to raise alerts based on preset rules.

(Note: Apart from this prometheus has push gateways which I am not covering here).



Let's consider a web server as an example application. I want to extract a certain metric like the number of API calls processed by my web server. So I add certain instrumentation code using the prometheus client library and expose the metrics information. Now that my web server also exposes its metrics I want it to be scraped by prometheus. So I run prometheus separately and configure it to fetch the metrics from the web server which is listening on xyz IP address port 7500 at a specific time interval, say, every minute.

I set prometheus up and expose my web server to be accessible for the frontend or other client to use.

At 11:00 PM when I make the server open to consumption. Prometheus scrapes the count metric and stores the value as 0

By 11:01 PM one request is processed. The instrumentation logic in my code makes the count to 1. When prometheus scraps the metric the value of count is 1 now

By 11:02 PM two requests are processed and the request count is 1+2 = 3 now. Similarly metrics are scraped and stored

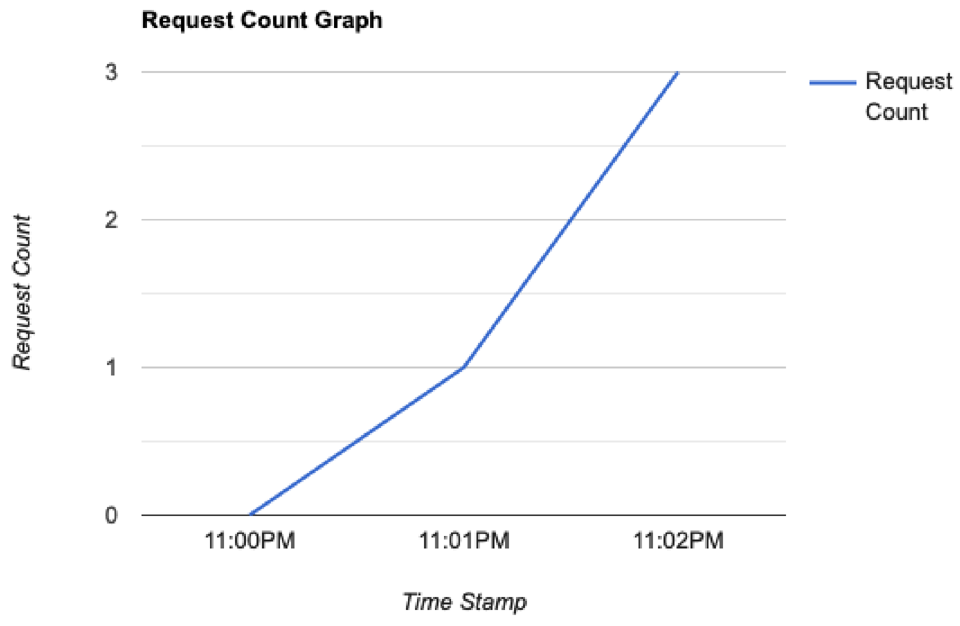
The user can control the frequency at which metrics are scraped by prometheus

Time Stamp	Request Count (metric)
11:00 PM	0
11:01 PM	1
11:02 PM	3

(Note: This table is just a representation for understanding purposes. Prometheus doesn't store the values in the exact format)

Prometheus also has a server which exposes the metrics which are stored by scraping. This server is used to query the metrics, create dashboards/charts on it etc. PromQL is used to query the metrics.

A simple Line chart created on my Request Count metric will look like this



I can scrape multiple metrics which will be useful to understand what is happening in my application and create multiple charts on them. Group the charts into a dashboard and use it to understand what is happening in my application.

4- how it is done.

Let’s get our hands dirty and setup prometheus. Prometheus is written using Go and all you need is the binary compiled for your operating system. Download the binary corresponding to your operating system from here and add the binary to your path.

Prometheus exposes its own metrics which can be consumed by itself or another prometheus server.

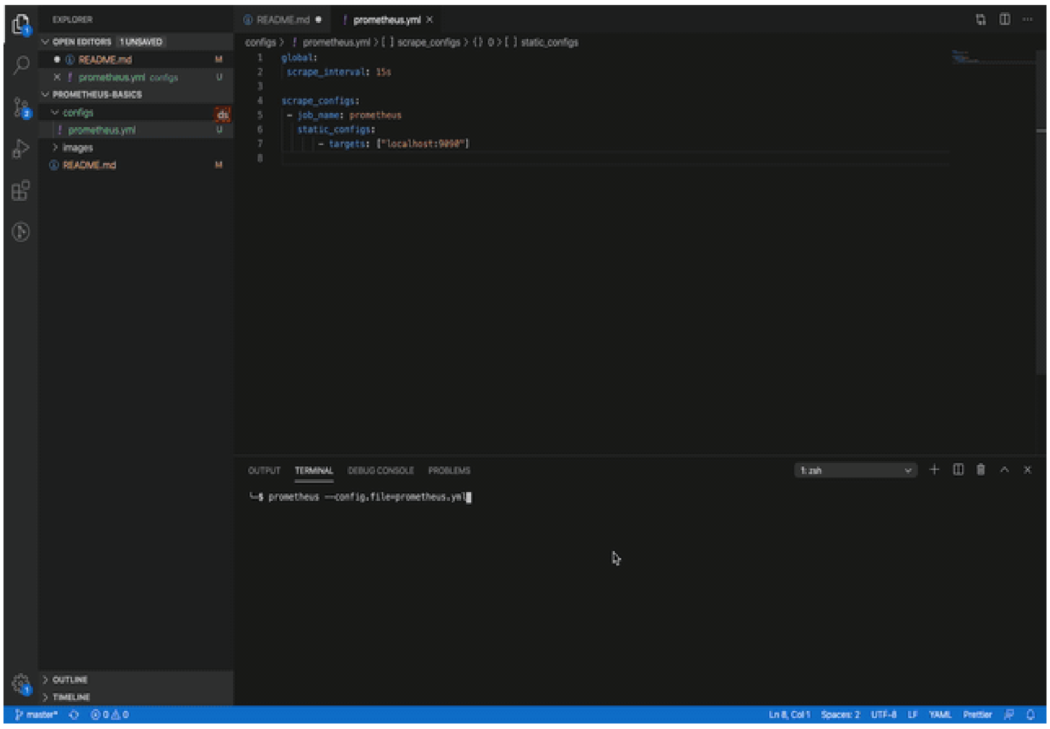
Now that we have Prometheus, the next step is to run it. All that we need is just the binary and a configuration file. Prometheus uses yaml files for configuration.

prometheus.yml

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ["localhost:9090"]
```

In the above configuration file we have mentioned the scrape_interval i.e how frequently you want prometheus to scrape the metrics. We have added **scrape_configs** which has a name and target to scrape the metrics from. Prometheus by default listens on port 9090. So we added it.

prometheus --config.file=prometheus.yml



configs > prometheus.yml > [ ] scrape_configs > {} 0 > [ ] static_configs

```yaml
global:
 scrape_interval: 15s

scrape_configs:
 - job_name: prometheus
   static_configs:
     - targets: ["localhost:9090"]
```

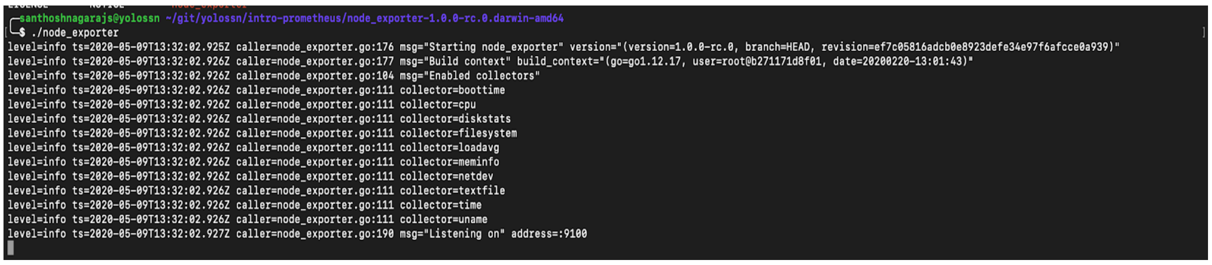
```
╰─$ prometheus --config.file=prometheus.yml
```

Now we have Prometheus up and running and scraping its own metrics every 15s. Prometheus has standard exporters available to export metrics. Next we will run a node exporter which is an exporter for machine metrics and scrape the same using prometheus. Download node metrics exporter from here

There are many standard exporters available like node exporter you can find them here

Run the node exporter in a terminal.

./node_exporter



```
┌─santhoshnagarajs@yolossn ~/git/yolossn/intro-prometheus/node_exporter-1.0.0-rc.0.darwin-amd64
└─$ ./node_exporter
level=info ts=2020-05-09T13:32:02.925Z caller=node_exporter.go:176 msg="Starting node_exporter" version="(version=1.0.0-rc.0, branch=HEAD, revision=ef7c05816adcb0e8923defe34e97f6afcce0a939)"
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:177 msg="Build context" build_context="(go=go1.12.17, user=root@b271171d8f01, date=20200220-13:01:43)"
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:104 msg="Enabled collectors"
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=boottime
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=cpu
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=diskstats
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=filesystem
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=loadavg
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=meminfo
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=netdev
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=textfile
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=time
level=info ts=2020-05-09T13:32:02.926Z caller=node_exporter.go:111 collector=uname
level=info ts=2020-05-09T13:32:02.927Z caller=node_exporter.go:190 msg="Listening on" address=:9100
```

Next Add node exporter to the list of scrape_configs

node_exporter.yml

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ["localhost:9090"]
  - job_name: node_exporter
    static_configs:
      - targets: ["localhost:9100"]
```

prometheus --config.file=node_exporter.yml

- **Types of metrics.**

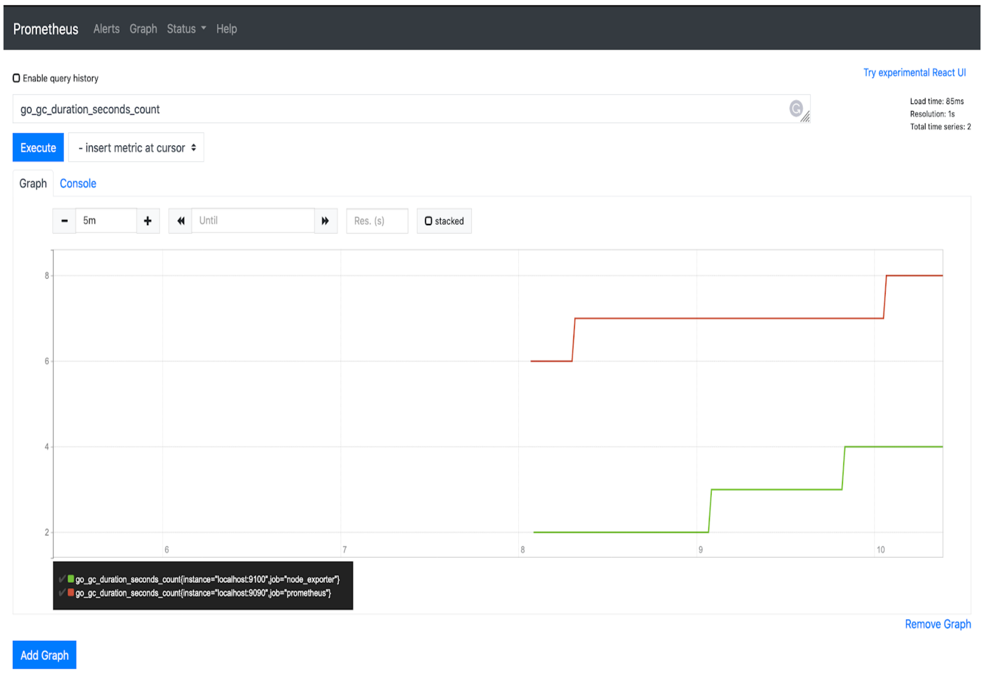
There are four types of metrics which prometheus client libraries support as of May 2020. They are:

1. Counter
2. Gauge
3. Histogram
4. Summary

- **Counter:**

Counter is a metric value which can only increase or reset i.e the value cannot reduce than the previous value. It can be used for metrics like number of requests, no of errors etc.

go_gc_duration_seconds_count



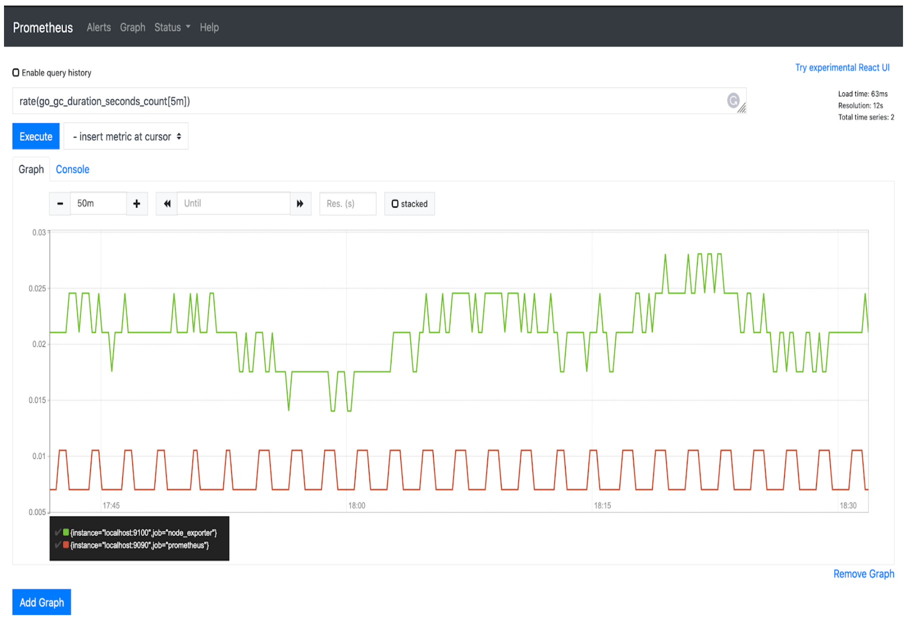
```
go_gc_duration_seconds_count
```

Load time: 85ms
Resolution: 1s
Total time series: 2

✓ 🟩 go_gc_duration_seconds_count{instance="localhost:9100",job="node_exporter"}
✓ 🟥 go_gc_duration_seconds_count{instance="localhost:9090",job="prometheus"}

The rate() function takes the history of metrics over a time frame and calculates how fast value is increasing per second. Rate is applicable on counter values only.

rate(go_gc_duration_seconds_count[5m])



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

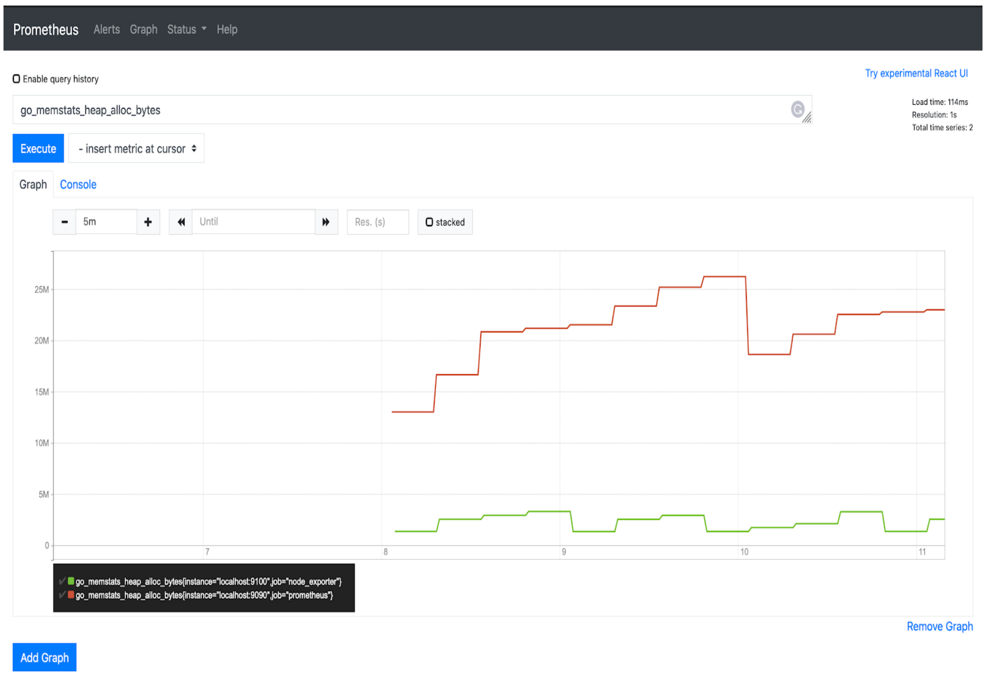
Load time: 63ms
Resolution: 12s
Total time series: 2

✓ 🟩 {instance="localhost:9100",job="node_exporter"}
✓ 🟥 {instance="localhost:9090",job="prometheus"}

- **Gauge:**

Gauge is a number which can either go up or down. It can be used for metrics like number of pods in a cluster, number of events in an queue etc.

go_memstats_heap_alloc_bytes



```
go_memstats_heap_alloc_bytes
```

Load time: 114ms
Resolution: 1s
Total time series: 2

🟩 go_memstats_heap_alloc_bytes{instance="localhost:9100",job="node_exporter"}
🟥 go_memstats_heap_alloc_bytes{instance="localhost:9090",job="prometheus"}

PromQL functions like **max_over_time**, **min_over_time** and **avg_over_time** can be used to query gauge metrics

- **Histogram**

Histogram is a little complex metric type when compared to the ones we have seen. Histogram can be used for any calculated value which is counted based on bucket values, the buckets can be configured by the user. It is a cumulative metric and provides sum of all the values by default. This is applicable for metrics like request time, cpu temperature etc

Example: I want to observe the time taken to process api requests. Instead of storing the request time for each request, histogram metric allows us to approximate and store frequency of requests which fall into particular buckets. I define buckets for time taken like 0.3,0.5,0.7,1,1.2. So these are my buckets and once the time taken for a request is calculated I add the count to all the buckets which are higher than the value.

Lets say Request 1 for endpoint **"/ping"** takes 0.25 s. The count values for the buckets will be.

Bucket	Count
0 - 0.3	1
0.3 - 0.5	1
0.5 - 0.7	1
0.7 - 1	1
1 - 1.2	1
+Inf	1

Note: +Inf bucket is added by default.

(Since histogram is a cumulative frequency 1 is added to all the buckets which are greater than the value)

Request 2 for endpoint “/ping” takes 0.4s The count values for the buckets will be this.

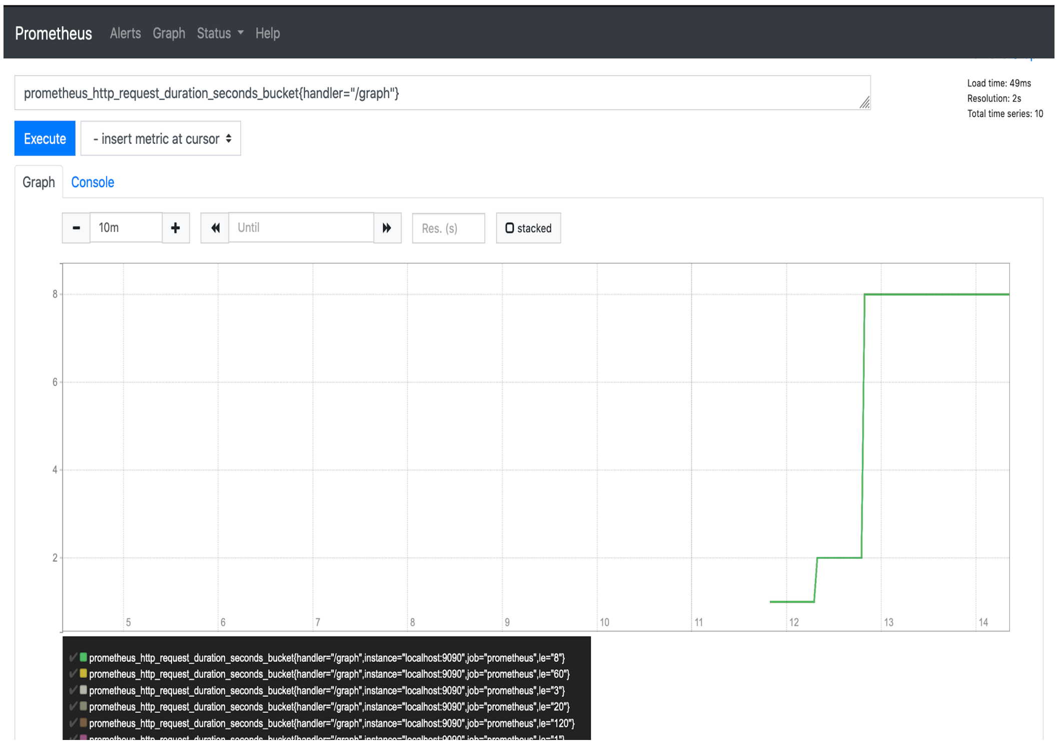
/ping

Bucket	**Count**
0 - 0.3	1
0.3 - 0.5	2
0.5 - 0.7	2
0.7 - 1	2
1 - 1.2	2
+Inf	2

Since 0.4 lies in the seconds bucket(0.3-0.5) all the buckets above the calculated values count is increased. Histogram is used to find average and percentile values.

Let's see a histogram metric scraped from prometheus and apply few functions.

prometheus_http_request_duration_seconds_bucket{handler="/graph"}

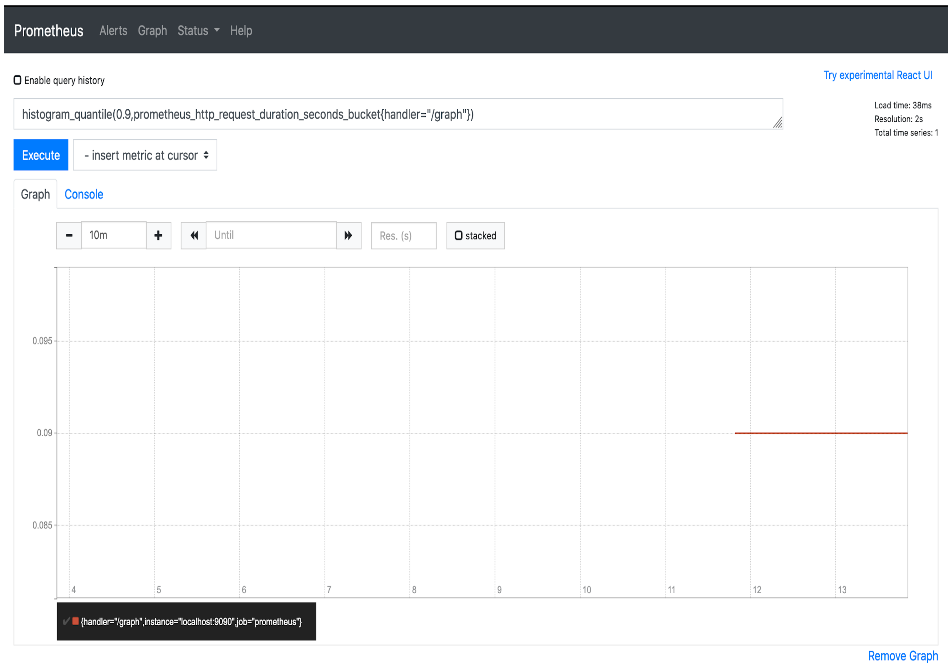


```
prometheus_http_request_duration_seconds_bucket{handler="/graph"}
```

Load time: 49ms
Resolution: 2s
Total time series: 10

```
✓ 🟩 prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="8"}
✓ 🟨 prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="60"}
✓ ⬜ prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="3"}
✓ ▨ prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="20"}
✓ 🟫 prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="120"}
✓ 🟪 prometheus_http_request_duration_seconds_bucket{handler="/graph",instance="localhost:9090",job="prometheus",le="1"}
```

histogram_quantile() function can be used to calculate calculate quantiles from histogram



Enable query history

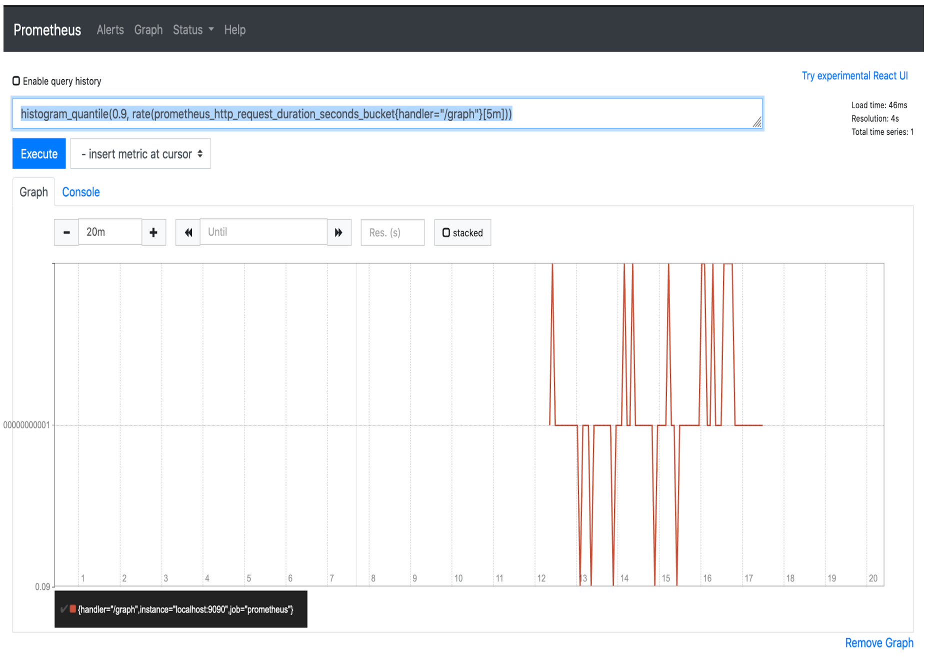
```
histogram_quantile(0.9,prometheus_http_request_duration_seconds_bucket{handler="/graph"})
```

Load time: 38ms
Resolution: 2s
Total time series: 1

```
✓ 🟥 {handler="/graph",instance="localhost:9090",job="prometheus"}
```

The graph shows that the 90th percentile is 0.09, To find the histogram_quantile over last 5m you can use the rate() and time frame

histogram_quantile(0.9, rate(prometheus_http_request_duration_seconds_bucket{handler="/graph"}[5m]))



```
histogram_quantile(0.9, rate(prometheus_http_request_duration_seconds_bucket{handler="/graph"}[5m]))
```

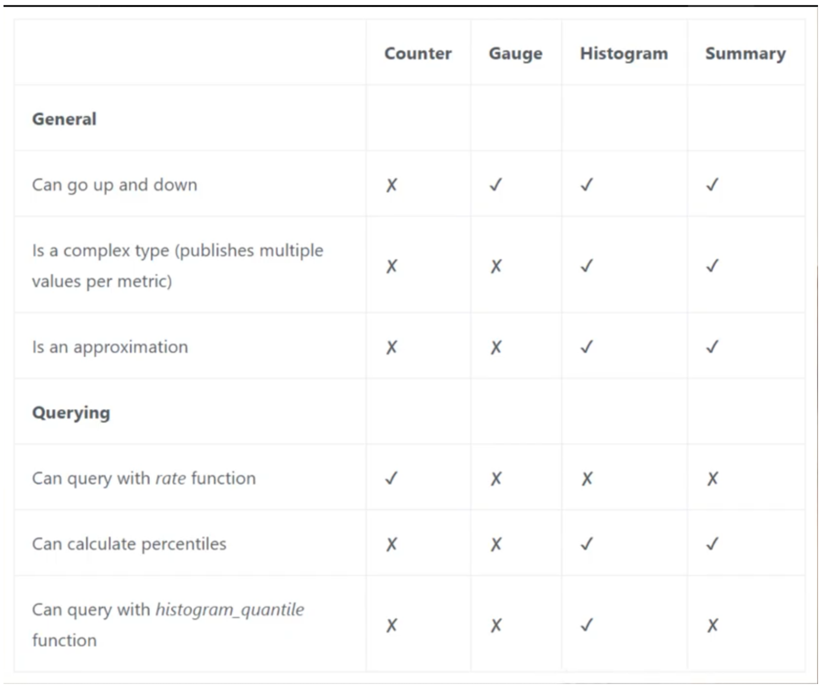
```
Load time: 46ms
Resolution: 4s
Total time series: 1
```

```
{handler="/graph",instance="localhost:9090",job="prometheus"}
```

- Summary

Summary is similar to histogram and calculates quantiles which can be configured, but it is calculated on the application level hence aggregation of metrics from multiple instances of the same process is not possible. It is used when the buckets of a metric is not known beforehand and is highly recommended to use histogram over summary whenever possible. Calculating summary on the application level is also quite expensive and it is not recommended to be used.

- Short Summary on Metric Types:



	Counter	Gauge	Histogram	Summary
General				
Can go up and down	✗	✓	✓	✓
Is a complex type (publishes multiple values per metric)	✗	✗	✓	✓
Is an approximation	✗	✗	✓	✓
Querying				
Can query with *rate* function	✓	✗	✗	✗
Can calculate percentiles	✗	✗	✓	✓
Can query with *histogram_quantile* function	✗	✗	✓	✗

5- Create instrumented application

We will create a request counter metric exporter using Go.

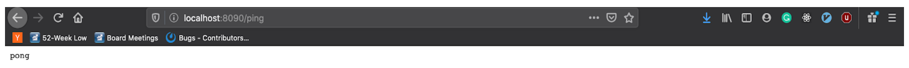
server.go

```
package main

import (
   "fmt"
   "net/http"
)

func ping(w http.ResponseWriter, req *http.Request){
   fmt.Fprintf(w,"pong")
}

func main() {
   http.HandleFunc("/ping",ping)

   http.ListenAndServe(":8090", nil)
}
```

Compile and run the server

```
go build server.go
./server.go
```

We have a simple server which when hit on localhost:8090/ping endpoint sends back pong



```
pong
```

Now we will add a metric to the server which will instrument the number of requests made to the ping endpoint.

We will use the counter metric type for this as we know the request count doesn't go down and only increases.

Create a prometheus counter

```
var pingCounter = prometheus.NewCounter(
   prometheus.CounterOpts{
       Name: "ping_request_count",
       Help: "No of request handled by Ping handler",
   },
)
```

Next we update the ping Handler to increase the count of the counter using `pingCounter.Inc()`.

```
func ping(w http.ResponseWriter, req *http.Request) {
    pingCounter.Inc()
    fmt.Fprintf(w, "pong")
}
```

Next we register the counter to the Default Register and expose the metrics.

```
func main() {
    prometheus.MustRegister(pingCounter)
    http.HandleFunc("/ping", ping)
    http.Handle("/metrics", promhttp.Handler())
    http.ListenAndServe(":8090", nil)
}
```

The `prometheus.MustRegister` function registers the pingCounter to the default Register. To expose the metrics the Go Prometheus client library provides the promhttp package. `promhttp.Handler()` provides a `http.Handler` which exposes the metrics registered in the Default Register.

Here is the complete ***serverWithMetric.go*** at this point.

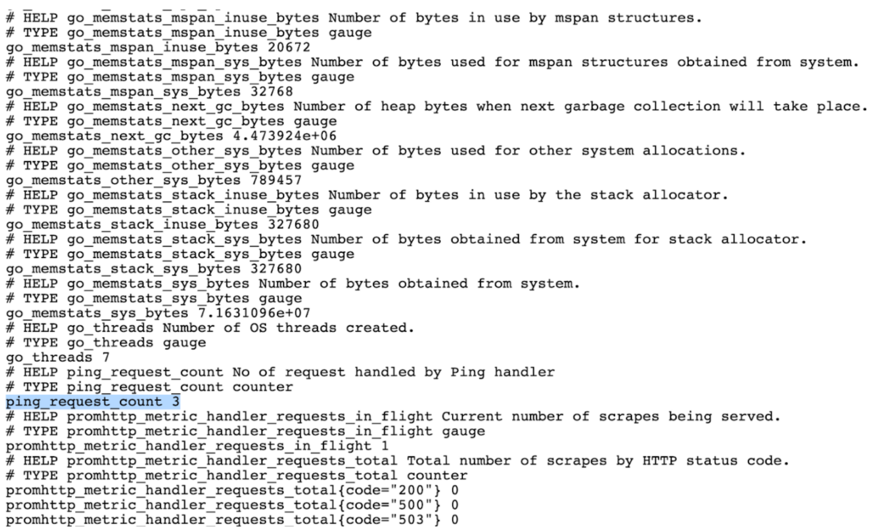
```
package main

import (
    "fmt"
    "net/http"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

var pingCounter = prometheus.NewCounter(
    prometheus.CounterOpts{
        Name: "ping_request_count",
        Help: "No of request handled by Ping handler",
    },
)

func ping(w http.ResponseWriter, req *http.Request) {
    pingCounter.Inc()
    fmt.Fprintf(w, "pong")
}
```

```
func main() {
   prometheus.MustRegister(pingCounter)

   http.HandleFunc("/ping", ping)
   http.Handle("/metrics", promhttp.Handler())
   http.ListenAndServe(":8090", nil)
}
```

- **Re-compile the binary**

```
go get github.com/prometheus/client_golang/prometheus
go get github.com/prometheus/client_golang/prometheus/promhttp
go build server.go
./server.go
```

Now hit the localhost:8090/ping endpoint a couple of times and sending a request to localhost:8090 will provide the metrics.



```
# HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.
# TYPE go_memstats_mspan_inuse_bytes gauge
go_memstats_mspan_inuse_bytes 20672
# HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures obtained from system.
# TYPE go_memstats_mspan_sys_bytes gauge
go_memstats_mspan_sys_bytes 32768
# HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection will take place.
# TYPE go_memstats_next_gc_bytes gauge
go_memstats_next_gc_bytes 4.473924e+06
# HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.
# TYPE go_memstats_other_sys_bytes gauge
go_memstats_other_sys_bytes 789457
# HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.
# TYPE go_memstats_stack_inuse_bytes gauge
go_memstats_stack_inuse_bytes 327680
# HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack allocator.
# TYPE go_memstats_stack_sys_bytes gauge
go_memstats_stack_sys_bytes 327680
# HELP go_memstats_sys_bytes Number of bytes obtained from system.
# TYPE go_memstats_sys_bytes gauge
go_memstats_sys_bytes 7.1631096e+07
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 7
# HELP ping_request_count No of request handled by Ping handler
# TYPE ping_request_count counter
ping_request_count 3
# HELP promhttp_metric_handler_requests_in_flight Current number of scrapes being served.
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 1
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP status code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 0
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

Here the ping_request_count shows that `/ping` endpoint was called 3 times.

The DefaultRegister comes with a collector for go runtime metrics and that is why we see other metrics like go_threads, go_goroutines etc.

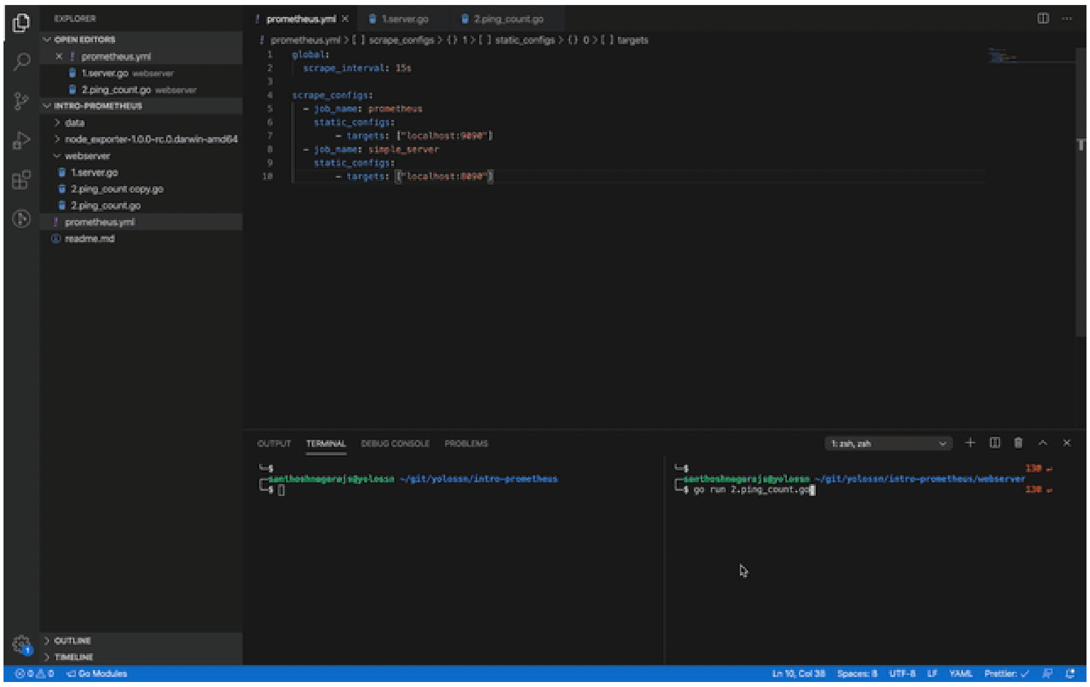
We have built our first metric exporter. Let's update our prometheus config to scrape the metrics from our server.

simple_server.yml

```
global:

  scrape_interval: 15s

scrape_configs:

  - job_name: prometheus
```

```
    static_configs:
      - targets: ["localhost:9090"]
  - job_name: simple_server
    static_configs:
      - targets: ["localhost:8090"]
```

prometheus --config.file=simple_server.yml



```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
        - targets: ["localhost:9090"]
  - job_name: simple_server
    static_configs:
        - targets: ["localhost:8090"]
```

```
╰─$
┌─santhoshnagarajs@yolossn ~/git/yolossn/intro-prometheus
└─$ 
```

```
╰─$
┌─santhoshnagarajs@yolossn ~/git/yolossn/intro-prometheus/webserver
└─$ go run 2.ping_count.go
```

Note:

- Make sure the authentication mechanism used by your application is supported by prometheus
- `promhttp.Handler` gzips the response, If you are using a gzip middleware then you must implement some skipper logic to avoid compressing the response twice.

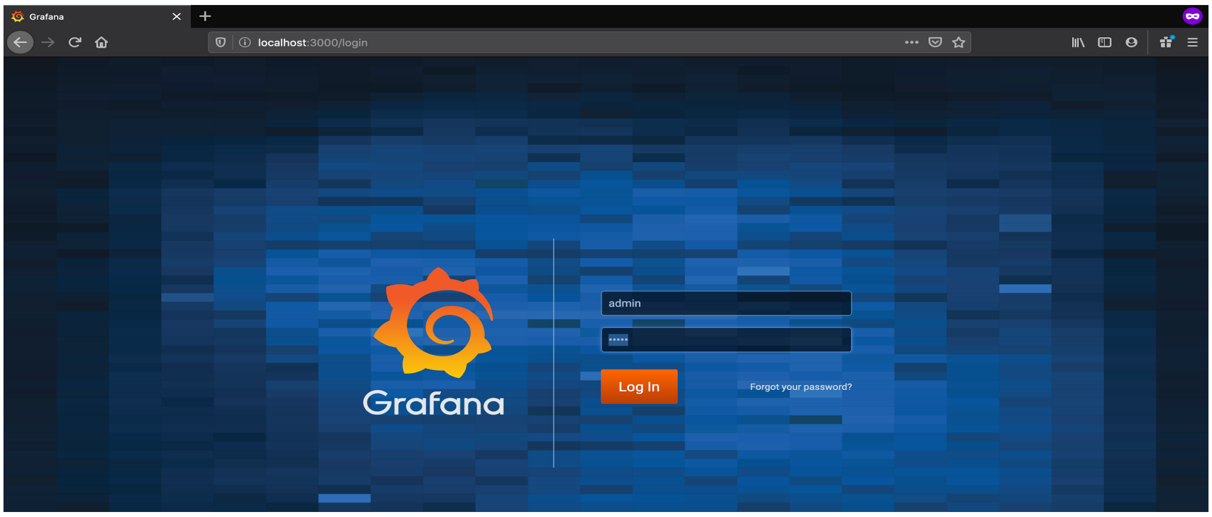
6- Use Grafana to visualize metrics

Now that we have our server with `ping_request_count` metric let's create a visualization dashboard. For this, we will use Grafana. If you wonder why should one use Grafana when we can create graphs using Prometheus. The answer is that the graph that we use to visualize our queries is used for ad-hoc queries and debugging. Prometheus official docs suggest using Grafana or Console Templates for graphs. Refer

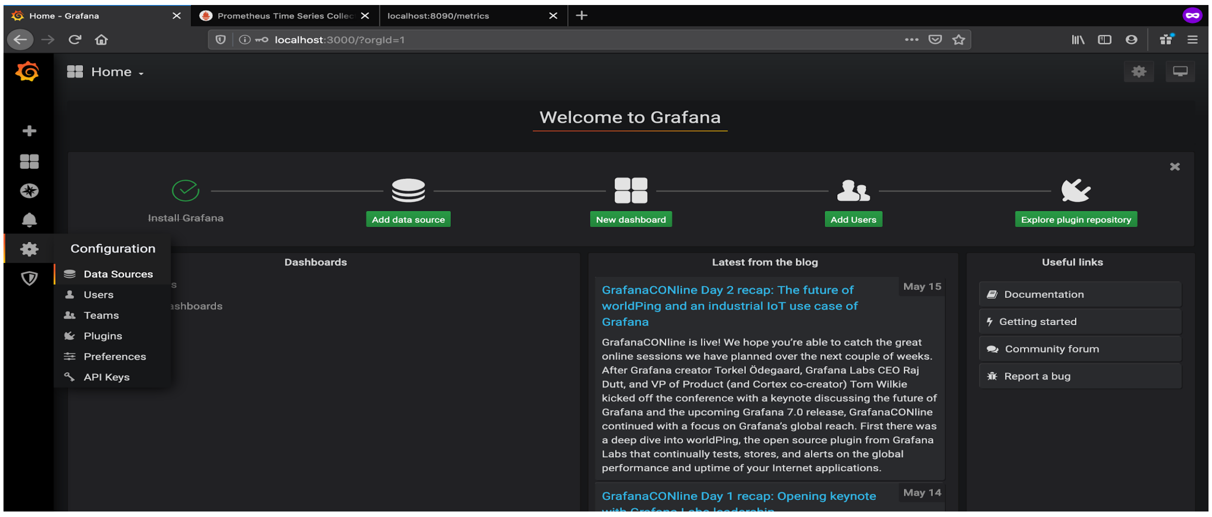
Console Templates is a way to create graphs using Go templates which I am not covering as it has a learning curve. Grafana is an analytics platform that allows you to query,visualize, and set alerts on your metrics. Comparatively Grafana is easy to use for a beginner.

Install Grafana by following the steps for your operating system from here.

Once Grafana is installed successfully go to localhost:3000 and Grafana dashboard should be visible. The default username and password is `admin`.



Let's add a source to grafana by clicking on the gear icon in the side bar and select Data Sources.



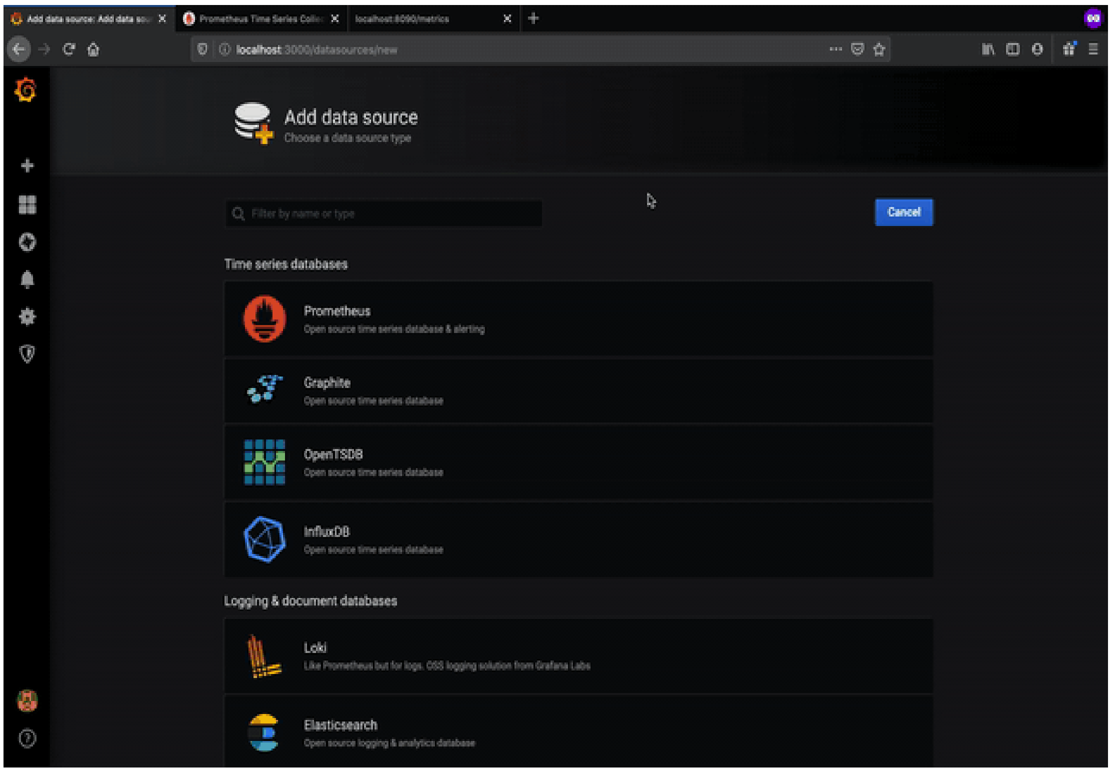
GrafanaCONline Day 2 recap: The future of worldPing and an industrial IoT use case of Grafana May 15

GrafanaCONline is live! We hope you're able to catch the great online sessions we have planned over the next couple of weeks. After Grafana creator Torkel Ödegaard, Grafana Labs CEO Raj Dutt, and VP of Product (and Cortex co-creator) Tom Wilkie kicked off the conference with a keynote discussing the future of Grafana and the upcoming Grafana 7.0 release, GrafanaCONline continued with a focus on Grafana's global reach. First there was a deep dive into worldPing, the open source plugin from Grafana Labs that continually tests, stores, and alerts on the global performance and uptime of your Internet applications.

GrafanaCONline Day 1 recap: Opening keynote May 14

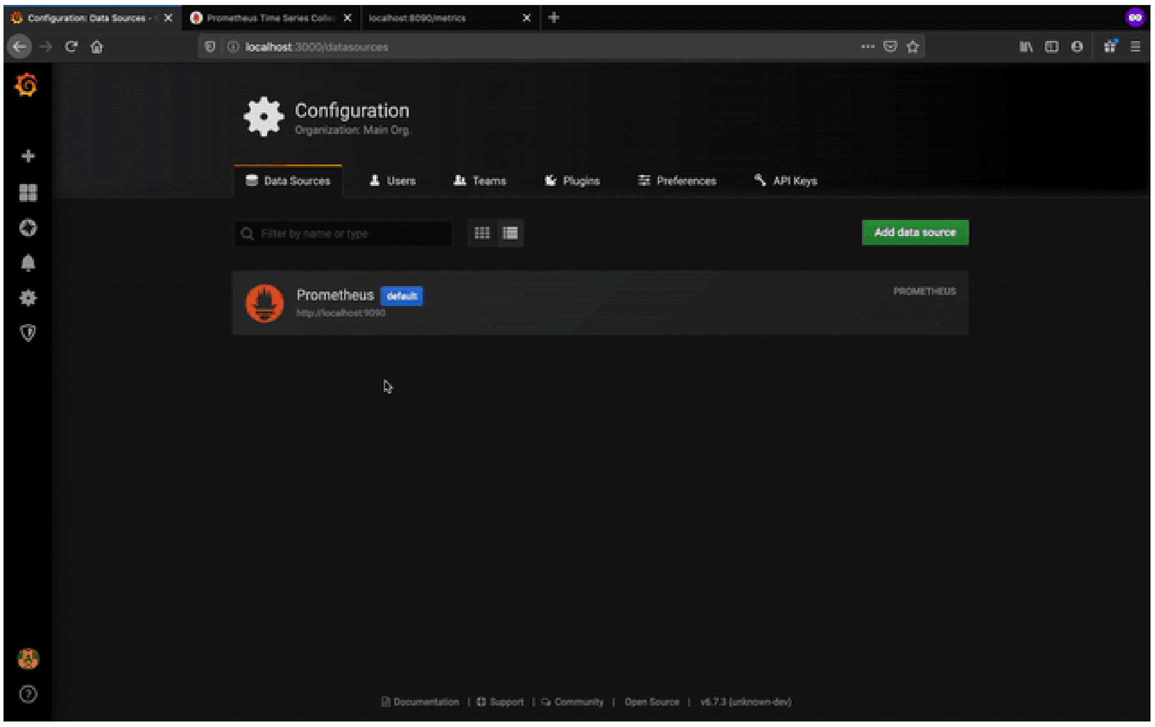
If you see Grafana also supports multiple data sources other than Prometheus like graphite, postgreSQL etc.

Adding Prometheus as Data Source


Add data source: Add data sou
Prometheus Time Series Colle
localhost:8090/metrics
localhost:3000/datasources/new
Add data source
Choose a data source type
Filter by name or type
Cancel
Time series databases
Prometheus
Open source time series database & alerting
Graphite
Open source time series database
OpenTSDB
Open source time series database
InfluxDB
Open source time series database
Logging & document databases
Loki
Like Prometheus but for logs. OSS logging solution from Grafana Labs
Elasticsearch
Open source logging & analytics database

Now we have successfully added Prometheus as our data source.

Creating your first dashboard


Configuration: Data Sources -
Prometheus Time Series Colle
localhost:8090/metrics
localhost:3000/datasources
Configuration
Organization: Main Org.
Data Sources
Users
Teams
Plugins
Preferences
API Keys
Filter by name or type
Add data source
Prometheus
default
http://localhost:9090
PROMETHEUS
Documentation | Support | Community | Open Source | v6.7.3 (unknown-dev)