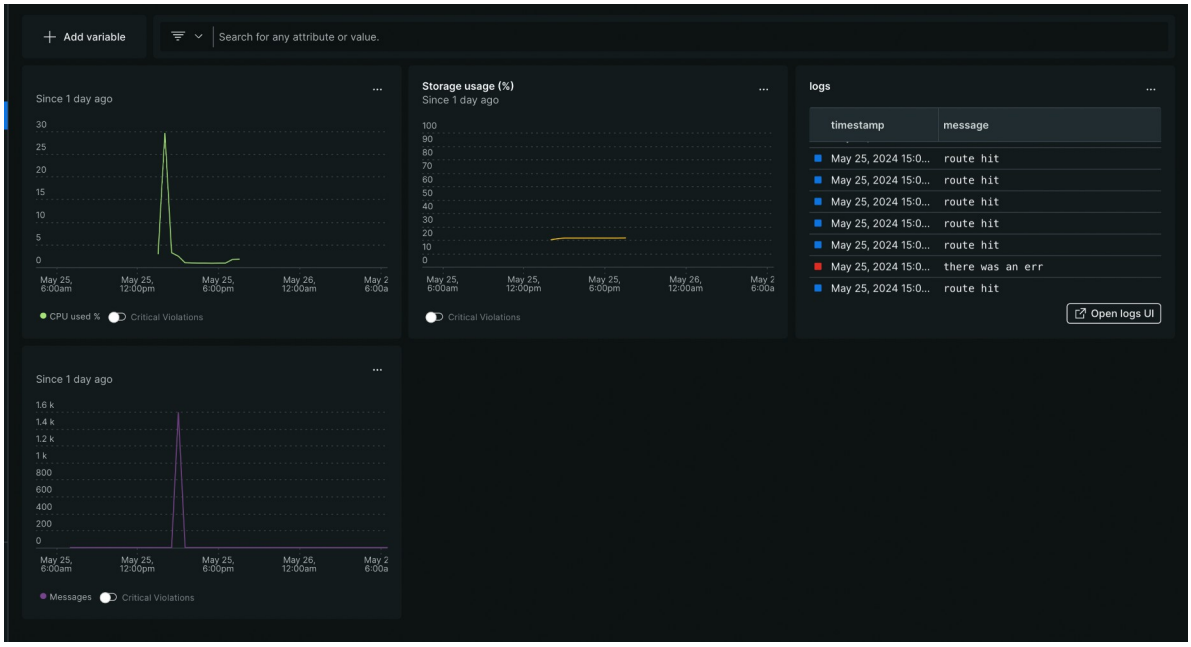
Prometheus and Grafana (Open source monitoring tools)
DATADOG AND NEWRELIC PAID ONES

for this we have to learn prometheus query language

Revise monitoring

In the last live, we understood what is monitoring.
We created dashboards that looked like the following -

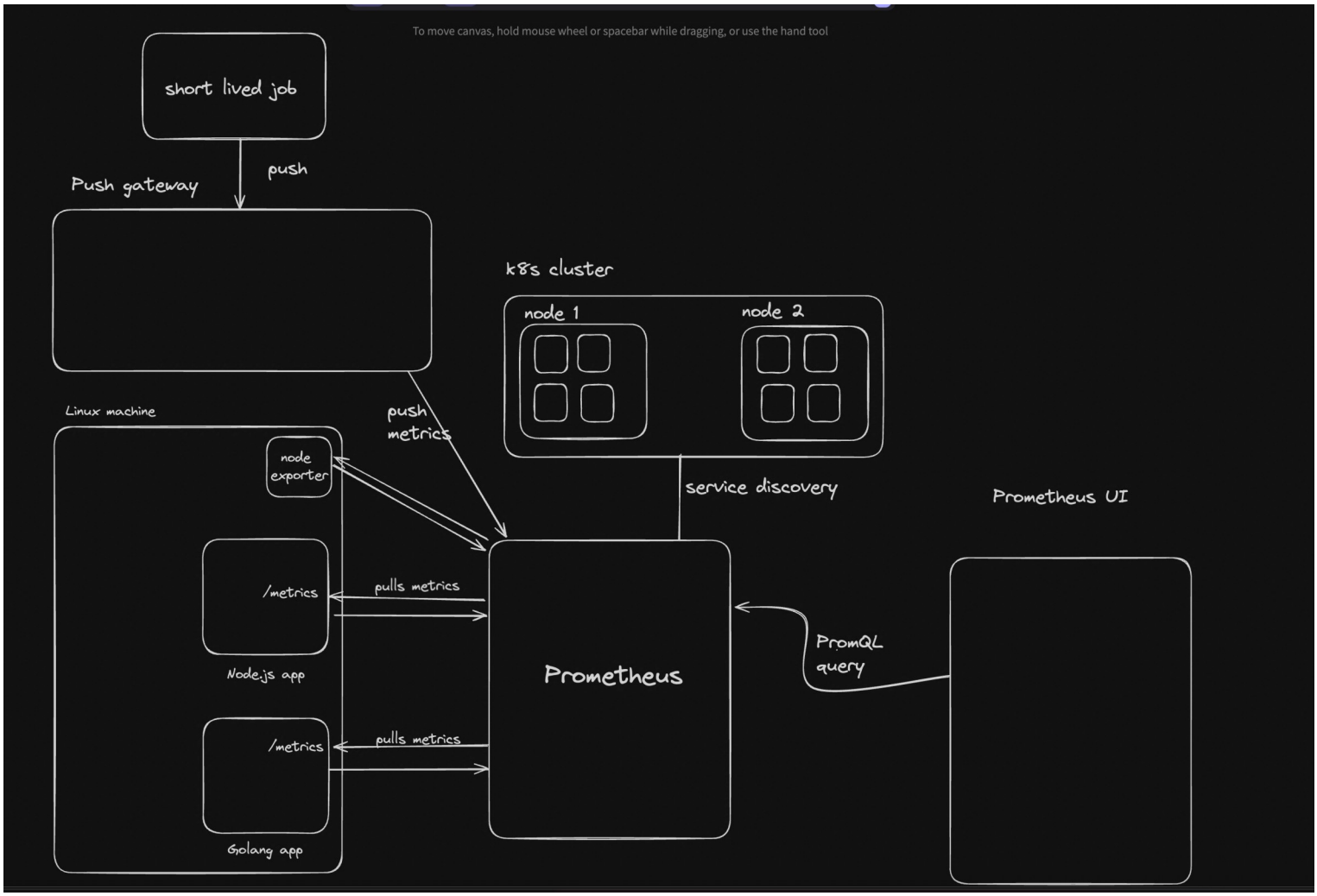


Problem with newrelic

1. It’s paid and can never be self hosted
2. They own your data
3. Very hard to move away from it once it’s ingrained in your system

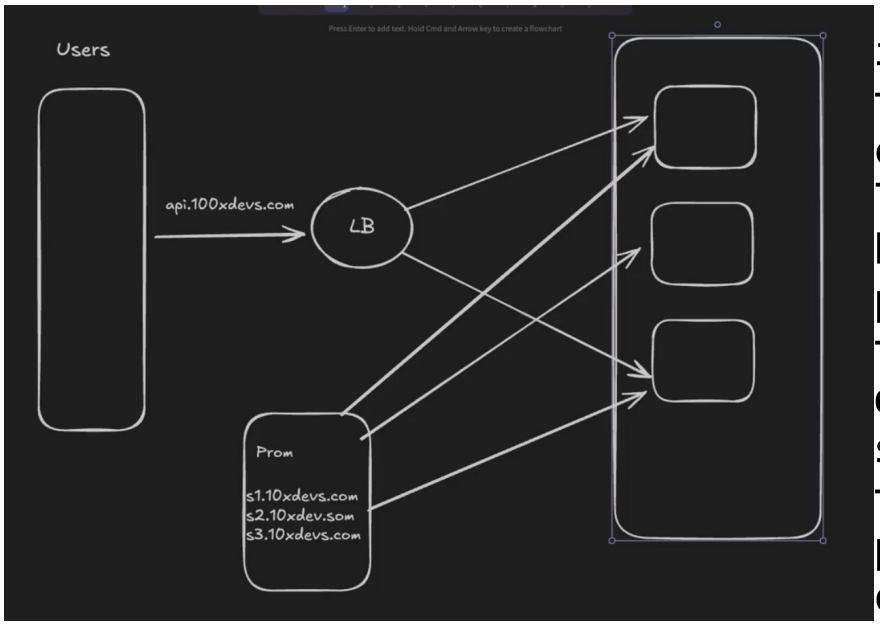
Prometheus works on a pull architecutre
Prometheus is a time series DB. It can monitor your

1. Processes (node, go, rust…)
2. Hosts



in this how it works is we create a linux machine which runs our code and also which contains things such as node js app, go lang app etc now to send the metric to prometheus which is running on some other machine it need to somehow get some data whci it get using the node exporter which the prometheus dev team has written which require a specific endpoint such as /metrics endpoint from both the application in which the developer has to send all the log then the prometheus machine can get the code and also this prometh code must be on another system as later we can use more things such as a kubernetes cluster, a short lived job (like a video transcoding) it stores them in a timeseries format

and for the node application it uses the pull architecture as it know the data is stored in the /metrics endpoint of golang.100xdevs.com/metrics and also for the transcoding machine it will not use pull arch it will use the push arch



in this if you have a three systems running is controlled by a loadbalancer which is hitted by the user so the prometheus should not hit the lb it should hit directly the system only so we should make sure that the prometheus is pointed to the systems only not the lb for the metrics

for kubernetes the prometheus use something called as service discovery to work
we visualize using the prometheus ui and is close to newrelic ui
GRAFANA Is just alternative to PROMETHEUS UI it under the hood uses the prometheus only but grafana provides more charts and all as compared to prometheus ui

---

OVERVIEW

What is Prometheus?

- What is Prometheus?
  - Features
  - What are metrics?
  - Components
  - Architecture
- When does it fit?
- When does it not fit?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.

For more elaborate overviews of Prometheus, see the resources linked from the media section.

Features

Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

prometheus cannot scale horizontally which means it cannot run on multiple system we have to just increase the compute of the main system to increase the load if we want(you have to use sharding)

we don't care that much for the old data as we keep only like 3-4 months of latest data only

SO PROMETHEUS IS A TIME SERIES DB THAT IS SCRAPPING DATA FROM BUNCH OF SOURCES AND PUTTING THEM IN A DB IN A TIME SERIES FASHION
TIME SERIES DB == IN THIS WE ARE DOING ENTRY EVERY 10 SEC USED IN BACKPACK
DB USED SUCH AS TIMESCALED DB USED FOR CHARTS,GRAPHS,TRADES

SO WHAT ALL THINGS WE WILL DO TODAY?
So we will create a machine with the node js code and the metrics end point exposed connected to prometheus and also which shows the prom ui and then make individual docker file of these and then just create a docker compose file to run these three file together in one go

now we are creating a node js machine so what all things can measure
request count, request duration, total number of 404 request, total number of 500 req, cpu usage of a process,

```
import express from "express";
const app = express();
app.get("/cpu",(req,res) =>{
    const startTime = Date.now();
    for(let i=0;i<100000;i++){
        Math.random()
    }
    res.json({
        message:"cpu"
    })
    const endTime = Date.now();
    console.log(`Time it took is ${endTime-startTime}`)
})
app.get("/users",(req,res) =>{
    const startTime = Date.now();
    res.json({
        message:"user"
    })
    const endTime = Date.now();
    console.log(`Time it took is ${endTime-startTime}ms`)
})
```

initialize a empty project bun init
bun add express@types/express

now see this works but there is code rep and it does not support any random endpoint to fix that use middleware

create the middleware and also in the index.ts use app.use(middleware) to use in all the request and if in just the endpoint use like this

app.get("/cpu",middleware,(req,res))

now it gives ms for every request

you can also do this
console.log(`Time it took is ${endTime-startTime}ms for method ${req.method} for route ${req.route.path}`)
Time it took is 3ms for method GET for route /cpu

Add prometheus
Lets try putting this data inside prometheus next.
Types of metrics in Prometheus
Counter

- A counter is a cumulative metric that only increases.
- Example: Counting the number of HTTP requests.

Gauge

- A gauge is a metric that can go up and down. It can be used to measure values that fluctuate, such as the current number of active users or the current memory usage.
- Example: Measuring the current memory usage,number of active users, number of pending req, mainly used in websocket server in which people are chatting or smthg

Histogram

- A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

COUNTER WE MOSTLY USE WHEN SOMETHING IS INCREASING AND WE WANT TO USE

Now how to use the counter the main thing is that

```
import express from "express";
import type{ Request, Response, NextFunction } from "express";
import client from "prom-client";
const requestCounter = new client.Counter({
    name: 'http_requests_total',
    help: 'Total number of HTTP requests',
    labelNames: ['method', 'route', 'status_code']
});
export const requestCountMiddleware = (req: Request, res: Response, next:
NextFunction) => {
    const startTime = Date.now();
    res.on('finish', () => {
        const endTime = Date.now();
        console.log(`Request took ${endTime - startTime}ms`);

        // Increment request counter
        requestCounter.inc({
            method: req.method,
            route: req.route ? req.route.path : req.path,
            status_code: res.statusCode
        });
    });
    next();
};
```

in this the request counter has things such as the name, description, and the labelname
which is very usefull in montoring things

method: such as get, post, put, delete,patch, trace etc

route: which all routes are hitted
status_code: 500,404 etc
this all are the metric which will be shown

now in this the middleware will help us get the actual time and all , the res.on command will run only when the task is fully done means the route is hitted after that a log will be created with all the labels with a increase in counter

```
app.get("/metrics", async (req, res) => {
    const metrics = await client.register.metrics();
    res.set('Content-Type', client.register.contentType);
    res.end(metrics);
})
```

this helps to register the metric which the prometheus system will use to show the graphs and all….
In this the second line help to take data in format like html, json format etc
Counter can only be incremented , gauges can be incremented and decremented both and gauge can also contain the labels thing if we want

Gauges are all same just the increase decrease feature helps us to identify things like total number of users of the system etc

- Create `metrics/activeRequests.ts` , export a `Gauge` from it

```ts
import client from "prom-client";

export const activeRequestsGauge = new client.Gauge({
  name: 'active_requests',
  help: 'Number of active requests'
});
```

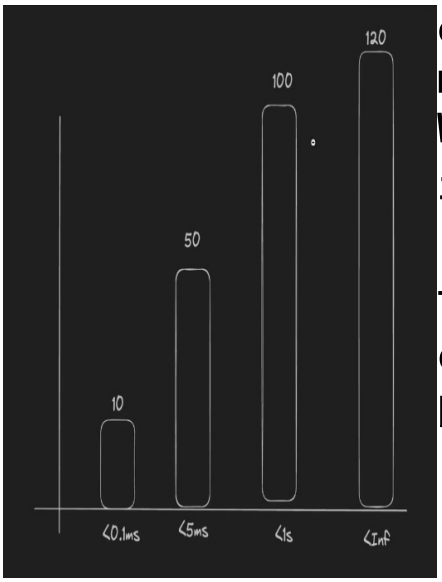
- Import it and update `metrics/index.ts`

```ts
import { NextFunction, Request, Response } from "express";
import { requestCounter } from "./requestCount";
import { activeRequestsGauge } from "./activeRequests";

export const cleanupMiddleware = (req: Request, res: Response, next: NextFu
  const startTime = Date.now();
  activeRequestsGauge.inc();

  res.on('finish', function() {
    const endTime = Date.now();
    console.log(`Request took ${endTime - startTime}ms`);

    requestCounter.inc({
      method: req.method,
      route: req.route ? req.route.path : req.path,
      status_code: res.statusCode
    });
    activeRequestsGauge.dec();
  });
}
```

in this its all same just the inc and after the thing finished a dec function is also added

```
# HELP http_requests_total Total number of HTTP requests
# TYPE http_requests_total counter
http_requests_total{method="GET",route="/metrics",status_code="200"} 3
http_requests_total{method="GET",route="/user",status_code="304"} 3

# HELP active_requests Number of active requests
# TYPE active_requests gauge
active_requests 4
```

Histograms
Histograms let you store data in various buckets in a cumulative fashion



cumulative means the 1s will contains the 5 ms data also
Which is like 50+ 50(for the 1s one) that is cumulative

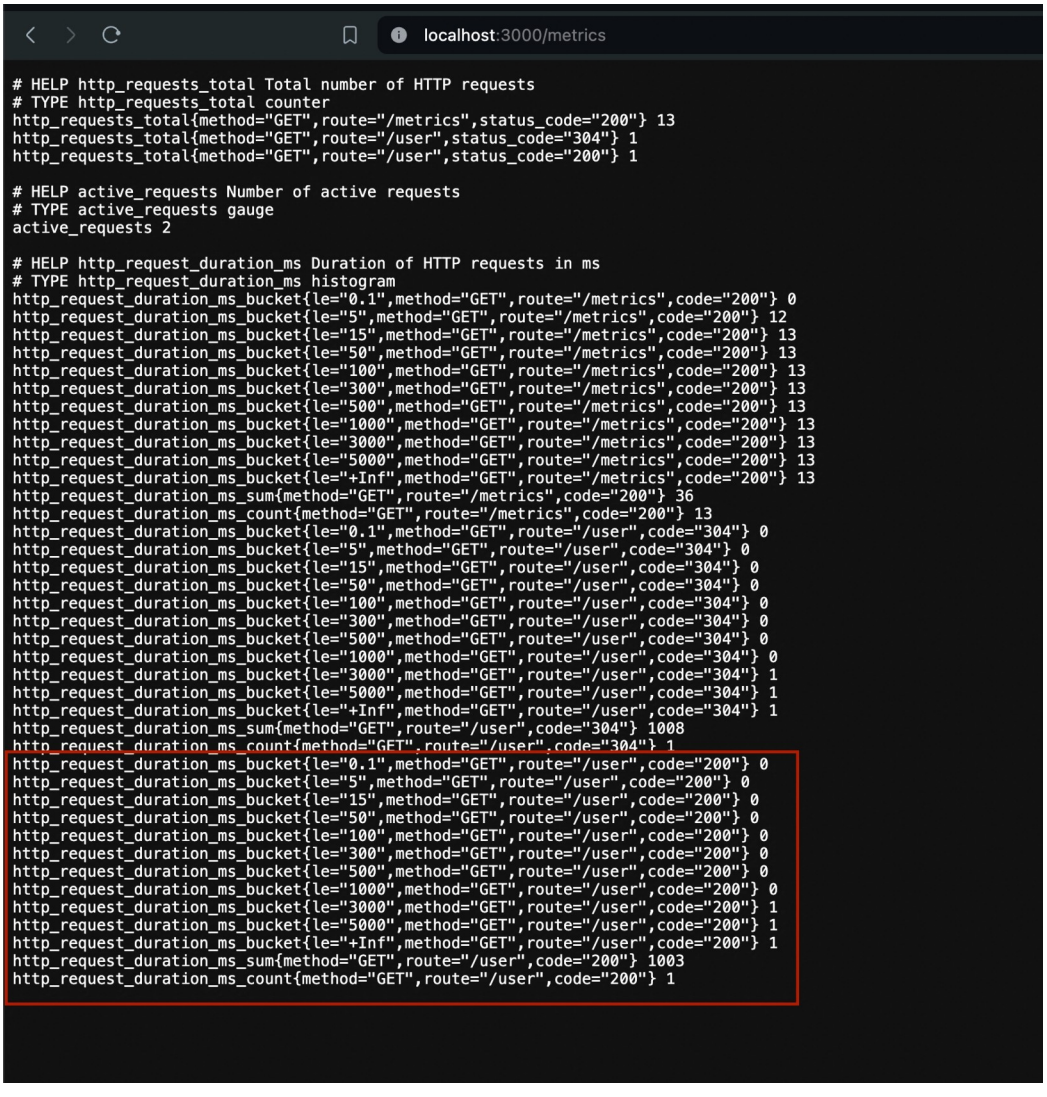
the parameter in which we are measuring is called as bucket eg 0.1ms , 5ms, 1s are all buckets in which we measure the data

```
import client from "prom-client";

export const httpRequestDurationMicroseconds = new
client.Histogram({
    name: 'http_request_duration_ms',
    help: 'Duration of HTTP requests in ms',
    labelNames: ['method', 'route', 'code'],
    buckets: [0.1, 5, 15, 50, 100, 300, 500, 1000, 3000,
5000] // Define your own buckets here
});
```

Buckets here represent the key points you want to measure in your app.
How many people had request handled in 0.1ms, 5ms, 15ms …
This is because prometheus is not a DB, it just exposes all the metrics on an endpoint. That endpoint cant server all the data, and hence prometheus doesnt store the exact values, but how many requests were less than 0.1, 5, 15 …

- Update metrics/index.ts

```ts
import { NextFunction, Request, Response } from "express";
import { requestCounter } from "./requestCount";
import { activeRequestsGauge } from "./activeRequests";
import { httpRequestDurationMicroseconds } from "./requestTime";

export const metricsMiddleware = (req: Request, res: Response, next: NextFur
  const startTime = Date.now();
  activeRequestsGauge.inc();

  res.on('finish', function() {
    const endTime = Date.now();
    const duration = endTime - startTime;

    // Increment request counter
    requestCounter.inc({
      method: req.method,
      route: req.route ? req.route.path : req.path,
      status_code: res.statusCode
    });

    httpRequestDurationMicroseconds.observe({
      method: req.method,
      route: req.route ? req.route.path : req.path,
      code: res.statusCode
    }, duration);

    activeRequestsGauge.dec();
  });
  next();
}
```



It says

1. There were 0 to /user requests that were handled in less than 0.1ms
2. There were 0 to /user requests that were handled in less than 5ms
3. There were 0 to /user requests that were handled in less than 15ms
4. There were 0 to /user requests that were handled in less than 50ms
5. There were 0 to /user requests that were handled in less than 100ms
6. There were 0 to /user requests that were handled in less than 500ms
7. There were 0 to /user requests that were handled in less than 1000ms
8. There were 1 to /user requests that were handled in less than 3000ms
9. There were 1 to /user requests that were handled in less than 5000ms

As you can see, this is cumulative.
Number of requests being handled in less than 5000ms =
Number of requests being handled in less than 3000ms +
Number of requests that took between 3000-5000ms

WHAT IS SHARDING?
SHARDING MEANS DIVIDING THE DB IF THE LOAD HAS INCREASED

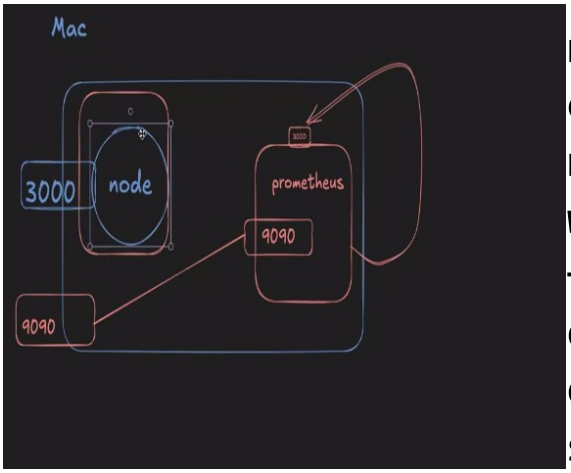
```
http_requests_total{method="GET",route="/",status_code="404"} 17
http_requests_total{method="GET",route="/metrics",status_code="200"} 6
http_requests_total{method="GET",route="/favicon.ico",status_code="404"} 1
http_requests_total{method="GET",route="/l;kzjdxfalsk;df",status_code="404"} 1
http_requests_total{method="GET",route="/l;kzjdxfalsk;dfask;dfjsl;kdf",status_code="404"} 1

# HELP active_requests Number of active requests
# TYPE active_requests gauge
active_requests 1

# HELP http_request_duration_ms Duration of HTTP requests in ms
# TYPE http_request_duration_ms histogram
http_request_duration_ms_bucket{le="0.1",method="GET",route="/",code="404"} 6
http_request_duration_ms_bucket{le="5",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="15",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="50",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="100",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="300",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="500",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="1000",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="3000",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="5000",method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="+Inf",method="GET",route="/",code="404"} 17
http_request_duration_ms_sum{method="GET",route="/",code="404"} 15
http_request_duration_ms_count{method="GET",route="/",code="404"} 17
http_request_duration_ms_bucket{le="0.1",method="GET",route="/metrics",code="200"} 2
http_request_duration_ms_bucket{le="5",method="GET",route="/metrics",code="200"} 6
http_request_duration_ms_bucket{le="15",method="GET",route="/metrics",code="200"} 6
http_request_duration_ms_bucket{le="50",method="GET",route="/metrics",code="200"} 6
http_request_duration_ms_bucket{le="100",method="GET",route="/metrics",code="200"} 6
http_request_duration_ms_bucket{le="300",method="GET",route="/metrics",code="200"} 6
http_request_duration_ms_bucket{le="500",method="GET",route="/metrics",code="200"} 6
```

the top one request is counter

the second one is gauge

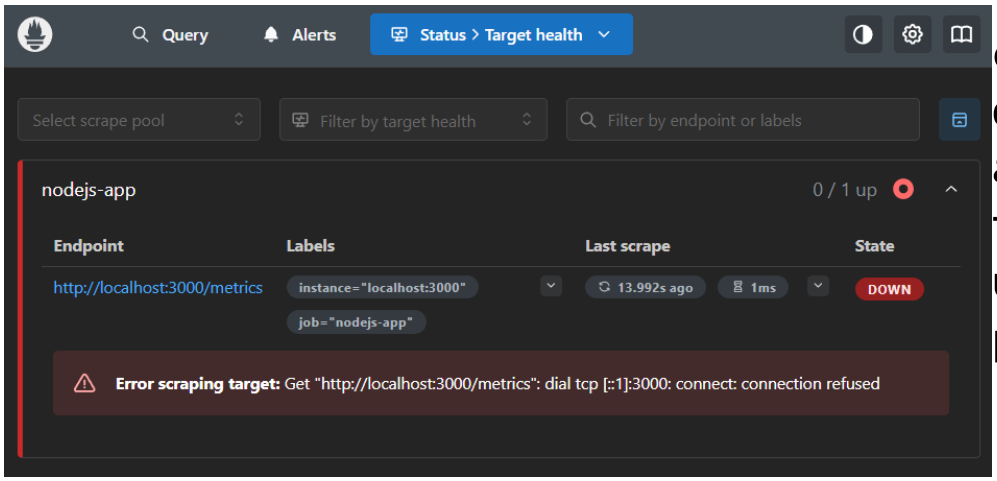
the below requests are the histogram data

NOW TO ACTUALLY START THE SYSTEM YOU HAVE TO CREATE A PROMETHEUS.YML FILE TO WORK WITH THE PROMETHEUS SYSTEM AND USE FOR SCRAPING THE DATA



now what we are doing is trying to connect the prometheus docker file to node js process but the issue is that whenever we are running the prome yml file it cannot talk to the 3000 port of the node js file which is running on our system to fix that we have to somehow containarize our nodejs app so that the prome yml file can talk to the node js application also and then we can connect them through a docker network

SO UNDERSTAND NOW THE NODE JS WHICH IS RUNNING ON OUR SYSTEM CAN TALK TO PROMETHEUS WHICH IS RUNNING IN DOCKER BUT OUR DOCKER CANNOT TALK TO THE NODE JS RUNNING IN THE SYSTEM NOT THE DOCKER

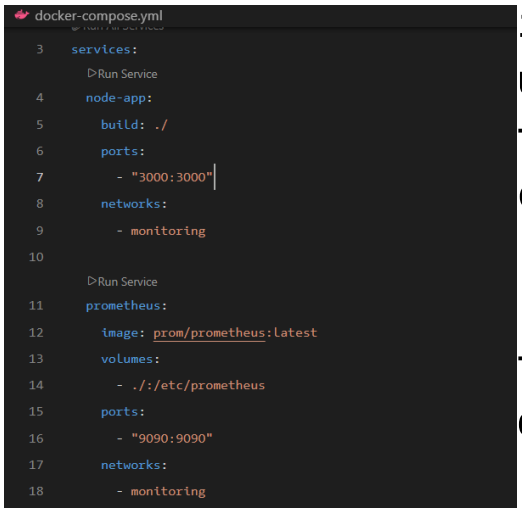


currently without dockerizing the node js application it shows this error as it is unable to use the 3000 port data

now to connect both the process first we have to create a docker file and then a docker network

docker network create prom_network
docker run -p 3000:3000 --network=prom_network node-app
then run the prometheus code

now this can be easily done by making a docker-compose file in which if we start these three containers it gets automatically connected together and can be used properly



in this the docker files are connected using the monitoring name network so that the docker file can explpore each other in the networks

to run the docker compose file use docker-compose up

just see how the queries in prom works but at the end we will use the grafana only

Queries in Prom
Simple queries (counters and gauges)

Here are some Prometheus queries you can run on localhost:9090 to analyze the metrics provided:

1. Total Number of HTTP Requests

To get the total number of HTTP requests per route

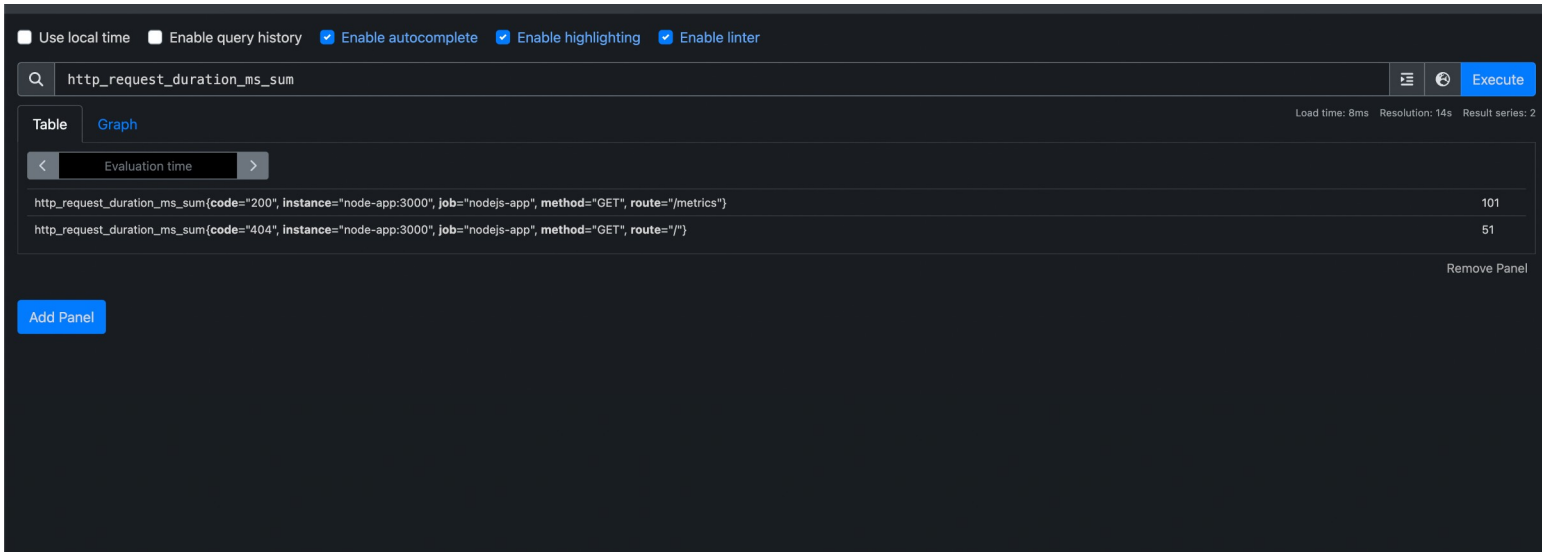
```
http_requests_total
```

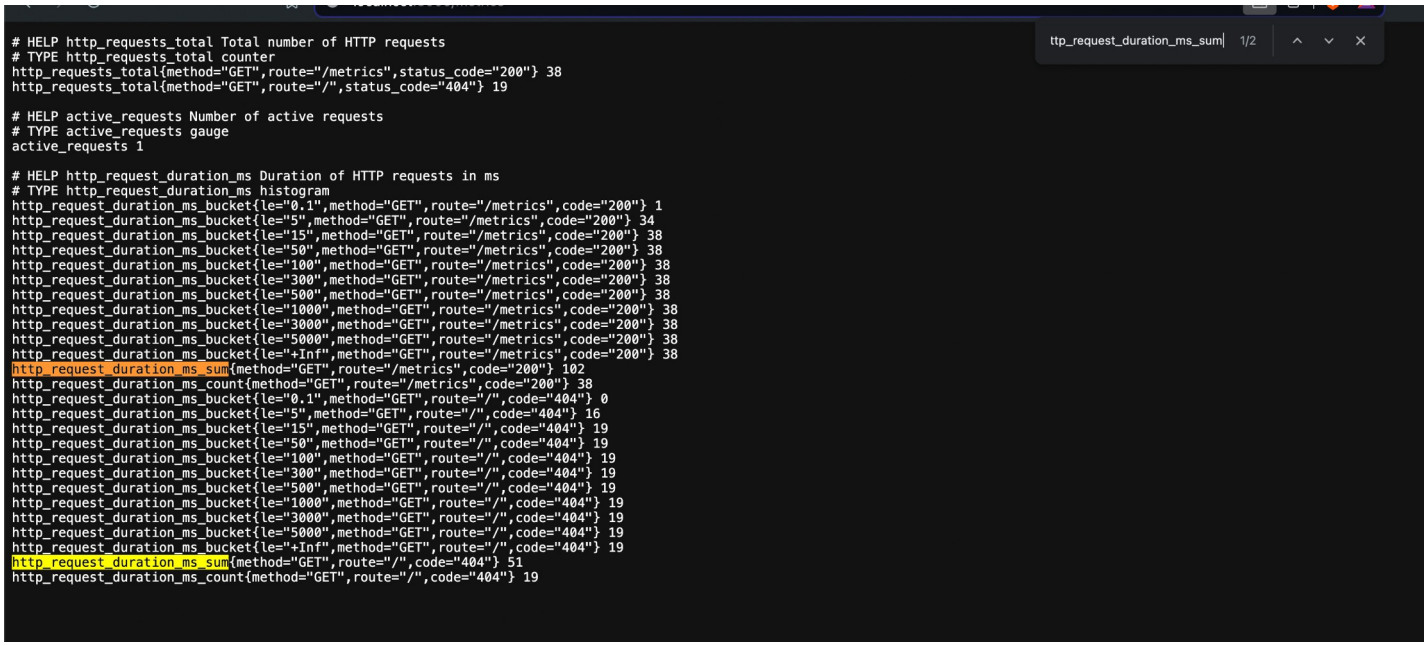
2. Total Number of HTTP Requests (cumulative)

```
sum(http_requests_total)
```

3. HTTP Request Duration

```
http_request_duration_ms_sum
```



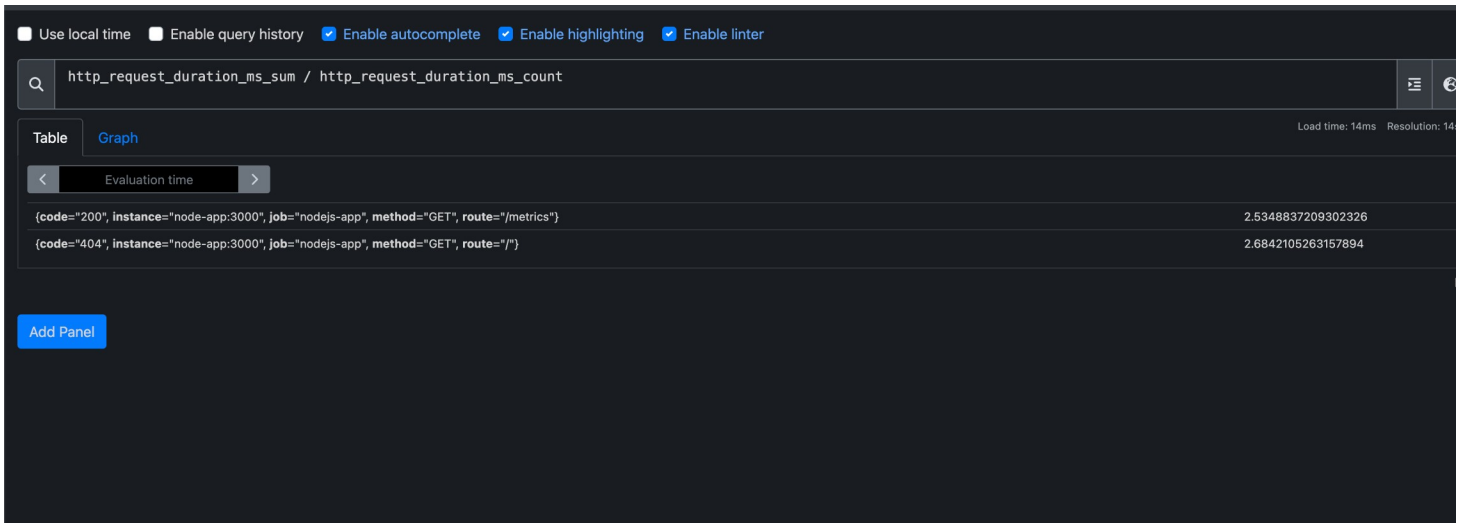


4. Count of total number of http requests

`http_request_duration_ms_count`

5. Average time it took to handle all requests

`http_request_duration_ms_sum / http_request_duration_ms_count`



Complex queries (histograms)

1. See the request duration in buckets

```
http_request_duration_ms_bucket
```

2. See requests for a specific route

```
http_request_duration_ms_bucket{method="GET", route="/metrics", code="200"}
```

Graphs in prom

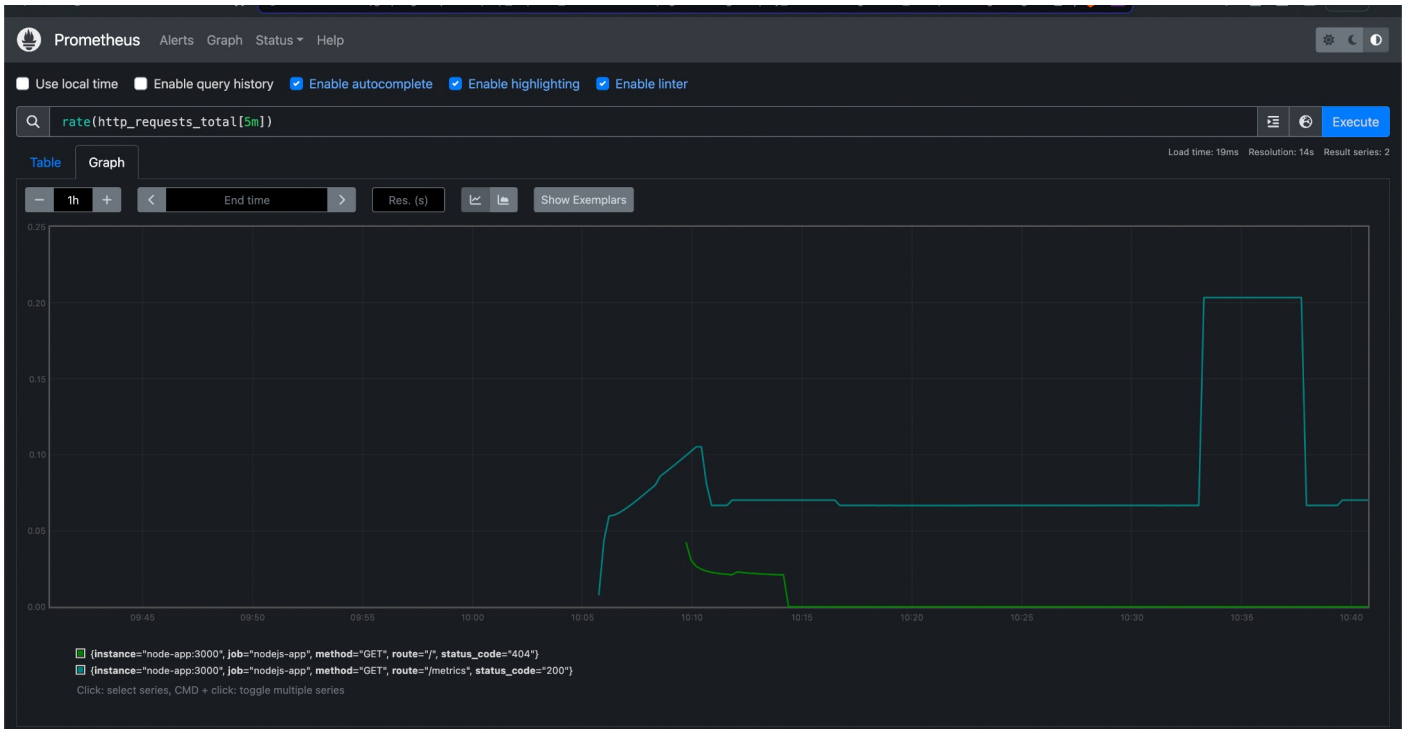
Prometheus also lets you visualise data as graphs

Lets see a few queries

1. Total number of requests

As you can tell, this is a very vague metric since it is cumulative. It is the total number of requests, but you usually want to see the rate at which requests are coming.

2. Rate of number of requests



3. Rate of all the requests (sum up /metrics and /user requests)

4. Average HTTP request duration with timeseries (5 minute buckets)

```
rate(http_request_duration_ms_sum[5m]) /
rate(http_request_duration_ms_count[5m])
```

DOWNSIDE OF USING PROMETHEUS UI IS THAT IT WORKS WITH ONLY PROMETHEUS BUT THE GRAFANNA WORKS WITH GOOGLE GOOGLE ANALYTICS,AMAZON DATASTORE AND ALSO PROMETHEUS

GRAFANA

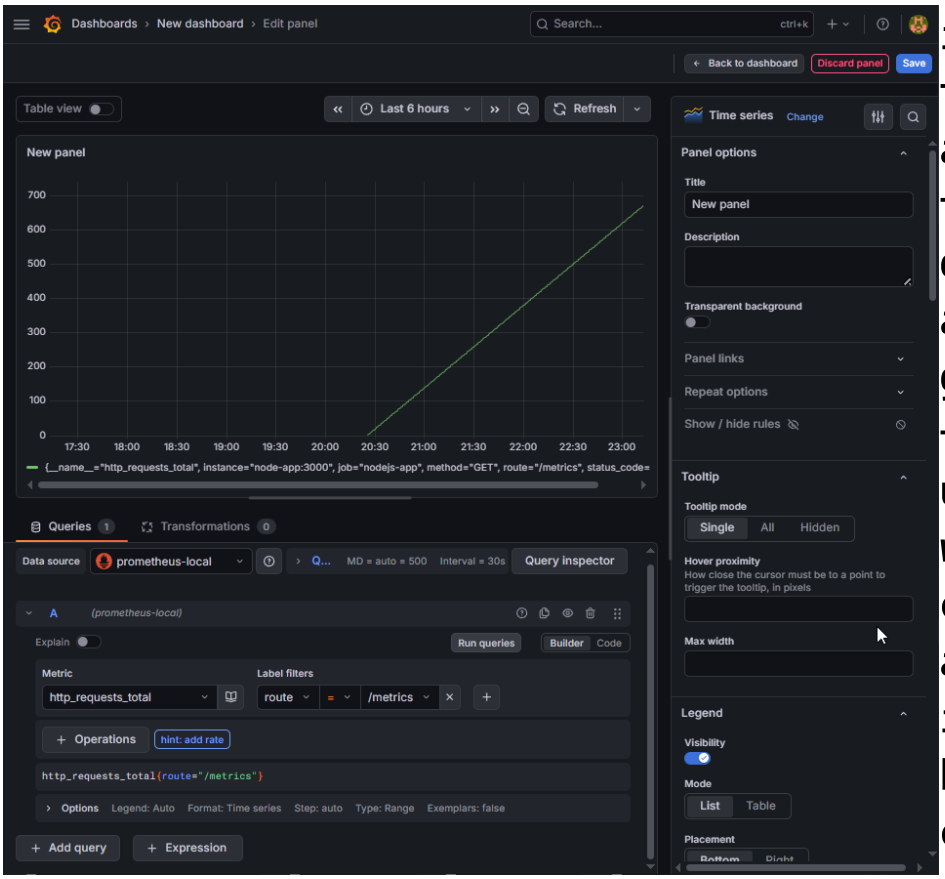
Even though you can use the prom interface, grafanna makes your life much easier
You can connect your prometheus data to grafana to be able to visualise your data better

Update docker-compose

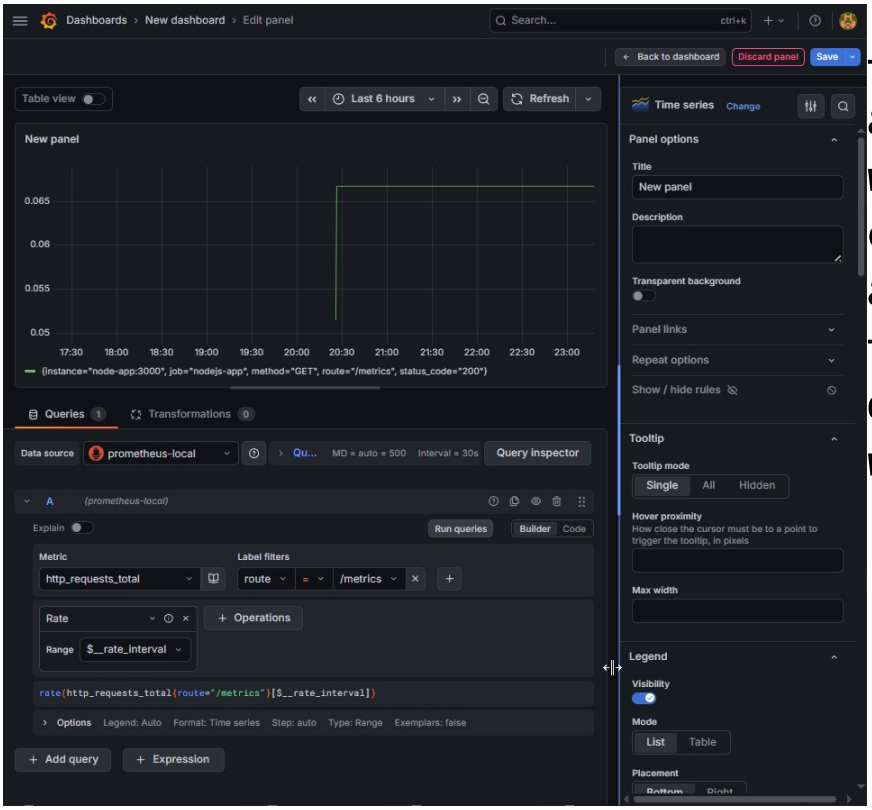
```
version: '3.8'
services:
  node-app:
    build: ./
    ports:
      - "3000:3000"
    networks:
      - monitoring
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./:/etc/prometheus
    ports:
      - "9090:9090"
    networks:
      - monitoring
  grafana:
    image: grafana/grafana:latest
    ports:
      - "3001:3000"
    networks:
      - monitoring
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
networks:
monitoring:
```

```
basic pass admin admin
```

so start the docker and then go to connection and add connection
add prometheus then add a new connection
then add promo url http://prometheus:9090



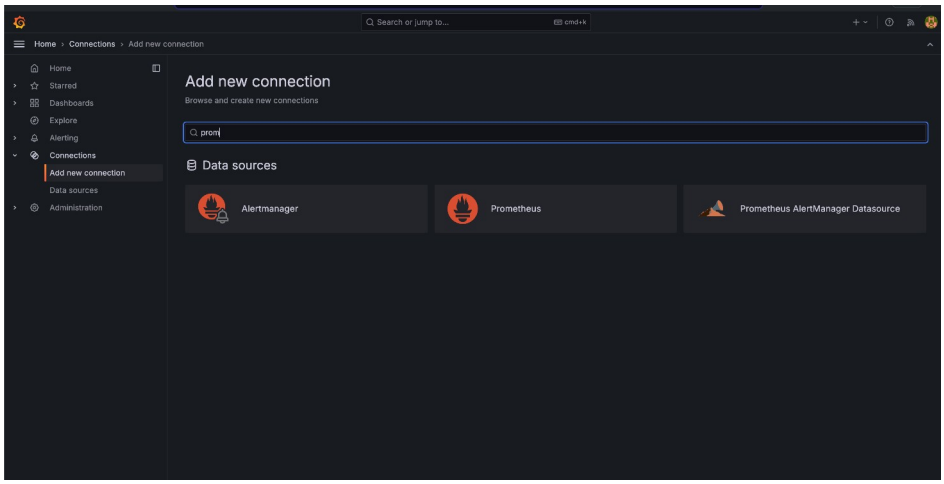
if you added header there than add here also
then build the dashboard
add then create one and go to all visualization for the graphs
using the time series we can use both query code or the ui also to add the code and all in this using the builder we can add the command and can see the graph how it looks



for this rate interval it also gives the code which we can use to run the code it gives a option to automatically make the time period based on the data to make the buckets we studied earlier

Adding prometheus as a source

- Create a connection



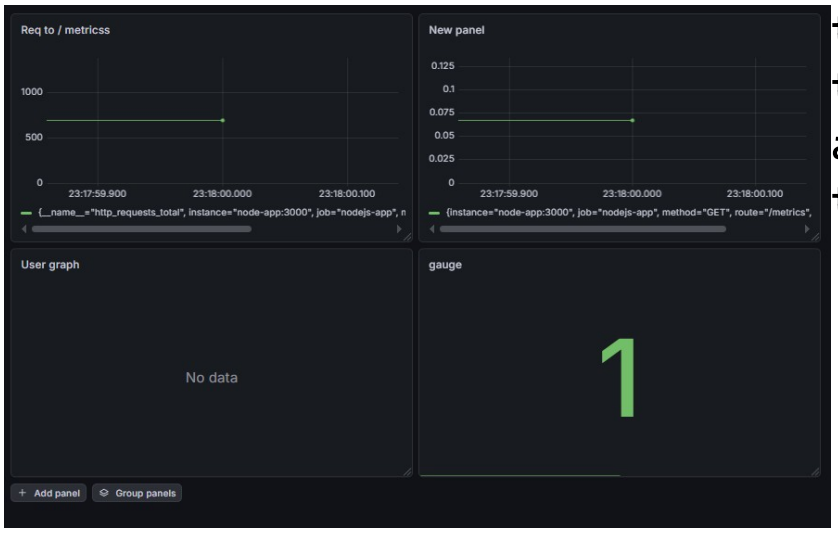
- Source URL

`http://prometheus:9090`

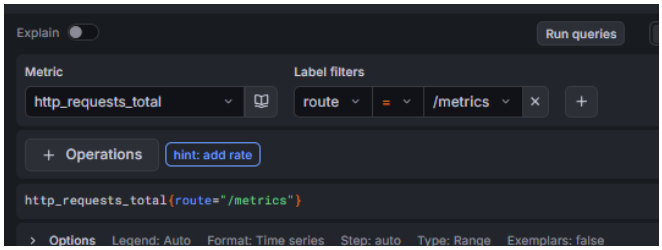
Assignment

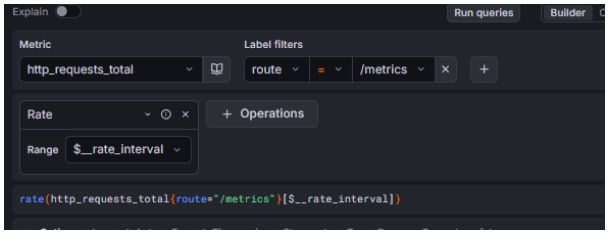
Try building a dashboard that has

1. Total number of requests to `/metrics` endpoint
2. Number of http requests to the `/metrics` endpoint per second
3. Total number of requests to the `/user` endpoint
4. Number of http request to the `/user` endpoint per second
5. A gauge that lets you see the current active requests

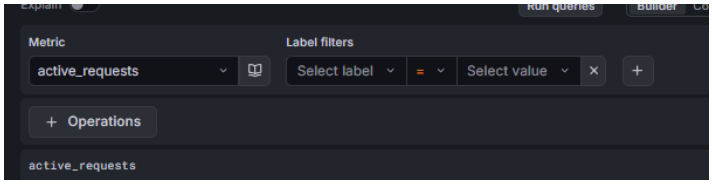


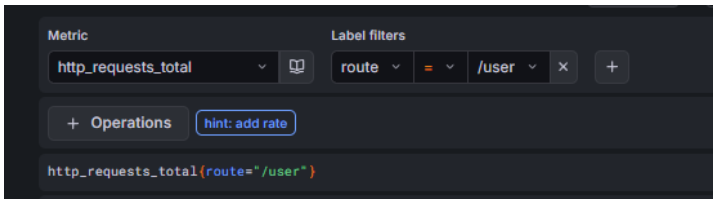
this is pretty easy just tak eth ename of the metric and the endpoint to create the graphs like this





for the gauge



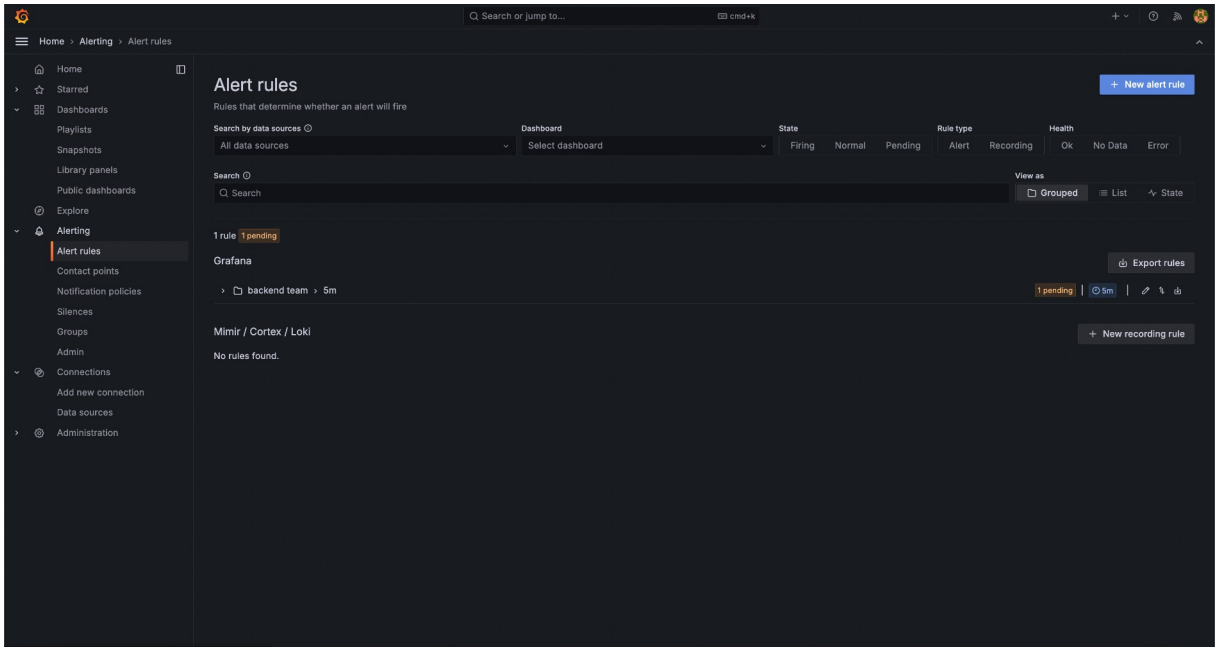


all are like this only

in the builder mode we can use the gui and and in the code mode we can add our own

Alerting

Grafana provides you with a way to set alerts on metrics.



Steps

1. Enter a name for it - High number of requests
2. Define query

```
rate(http_requests_total{route="/user"}[$__rate_interval])
```

1. Setup alert threshold (lets say 50 requests/s)
2. Set evaluation behaviour
    1. How often should we check this alert?
    2. Create folder so that it can be re-used later
3. Add labels
    1. Team: Backend
    2. Type: Error
4. Save

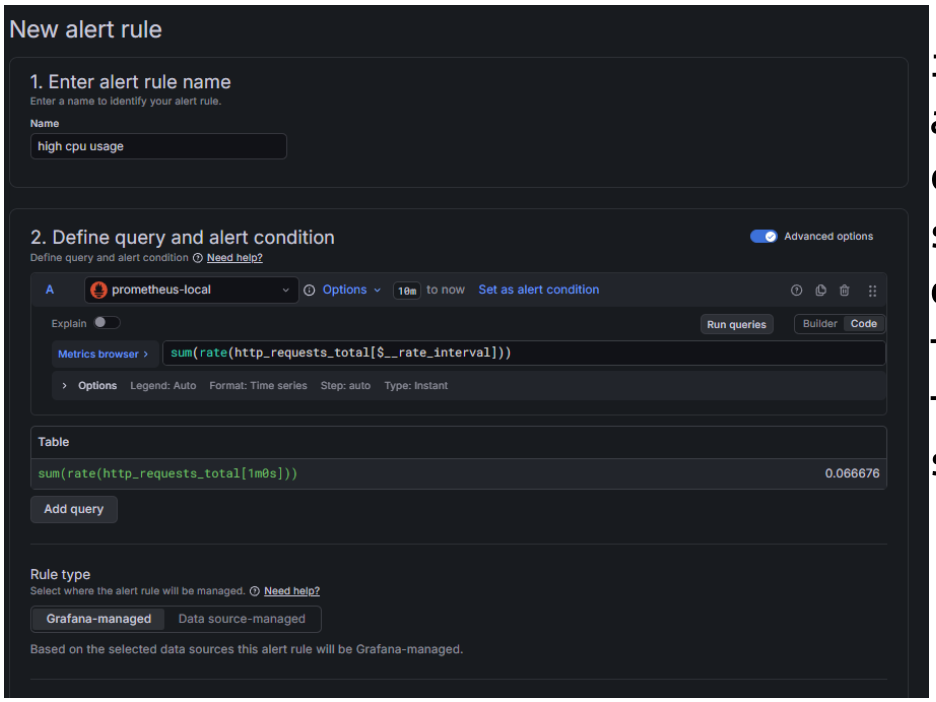
Testing

Send a lot of requests to the `/user` endpoint and ensure it triggers the alert

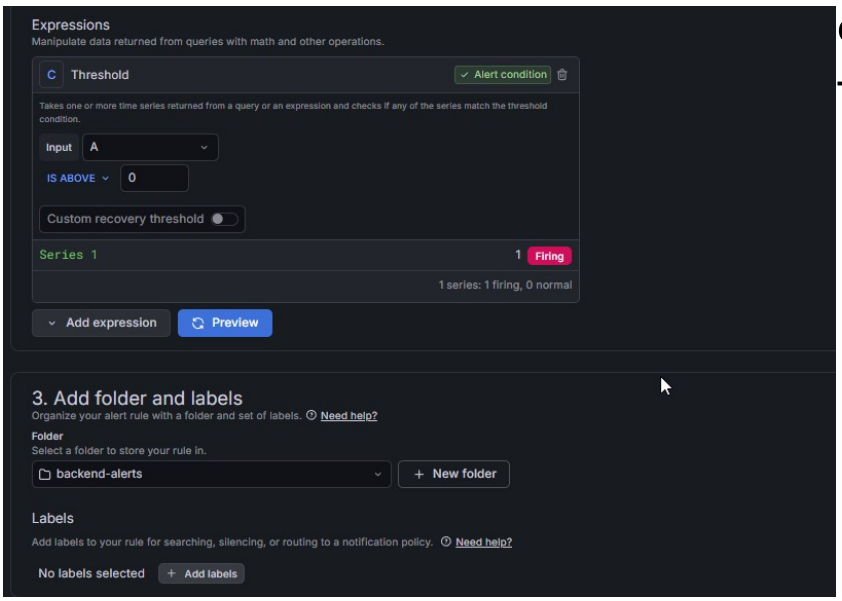
Notifying

1. Create a new contact point
2. Connect the alert to the contact point in Notification policies

This will not send a real email unless you've put in SMTP credentials while starting the apps



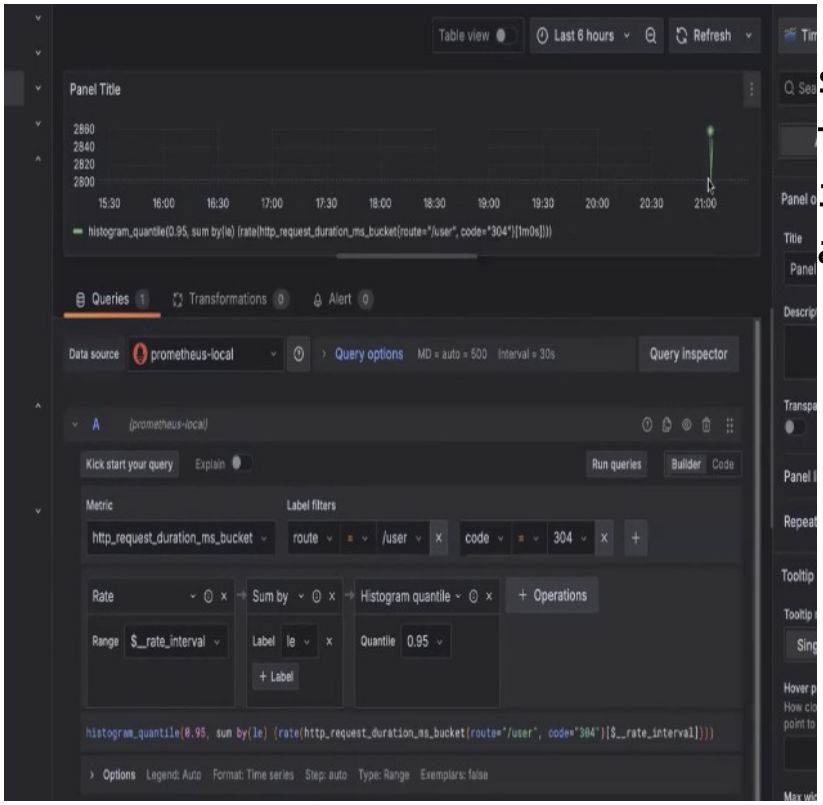
in this create the alert rule and then click on advanced settings to open the expression tab for threshold and also use the code mode for the sum and all



choose the threshold and the folder if you want to use

you can also add evaluation behaviour which can be used to send a notification to on call engineer for a certain period if there is a spike

configure notification we can send the notification wherever we want
you can also add many notification configuration for the emergency purposes
for things like P99 and P95 how can we visualize in the graffana



see the settings in this how they are configured
in this we can remove /user and use the /metrics also