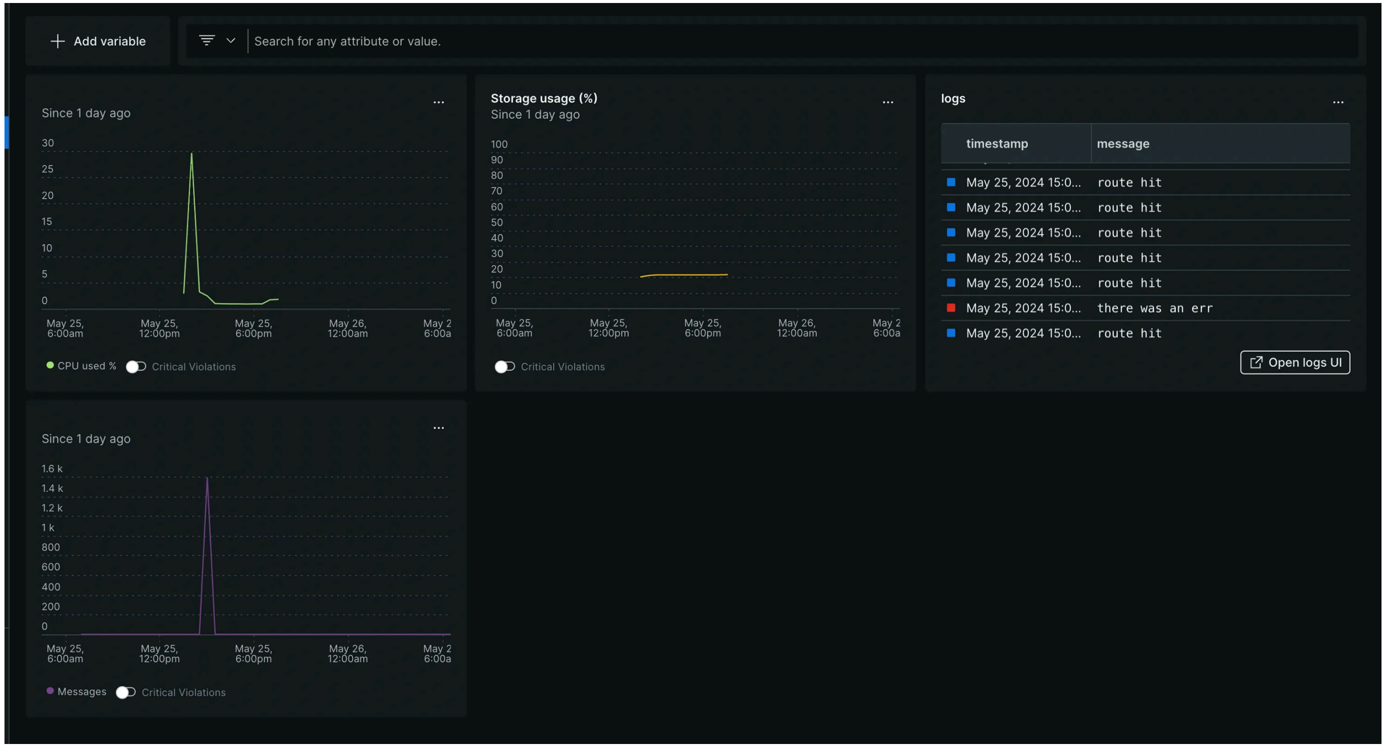
Revise monitoring

In the last live, we understood what is monitoring.

We created dashboards that looked like the following -



	timestamp	message
🟦	May 25, 2024 15:0...	route hit
🟦	May 25, 2024 15:0...	route hit
🟦	May 25, 2024 15:0...	route hit
🟦	May 25, 2024 15:0...	route hit
🟦	May 25, 2024 15:0...	route hit
🟥	May 25, 2024 15:0...	there was an err
🟦	May 25, 2024 15:0...	route hit

Problem with newrelic

1. It’s paid and can never be self hosted

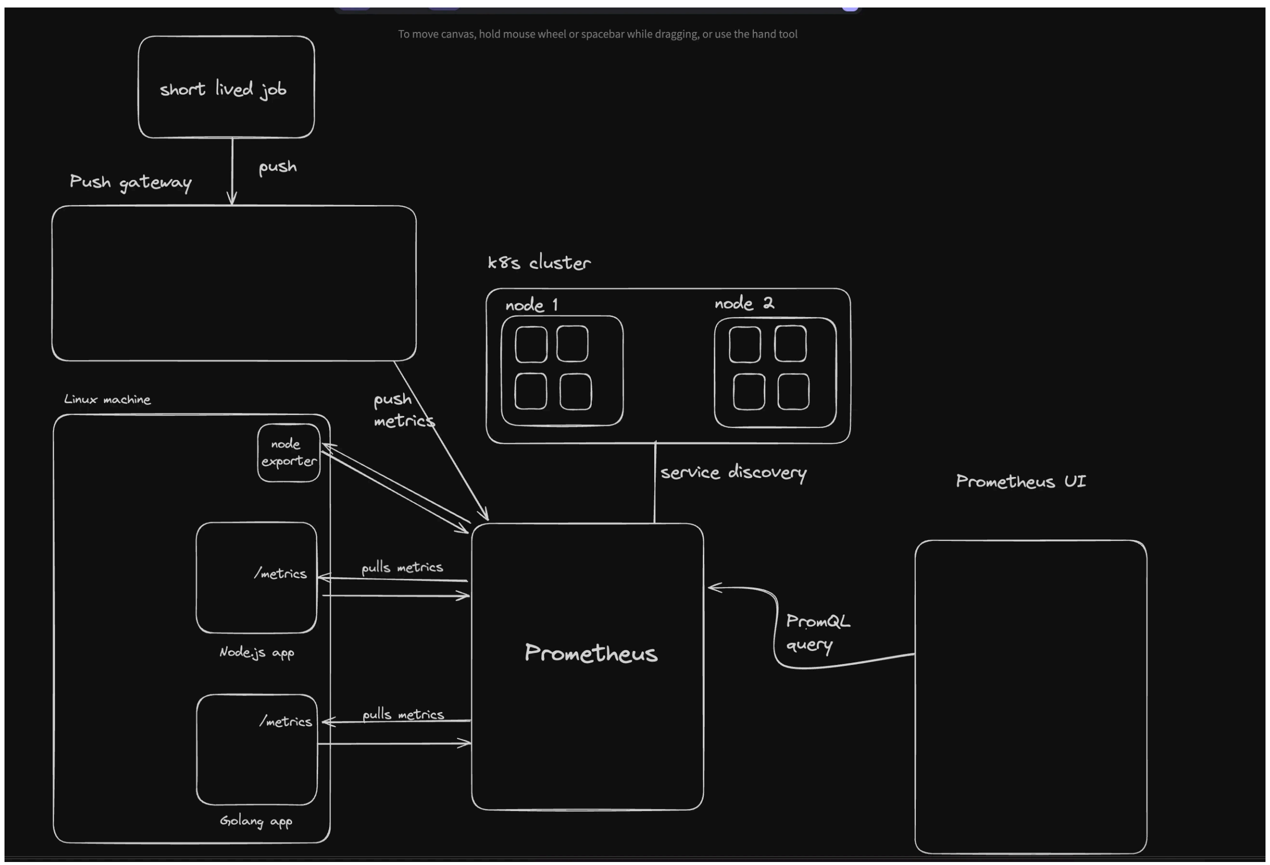
2. They own your data
3. Very hard to move away from it once it’s ingrained in your system

Prometheus

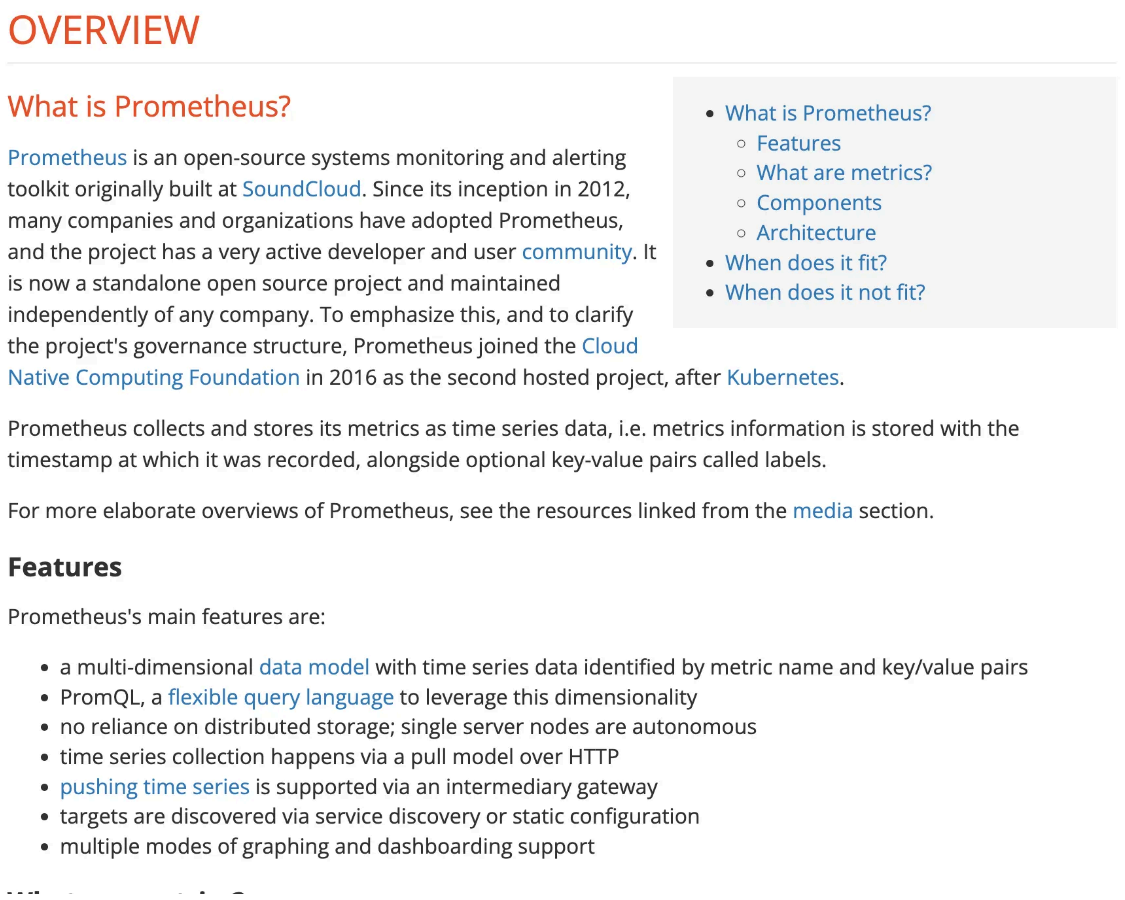
Prometheus architecture

Prometheus is a time series DB. It can monitor your

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



https://prometheus.io/docs/introduction/overview/



OVERVIEW

What is Prometheus?

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

1. Multi-dimensional data model with time series data identified by metric name and key/value pairs

Prometheus stores its data in a time series format where each data point consists of:

- **Metric Name:** A name that identifies the type of data, e.g., `http_requests_total` .
- **Labels (Key/Value Pairs):** Additional metadata that further identifies and differentiates the time series, e.g., `method="GET"` and `handler="/api"` . Labels provide a way to add dimensions to the metric data, allowing for flexible and detailed querying and analysis.

2. PromQL, a flexible query language to leverage this dimensionality

PromQL lets you query on top of all your timeseries data.

For example

```
sum(http_requests_total{job="api-server", status="500"})
```

would give you all the http requests that your server handled with status code 500

3. No reliance on distributed storage; single server nodes are autonomous

Prometheus is designed to be a standalone, single-node system that does not require external distributed storage solutions. Each Prometheus server node is autonomous, meaning it can independently scrape, store, and query time series data. This design simplifies the system architecture and operational overhead but also means that Prometheus is not inherently horizontally scalable. However sharding techniques can be used to manage larger deployments.

4. Time series collection happens via a pull model over HTTP

Prometheus primarily uses a **pull model** to collect metrics:

- **Pull Model:** Prometheus periodically scrapes metrics from configured targets by making HTTP requests to the `/metrics` endpoint exposed by the targets. This approach allows Prometheus to control the scraping intervals and retry logic.
- Targets expose their metrics in a specific format that Prometheus understands, typically using client libraries provided by Prometheus for various languages and environments.

5. Pushing time series is supported via an intermediary gateway

While Prometheus generally uses a pull model, it also supports a **push model** through the **Pushgateway**:

- **Pushgateway:** An intermediary service that allows applications and batch jobs to push metrics to it. The Pushgateway then exposes these metrics for Prometheus to scrape. This is useful for short-lived jobs or services that cannot be scraped reliably.

6. Targets are discovered via service discovery or static configuration

Prometheus supports multiple methods for discovering targets to scrape:

- **Service Discovery:** Dynamically discovers targets using various service discovery mechanisms like Kubernetes, Consul, AWS, etc. This allows Prometheus to automatically update its target list as the environment changes.
- **Static Configuration:** Manually specifies the list of targets in the configuration file. This is straightforward but less flexible compared to service discovery.

7. Multiple modes of graphing and dashboarding support

Prometheus offers several ways to visualize and interact with the collected metrics:

- **Prometheus UI:** A built-in web interface for ad-hoc queries and simple graphing.
- **Grafana:** A popular open-source dashboarding tool that integrates well with Prometheus, providing rich visualization and dashboarding capabilities.
- **Alertmanager:** A component of the Prometheus ecosystem used to handle alerts generated by Prometheus queries, allowing for complex alerting rules and notification integrations.

Adding raw metrics

Lets build an express app that exports metrics

Let’s add some hand made metrics to an express app

- Initialize a TS project

```
npm init -y
npx tsc --init
```

- Replace rootDir and outDir

```
"rootDir": "./src",
"outDir": "./dist",
```

- Add dependencies

```
npm install express @types/express
```

- Create `src/index.ts`

```
import express from "express";

const app = express();

app.use(express.json());

app.get("/user", (req, res) => {
  res.send({
    name: "John Doe",
    age: 25,
  });
});

app.post("/user", (req, res) => {
  const user = req.body;
  res.send({
    ...user,
    id: 1,
  });
});

app.listen(3000);
```

- Create a middleware that tracks the total time to handle a request (`middleware.ts`)

```
import { NextFunction, Request, Response } from "express";

export const middleware = (req: Request, res: Response, next: NextFunction) =
   const startTime = Date.now();
   next();
   const endTime = Date.now();
   console.log(`Request took ${endTime - startTime}ms`);
}
```

- Add the middleware globally

```
app.use(middleware);
```

- Update package.json to add scripts

```
"scripts": {
   "build": "tsc -b",
   "start": "npm run build && node dist/index.js"
},
```

- Run the application

```
npm run start
```

- Try to send a request and notice the logs

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

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.

- Example: Measuring the duration of HTTP requests.

Counters

Let’s add logic to count the number of requests (throughput) of our application.

- Install prom-client

```
npm install prom-client
```

- Create a new `metrics/requestCount.ts` file

```
import { NextFunction, Request, Response } from "express";
import client from "prom-client";

// Create a counter metric
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: N
  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();
};
```

- Add the middleware to `src/index.ts`
- Add a `/metrics` endpoint to `src/index.ts`

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

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

- Start the app

```
npm run start
```

Better structure

Before we proceed, lets aggregate all the metric creation and cleanup logic in the same file.

- Create a new middleware called `metrics/index.ts`

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

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

  res.on('finish', function() {
    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();
}
```

- Update the `metrics/requestCount.ts` to export the `requestCounter` and remove the cleanup logic from here

```
import { NextFunction, Request, Response } from "express";
import client from "prom-client";

// Create a counter metric
export const requestCounter = new client.Counter({
  name: 'http_requests_total',
  help: 'Total number of HTTP requests',
  labelNames: ['method', 'route', 'status_code']
});
```

- Update `src/index.ts` to use the `metricsMiddleware`

```
import { metricsMiddleware } from "./metrics";
app.use(metricsMiddleware);
```

Gauge

Lets add a gauge metric to our app

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

```
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`

```
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();
  });
}
```

- Add an artificial delay to the get endpoint

```
app.get("/user", async (req, res) => {
  await new Promise((resolve) => setTimeout(resolve, 1000));
  res.send({
    name: "John Doe",
    age: 25,
  });
});
```

- Hit the `/user` endpoint a few times
- Check the metrics

Histograms

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

- Add metrics/requestCount.ts

```
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 buc
});
```

💡 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

```
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: NextFun
  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();
}
```

- Go to the metrics endpoint

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

Final code

https://github.com/100xdevs-cohort-2/week-26-prom

With this code, you can run an application and see a bunch of metrics on it in a slightly ugly fashion on an endpoint

You can also try this on your side here - https://prom.100xdevs.com/metrics

Actually starting prometheus

Let’s start an actual prometheus process that scrapes the linux machine

Until now, we’ve exposed a `/metrics` endpoint but no one is `scraping` using it.

Prometheus actually scrapes (pulls) these metrics so you can visualise them over time (time series data)

For that, you need to start prometheus and give it the `source` of the metrics

- Add `prometheus.yml`

```
global:
  scrape_interval: 15s # How frequently to scrape targets

scrape_configs:
  - job_name: 'nodejs-app'
    static_configs:
      - targets: ['localhost:3000']
```

- Start prometheus locally

```
docker run -p 9090:9090 -v ./prometheus.yml:/etc/prometheus/prometheus
```

> 💡 You can start it w/o docker as well by installing it from source

- Try visiting `localhost:9090` , you will notice a problem in the `status/targets` section

> 💡 The problem is that nothing is running on port 3000 on the prom container, and hence it cant discover our service

Containerising the app

- Create a Dockerfile for the Node app

```
FROM node:20

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json ./

RUN npm install

# Bundle app source
COPY . .
```

```
EXPOSE 3000
CMD [ "node", "app.js" ]
```

- Create a docker-compose that starts the nodejs app as well as the prom container

```
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

networks:
 monitoring:
```

- Update prometheus.yml

```yaml
global:
  scrape_interval: 15s # How frequently to scrape targets

scrape_configs:
  - job_name: 'nodejs-app'
    static_configs:
      - targets: ['node-app:3000']
```

- Start docker compose

```
docker-compose up
```

- Try going to http://localhost:9090/

- Try executing a query

```
http_requests_total
```

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=
```

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_
```

Grafana

Ref - https://grafana.com/

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

Installing grafana in docker compose

- 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:
```

Try visiting `localhost:3001`

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

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