

**Laporan Proyek:  
Implementasi Sistem Monitoring Microservices Berbasis Docker Menggunakan  
Prometheus dan Grafana**

**Diajukan Untuk Memenuhi Tugas Mata Kuliah  
“*Workshop Administrasi Jaringan*”  
Dosen Pengampu: Dr. Ferry Astika Saputra ST, M.Sc.**



Disusun Oleh:

1. Agung Dwi Nugroho (3122600006)
2. Jordan Frisay Himawan (3122600007)
3. Sultan Argya Safa Firdaus (3122600022)

**D4 Teknik Informatika  
Departemen Teknik Informatika dan Komputer  
Politeknik Elektronika Negeri Surabaya**

# **BAB 1 PENDAHULUAN**

## **1.1 Latar Belakang**

Perkembangan teknologi informasi yang pesat telah mendorong banyak organisasi untuk mengadopsi arsitektur *microservices* dalam pengembangan aplikasi mereka. *Microservices* memungkinkan pengembangan dan penyebaran aplikasi secara modular, yang mendukung skalabilitas, fleksibilitas, dan pemeliharaan yang lebih mudah. Selain itu, penggunaan container seperti Docker semakin populer karena kemampuannya untuk menyediakan lingkungan yang konsisten dan terisolasi bagi aplikasi.

Namun, dengan meningkatnya kompleksitas sistem *microservices*, kebutuhan akan sistem monitoring yang efektif menjadi semakin penting. Monitoring yang baik dapat membantu mengidentifikasi masalah kinerja, menganalisis pemakaian sumber daya, dan memastikan bahwa semua layanan berjalan dengan optimal. Dalam konteks ini, Prometheus dan Grafana muncul sebagai solusi monitoring yang andal dan efisien.

Prometheus adalah sistem monitoring dan peringatan yang dirancang khusus untuk aplikasi berbasis *cloud-native*, sementara Grafana adalah platform analisis dan visualisasi yang memungkinkan pengguna untuk membuat dashboard interaktif dari data yang dikumpulkan oleh Prometheus. Kombinasi keduanya memberikan alat yang kuat untuk memonitor aplikasi yang berjalan di atas Docker.

Proyek akhir ini bertujuan untuk mengimplementasikan sistem monitoring *microservices* berbasis Docker menggunakan Prometheus dan Grafana. Selain itu, proyek ini juga akan membandingkan performa *web server services* dengan *serverless services* yang berjalan di atas Docker.

## **1.2 Rumusan Masalah**

Berdasarkan latar belakang yang telah dijelaskan sebelumnya, adapun rumusan masalah yang akan dibahas dalam proyek ini adalah sebagai berikut:

1. Bagaimana cara mengimplementasikan sistem monitoring untuk *microservices* berbasis Docker menggunakan Prometheus dan Grafana?
2. Bagaimana cara mengumpulkan dan menganalisis metrik yang relevan, seperti penggunaan sumber daya dan jumlah *traffic* pada setiap kontainer?
3. Bagaimana performa *web server services* dibandingkan dengan *serverless services* yang berjalan di atas Docker?

## 1.3 Tujuan Proyek

Proyek ini memiliki beberapa tujuan utama, yaitu:

1. Mengimplementasikan sistem monitoring untuk microservices berbasis Docker menggunakan Prometheus dan Grafana.
2. Mengumpulkan dan menganalisis metrik yang meliputi penggunaan sumber daya dan jumlah traffic pada masing-masing kontainer.
3. Melakukan analisa perbandingan performa antara web server services dan serverless services yang berjalan di atas Docker.

## 1.4 Manfaat Proyek

Adapun manfaat yang diharapkan dari proyek ini antara lain:

1. Menyediakan solusi monitoring yang komprehensif untuk aplikasi berbasis microservices di lingkungan Docker.
2. Membantu organisasi dalam mengidentifikasi dan menyelesaikan masalah kinerja aplikasi dengan efisien.
3. Memberikan wawasan tentang perbandingan performa antara web server services dan serverless services, yang dapat digunakan sebagai referensi dalam pengambilan keputusan arsitektur aplikasi.

## 1.5 Metodologi

Metodologi yang digunakan dalam proyek ini meliputi beberapa tahapan, yaitu:

1. **Studi Literatur:** Mengkaji berbagai referensi utamanya pada dokumentasi resmi mengenai monitoring microservices, Docker, Prometheus, dan Grafana.
2. **Perancangan Sistem:** Merancang arsitektur sistem monitoring yang akan diimplementasikan.
3. **Implementasi:** Membangun dan mengkonfigurasi sistem monitoring menggunakan Docker, Prometheus, dan Grafana.
4. **Pengumpulan Data:** Mengumpulkan metrik dari aplikasi yang berjalan dan menyimpan data tersebut di Prometheus.
5. **Analisis:** Menganalisis data yang telah dikumpulkan untuk mendapatkan wawasan tentang kinerja aplikasi.
6. **Perbandingan Performa:** Melakukan perbandingan antara web server services dan serverless services berdasarkan data yang telah dianalisis.
7. **Penyusunan Laporan:** Menyusun laporan akhir yang mendokumentasikan seluruh proses dan hasil dari proyek ini.

## **BAB 2 RUANG LINGKUP**

Proyek ini memiliki ruang lingkup yang mencakup berbagai aspek dari implementasi sistem monitoring microservices berbasis Docker menggunakan Prometheus dan Grafana. Berikut adalah rincian ruang lingkup dari proyek ini:

### **1. Monitoring Penggunaan Memori:**

- Melacak penggunaan memori oleh setiap kontainer Docker.
- Mengidentifikasi kebocoran memori atau penggunaan memori yang tidak efisien.
- Membuat dasbor untuk memvisualisasikan penggunaan memori secara real-time.

### **2. Monitoring Virtual Memori:**

- Mengukur dan memonitor penggunaan semua memori yang dialokasikan oleh proses.
- Menganalisis distribusi alokasi proses untuk mengidentifikasi kontainer yang bekerja terlalu keras atau terlalu sedikit.

### **3. Monitoring Request Rate:**

- Memonitor jumlah request yang masuk dan keluar dari setiap kontainer service Docker.
- Mengidentifikasi potensi bottleneck atau masalah jaringan.
- Membuat visualisasi request rate untuk membantu dalam perencanaan kapasitas dan pengambilan keputusan.

### **4. Pengumpulan dan Analisis Metrik:**

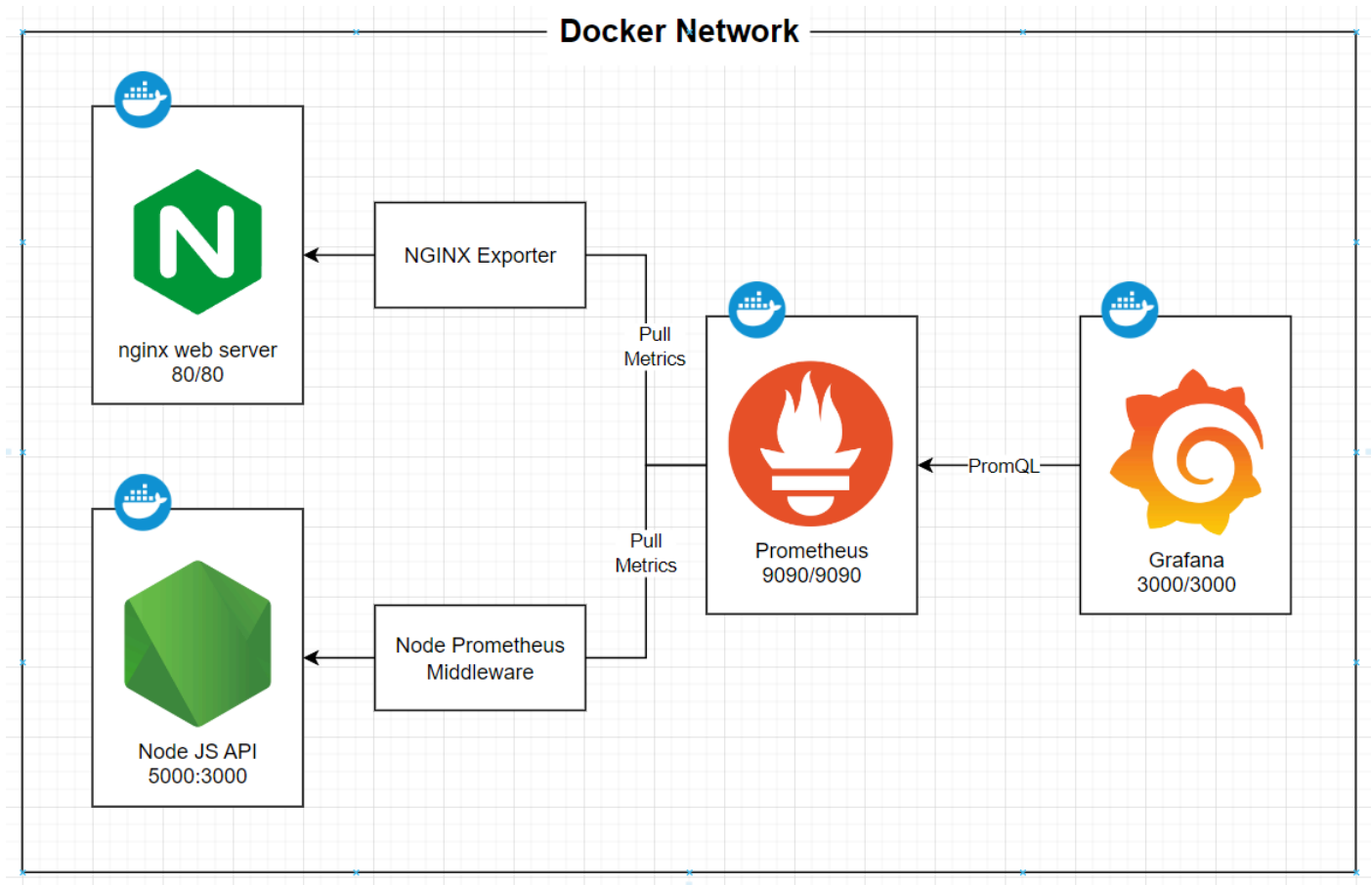
- Menggunakan Prometheus untuk mengumpulkan metrik dari semua sumber daya yang dimonitor.
- Menggunakan Grafana untuk membuat dasbor yang interaktif dan informatif berdasarkan data metrik yang dikumpulkan.

### **5. Perbandingan Performa:**

- Membandingkan performa web server services dengan serverless services yang berjalan di atas Docker.
- Menganalisis perbedaan dalam penggunaan sumber daya, traffic, dan kinerja secara keseluruhan.
- Menyediakan wawasan yang dapat digunakan untuk pengambilan keputusan dalam desain arsitektur aplikasi.

Dengan ruang lingkup yang jelas ini, diharapkan proyek ini dapat memberikan pandangan yang komprehensif mengenai performa dan penggunaan sumber daya dari layanan berbasis microservices di atas Docker. Implementasi sistem monitoring yang baik akan membantu dalam memastikan bahwa aplikasi berjalan secara optimal dan efisien.

## BAB 3 DESAIN ARSITEKTUR SISTEM



**Gambar 1. Rancangan Arsitektur Sistem Monitoring**

Berdasarkan **Gambar 1**, sistem monitoring dan services akan berjalan pada satu jaringan Docker. Dimana masing-masing dari services dan monitoring tool berjalan pada kontainer terpisah.

Pada proyek ini akan menggunakan dua services yang akan dimonitoring, yakni API PHP yang akan berjalan di nginx web server dan aplikasi Express yang berjalan di Node JS API sebagai serverless service.

Prometheus akan melakukan pull metrics atau monitoring metrics seperti CPU, memori, dan traffic pada masing-masing services melalui Exporter dan middleware yang tersedia berdasarkan servicenya. Selanjutnya, Grafana melalui PromQL akan mengambil data dari Prometheus dan memvisualisasikan dalam bentuk dashboard serta dilakukan analisis perbandingan performa kedua service yang dimonitoring.

## BAB 4 WAKTU PELAKSANAAN

Backlog	27/05/2024	28/05/2024	29/05/2024	30/05/2024	31/05/2024	01/06/2024	02/06/2024	03/06/2024
Studi Literatur Grafana dan Prometheus								
Pengerjaan <i>Project Charter</i>								
Rancangan Arsitektur Sistem								
Setup Container dan Network Pada Docker								
Integrasi Prometheus, Grafana, dan Postgre SQL Pada Services								
Testing dan Perbaikan Bug								
Pengerjaan Laporan Akhir								

Secara keseluruhan, pelaksanaan proyek ini dikerjakan dalam rentang waktu satu pekan. Dimulai pada hari Senin, 27 Mei 2024 hingga Senin, 3 Juni 2024. Pada hari pertama tim berfokus pada studi literatur merujuk dokumentasi resmi dari Prometheus dan Grafana. Selain itu juga dilakukan pengerjaan kesepahaman proyek atau *project charter*. Rancangan desain arsitektur sistem juga dilakukan pada hari pertama dimulainya pengerjaan proyek.

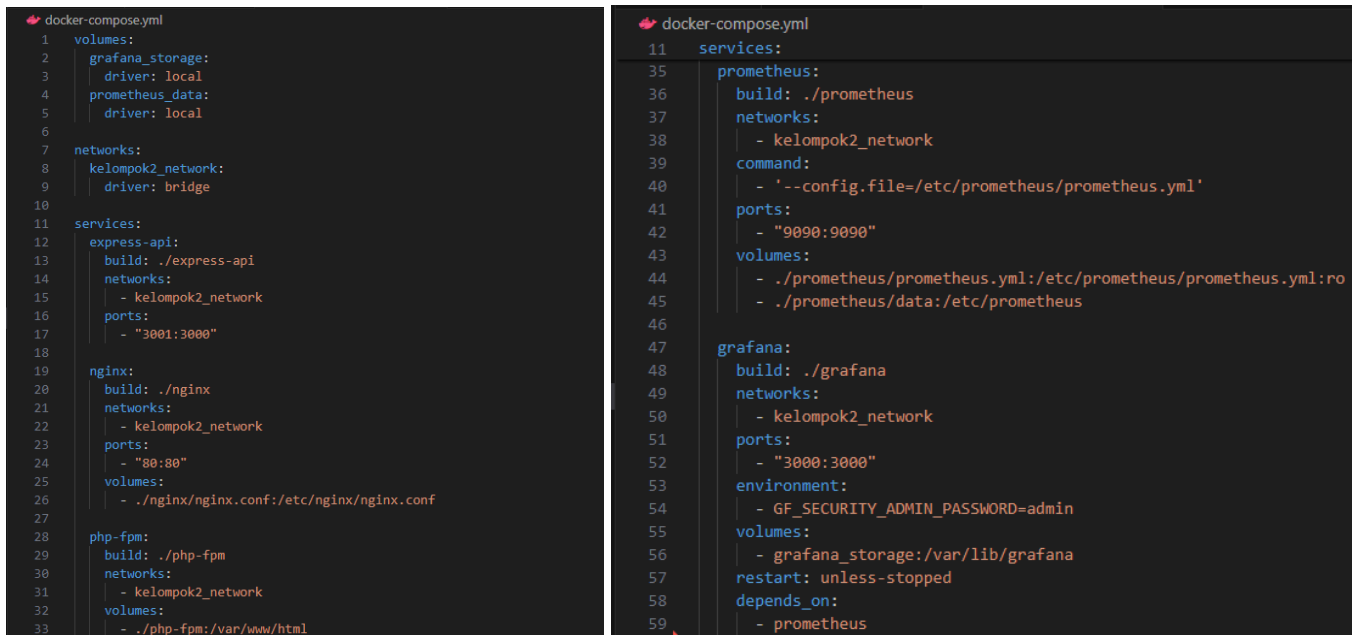
Memasuki hari kedua hingga hari ketujuh yang dimulai pada hari Selasa, 28 Mei 2024 hingga Minggu, 2 Juni 2024, tahap implementasi dilakukan oleh tim. Tahapan ini meliputi setup dan konfigurasi docker serta integrasi tools monitoring, database, dan microservice yang akan dimonitoring.

Uji coba dilakukan pada 2 hari terakhir dari pengerjaan proyek, yakni pada hari Minggu, 2 Juni 2024 hingga hari Senin, 3 Juni 2024. Pengujian dilakukan untuk mengambil data dari metrik yang akan dianalisis dan perbaikan bug atau error yang terjadi pada sistem selama pengerjaan proyek.

Tahap terakhir adalah pengerjaan laporan yang dilakukan mulai pada hari Rabu, 29 Mei 2024 hingga Senin, 3 Juni 2024. Laporan akhir dikerjakan secara paralel mengikuti tahapan konfigurasi dan integrasi monitoring tool terhadap microservices.

# BAB 5 IMPLEMENTASI

## 5.1 Konfigurasi Docker



**Gambar 2. docker-compose.yml**

File konfigurasi docker-compose.yml digunakan untuk mengelola dan mengorquestrasi beberapa layanan yang dibutuhkan oleh aplikasi kami. Selain itu, file konfigurasi ini dibutuhkan supaya setiap service dapat berkomunikasi satu sama lain dalam lingkungan yang terisolasi namun juga terintegrasi. Berikut adalah penjelasan lebih rinci mengenai setiap bagian dari file tersebut:

### 1. Volumes:

- **grafana\_storage:** Volume ini menggunakan driver lokal dan digunakan untuk menyimpan data Grafana secara persisten.
- **prometheus\_data:** Volume ini juga menggunakan driver lokal dan digunakan untuk penyimpanan data Prometheus.

### 2. Networks:

- **kelompok2\_network:** Kami mendefinisikan sebuah jaringan dengan driver bridge bernama kelompok2\_network yang digunakan oleh semua layanan untuk berkomunikasi satu sama lain.

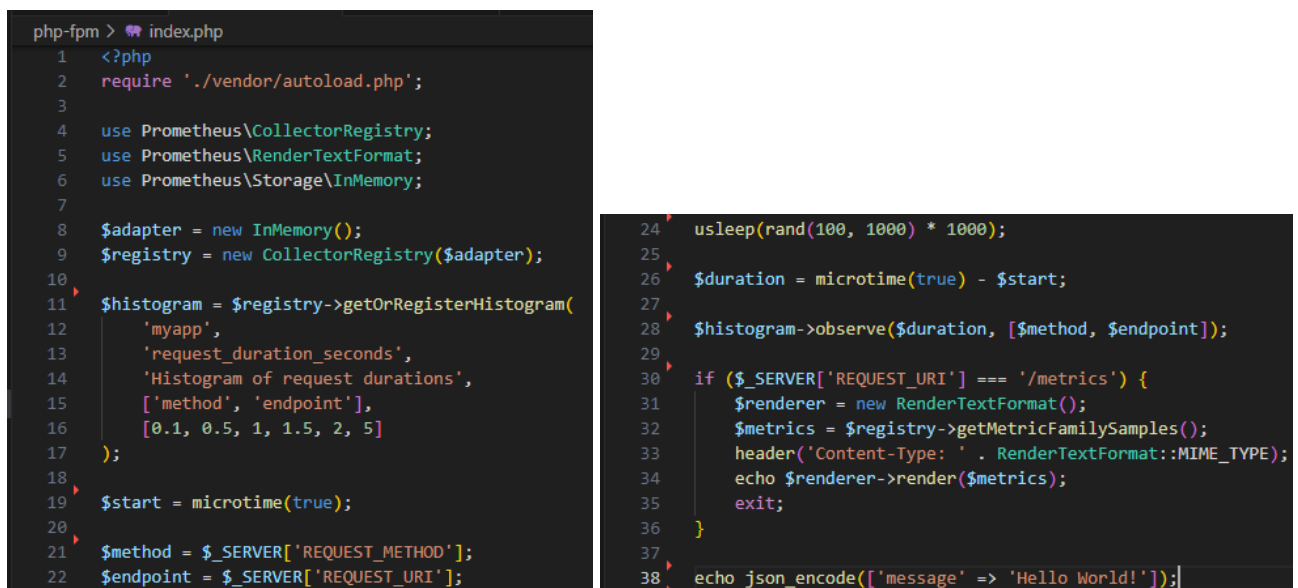
### 3. Services:

- **express-api:** Layanan ini membangun API menggunakan Express.js dari direktori ./express-api. Layanan ini terhubung ke jaringan kelompok2\_network dan map port 3001 di host ke port 3000 di container.
- **nginx:** Layanan ini membangun Nginx dari direktori ./nginx. Layanan ini juga terhubung ke jaringan kelompok2\_network dan memetakan port 80 di host ke port 80 di container. Selain itu, file konfigurasi Nginx di-mount dari ./nginx/nginx.conf ke dalam container.

- **php-fpm:** Layanan ini membangun PHP-FPM dari direktori ./php-fpm. Layanan ini terhubung ke jaringan kelompok2\_network dan file di ./php-fpm di-mount ke direktori /var/www/html di dalam container.
- **prometheus:** Layanan ini membangun Prometheus dari direktori ./prometheus. Layanan ini terhubung ke jaringan kelompok2\_network dan menggunakan file konfigurasi prometheus.yml yang di-mount dari ./prometheus/prometheus.yml dengan mode read-only. Port 9090 di host dipetakan ke port 9090 di container, dan direktori ./prometheus/data di-mount ke direktori data Prometheus di dalam container.
- **grafana:** Layanan ini membangun Grafana dari direktori ./grafana. Layanan ini terhubung ke jaringan kelompok2\_network dan memetakan port 3000 di host ke port 3000 di container. Environment variable GF\_SECURITY\_ADMIN\_PASSWORD diatur dengan nilai admin. Volume grafana\_storage di-mount ke direktori /var/lib/grafana di dalam container untuk penyimpanan data Grafana yang persisten. Layanan ini diatur untuk restart secara otomatis kecuali dihentikan secara manual, dan memiliki ketergantungan pada layanan prometheus.

## 5.2 Mempersiapkan Microservices

### 1. PHP-FPM NGINX Service:



```

php-fpm > index.php
1  <?php
2  require './vendor/autoload.php';
3
4  use Prometheus\CollectorRegistry;
5  use Prometheus\RenderTextFormat;
6  use Prometheus\Storage\InMemory;
7
8  $adapter = new InMemory();
9  $registry = new CollectorRegistry($adapter);
10
11 $histogram = $registry->getOrRegisterHistogram(
12     'myapp',
13     'request_duration_seconds',
14     'Histogram of request durations',
15     ['method', 'endpoint'],
16     [0.1, 0.5, 1, 1.5, 2, 5]
17 );
18
19 $start = microtime(true);
20
21 $method = $_SERVER['REQUEST_METHOD'];
22 $endpoint = $_SERVER['REQUEST_URI'];
23
24 usleep(rand(100, 1000) * 1000);
25
26 $duration = microtime(true) - $start;
27
28 $histogram->observe($duration, [$method, $endpoint]);
29
30 if ($_SERVER['REQUEST_URI'] === '/metrics') {
31     $renderer = new RenderTextFormat();
32     $metrics = $registry->getMetricFamilySamples();
33     header('Content-Type: ' . RenderTextFormat::MIME_TYPE);
34     echo $renderer->render($metrics);
35     exit;
36 }
37
38 echo json_encode(['message' => 'Hello World!']);

```

**Gambar 3. Web Server Service**

Untuk services yang berjalan diatas web server, kami menggunakan API PHP yang dikonfigurasi untuk mengumpulkan metrik dengan bantuan library Prometheus. Berikut adalah penjelasan rinci mengenai kode yang digunakan:

1. **Inisialisasi Library:** Memuat pustaka Prometheus dan menginisialisasi adapter penyimpanan InMemory.
2. **Deklarasi Histogram:** Mendefinisikan histogram untuk mengukur durasi permintaan dengan label method dan endpoint.



3. **Pengukuran Durasi:** Mengukur durasi permintaan mulai dari awal hingga akhir, mensimulasikan waktu pemrosesan dengan `usleep`.
4. **Perekaman Metrik:** Merekam durasi permintaan dalam histogram bersama labelnya.
5. **Endpoint /metrics:** Jika permintaan menuju `/metrics`, metrik yang dikumpulkan ditampilkan dalam format yang bisa dibaca oleh Prometheus.
6. **Respons Default:** Untuk endpoint lainnya, aplikasi mengembalikan respons JSON dengan pesan Hello World!.

## 2. Node Express Service:

```
express-api > JS app.js
1  const express = require('express');
2  const promMid = require('express-prometheus-middleware');
3
4  const app = express();
5
6  app.use(promMid({
7    metricsPath: '/metrics',
8    collectDefaultMetrics: true,
9    requestDurationBuckets: [0.1, 0.5, 1, 1.5],
10 }));
11
12 app.get('/', (req, res) => {
13   res.send('Hello World!');
14 });
15
16 app.listen(3000, () => {
17   console.log('Express app listening on port 3000!');
18 });
```

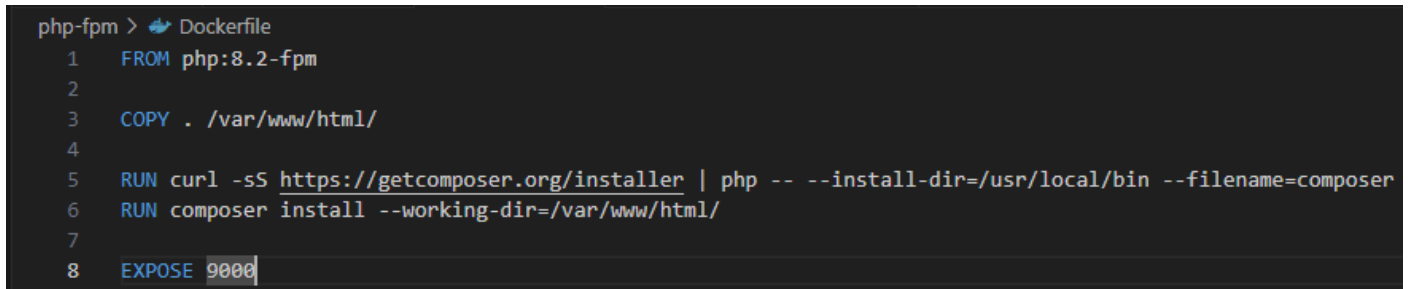
**Gambar 4. Node JS Service**

Untuk services Node JS, kami menggunakan aplikasi Express yang dikonfigurasi untuk mengumpulkan metrik dengan menggunakan middleware Prometheus. Berikut adalah penjelasan rinci mengenai kode yang digunakan:

1. **Inisialisasi Aplikasi:** Menggunakan Express untuk membuat aplikasi web.
2. **Middleware Prometheus:** Menambahkan middleware Prometheus yang mengumpulkan metrik kinerja dan menempatkannya di endpoint `/metrics`. Metrik default dikumpulkan, dan bucket durasi permintaan diatur pada 0.1, 0.5, 1, dan 1.5 detik.
3. **Endpoint Root:** Mendefinisikan endpoint root (`/`) yang mengembalikan pesan Hello World!
4. **Menjalankan Server:** Menjalankan server pada port 3000 dan mencetak pesan ke konsol saat server sudah berjalan.

## 5.3 Konfigurasi Dockerfile Microservices

### 1. Web Service Dockerfile



```
php-fpm > Dockerfile
1 FROM php:8.2-fpm
2
3 COPY . /var/www/html/
4
5 RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer
6 RUN composer install --working-dir=/var/www/html/
7
8 EXPOSE 9000
```

**Gambar 5. Web Service Dockerfile**

Dockerfile ini digunakan untuk membangun image Docker bagi layanan API PHP dan berfungsi untuk memastikan bahwa semua dependensi PHP diinstal dan layanan PHP-FPM siap menerima permintaan pada port 9000. Berikut adalah penjelasan konfigurasinya:

#### 1. Base Image:

FROM php:8.2-fpm: Menggunakan image dasar PHP versi 8.2 dengan FPM (FastCGI Process Manager) yang cocok untuk menjalankan aplikasi PHP dalam lingkungan produksi.

#### 2. Menyalin Source Code:

COPY . /var/www/html/: Menyalin semua file dari direktori proyek ke dalam direktori /var/www/html/ di dalam container.

#### 3. Instalasi Composer:

- RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer: Mengunduh dan menginstal Composer, alat manajemen dependensi untuk PHP, ke dalam container.
- RUN composer install --working-dir=/var/www/html/: Menjalankan perintah composer install untuk menginstal semua dependensi yang didefinisikan dalam file composer.json yang ada di direktori proyek.

#### 4. Ekspos Port:

EXPOSE 9000: Menginformasikan Docker bahwa container akan mendengarkan pada port 9000. Port ini digunakan oleh PHP-FPM untuk menerima permintaan dari server web seperti Nginx.

## 2. Node JS Dockerfile

```
express-api > Dockerfile
1  FROM node:20
2  WORKDIR /usr/src/app
3  COPY package*.json ./
4  RUN npm install
5  COPY . .
6  EXPOSE 3000
7  CMD [\"node\", \"app.js\"]
```

**Gambar 6. Node JS API Dockerfile**

Dockerfile ini digunakan untuk membangun image Docker bagi layanan Node JS API dan berfungsi untuk memastikan bahwa semua dependensi Node JS diinstal dan layanan Express siap menerima permintaan pada port 3000. Berikut adalah penjelasan konfigurasinya:

### 1. Base Image:

FROM node:20: Menggunakan image dasar Node.js versi 20 yang mencakup runtime Node.js dan npm (Node Package Manager).

### 2. Menentukan Direktori Kerja:

WORKDIR /usr/src/app: Menentukan direktori kerja di dalam container sebagai /usr/src/app. Semua perintah berikutnya akan dijalankan dalam direktori ini.

### 3. Menyalin dan Menginstal Dependensi:

- COPY package\*.json ./: Menyalin file package.json dan package-lock.json (jika ada) ke direktori kerja dalam container. File ini berisi daftar dependensi proyek.
- RUN npm install: Menjalankan perintah npm install untuk menginstal semua dependensi yang didefinisikan dalam package.json.

### 4. Menyalin Sisa Kode Aplikasi:

COPY . .: Menyalin semua file dari direktori proyek di host ke direktori kerja dalam container.

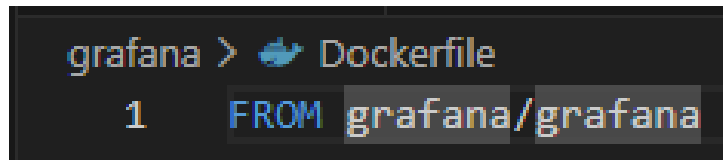
### 5. Ekspos Port:

EXPOSE 3000: Menginformasikan Docker bahwa container akan mendengarkan pada port 3000. Port ini digunakan oleh aplikasi Node.js untuk menerima permintaan HTTP.

### 6. Menjalankan Aplikasi:

CMD [\"node\", \"app.js\"]: Menentukan perintah yang akan dijalankan saat container dimulai. Dalam hal ini, menjalankan aplikasi Node.js dengan node app.js.

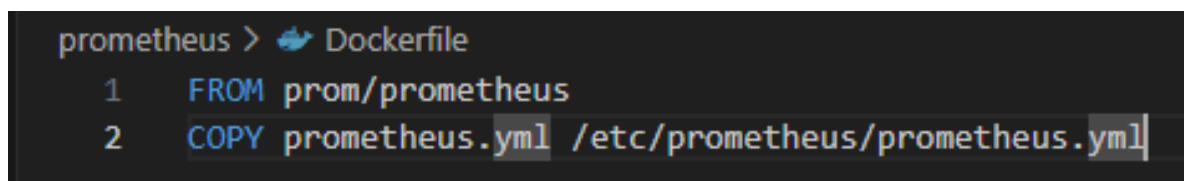
## 5.4 Setup Prometheus dan Grafana



```
grafana > Dockerfile
1 FROM grafana/grafana
```

**Gambar 7. Grafana Dockerfile**

Dockerfile ini digunakan untuk membangun image Docker bagi layanan Grafana. Dockerfile ini tidak membutuhkan konfigurasi tambahan karena image dasar sudah mencakup semua yang diperlukan untuk menjalankan Grafana. Setiap penyesuaian lebih lanjut, seperti konfigurasi data source atau dashboard, biasanya dilakukan melalui file konfigurasi atau volume yang di-mount ke container Grafana saat menjalankannya.



```
prometheus > Dockerfile
1 FROM prom/prometheus
2 COPY prometheus.yml /etc/prometheus/prometheus.yml
```

**Gambar 8. Dockerfile Prometheus**

Dockerfile ini digunakan untuk membangun image Docker bagi layanan Prometheus. Pada Dockerfile tersebut hanya menggunakan konfigurasi tambahan yang berfungsi untuk Menyalin file konfigurasi Prometheus (prometheus.yml) dari direktori build di host ke direktori /etc/prometheus/ di dalam container. File ini berisi konfigurasi job dan target monitoring yang akan digunakan oleh Prometheus.

## BAB 6 UJI COBA SISTEM

### 6.1 Build dan Deployment Image

Setelah lingkungan Docker, service, dan monitoring selesai dikonfigurasi langkah selanjutnya adalah melakukan build dan deployment image. Perintah yang digunakan adalah *sudo docker compose build*. Berikut adalah penjelasan lebih rinci mengenai perintah tersebut dalam konteks proyek ini:

#### 1. Navigasi ke Direktori Proyek:

Sebelum melakukan build, harus dipastikan terlebih dahulu untuk berpindah ke direktori proyek menggunakan perintah *cd*.

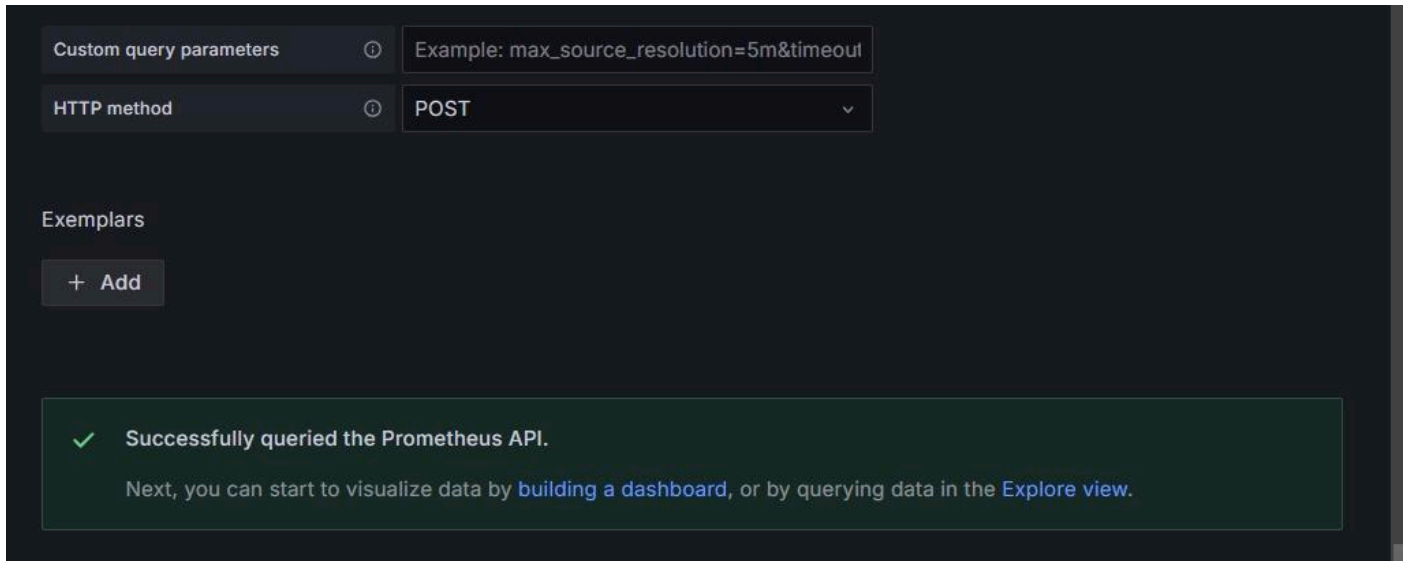
#### 2. Menjalankan Perintah Build:

- Perintah *sudo docker compose build* digunakan untuk membangun image Docker sesuai dengan definisi yang ada dalam file *docker-compose.yml*. Perintah ini akan membaca setiap bagian service dalam file *docker-compose.yml* dan menjalankan langkah-langkah build yang didefinisikan dalam Dockerfile masing-masing service.
- **Build Services:** Setiap service yang didefinisikan dalam *docker-compose.yml* dan memiliki konfigurasi build akan dibangun sesuai dengan instruksi dalam Dockerfile terkait. Ini termasuk menyalin file, menginstal dependensi, dan mengkonfigurasi service seperti yang telah dijelaskan sebelumnya.
- **Caching:** Docker Compose menggunakan caching untuk mempercepat proses build. Jika tidak ada perubahan dalam Dockerfile atau file yang terkopi, Docker Compose akan menggunakan layer build yang telah di-cache sebelumnya.

#### 3. Menjalankan Container

Perintah *sudo docker compose up* akan menjalankan container berdasarkan image yang telah dibangun. Perintah ini juga akan menghubungkan jaringan dan volume yang telah didefinisikan dalam *docker-compose.yml*.

## 6.2 Testing Koneksi Grafana dan Prometheus

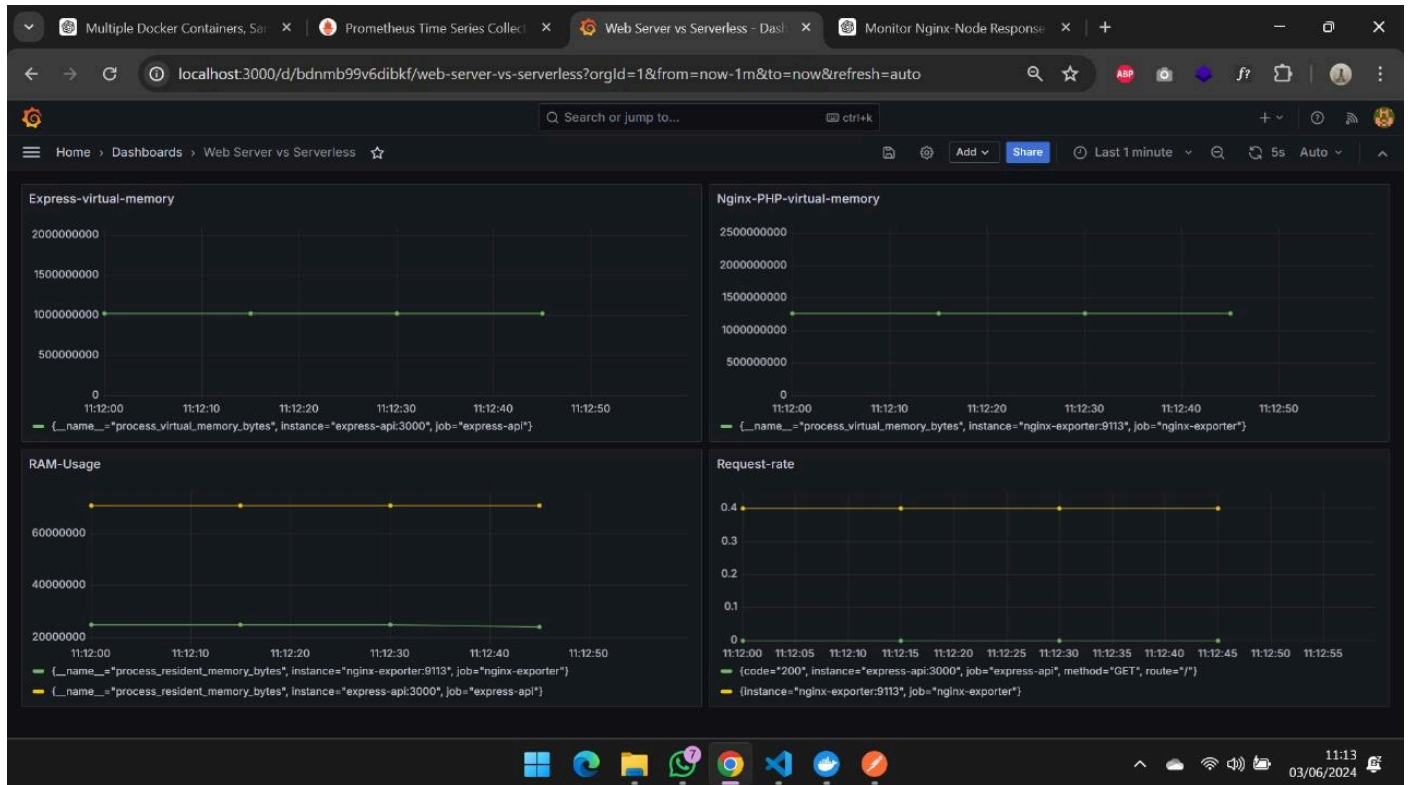


**Gambar 9. Koneksi Grafana dan Prometheus**

Setelah berhasil membangun dan mendeploy container menggunakan Docker Compose, langkah selanjutnya adalah menguji koneksi antara Grafana dan Prometheus. Berikut adalah langkah-langkah yang perlu dilakukan untuk memastikan koneksi ini berfungsi dengan baik:

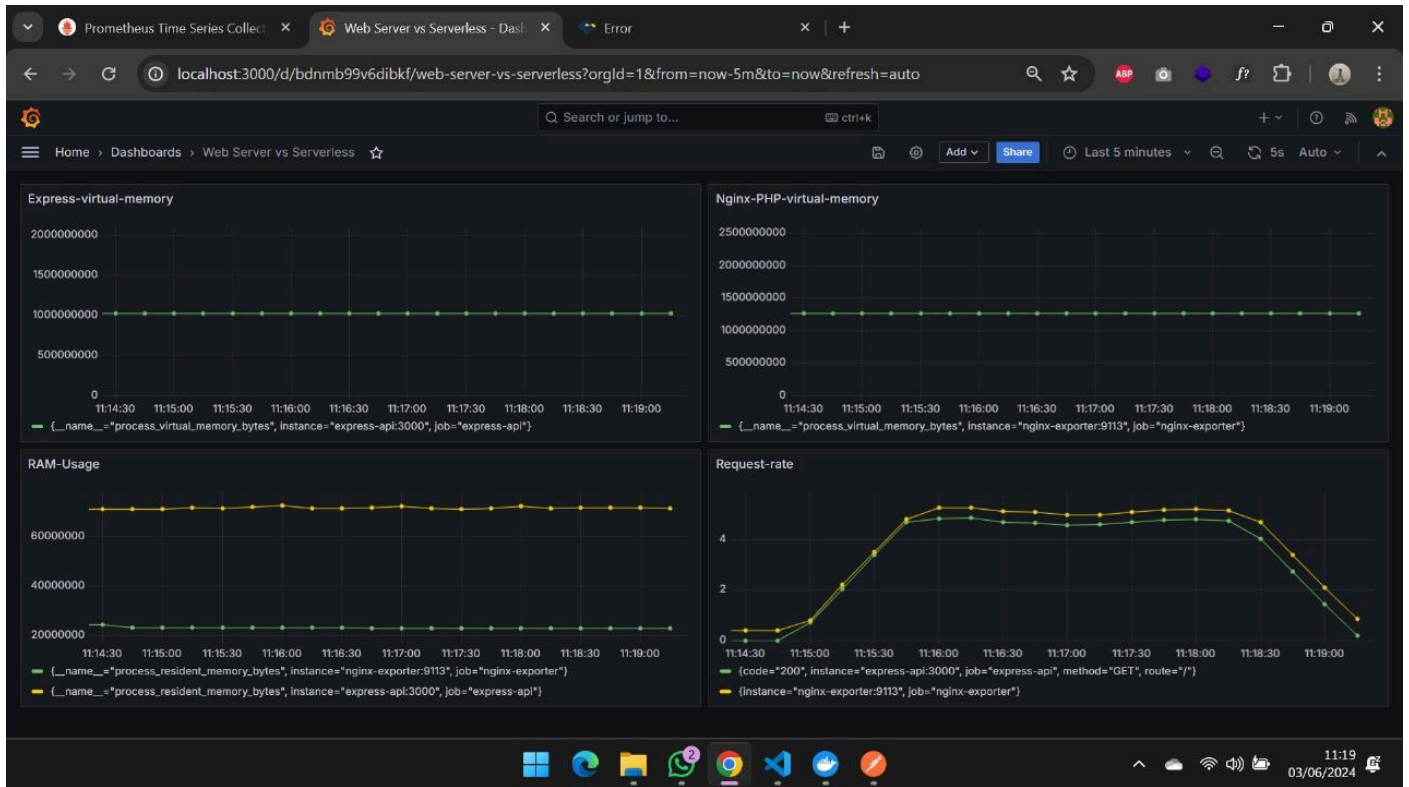
1. **Akses Grafana:** Setelah container berjalan, buka browser web dan akses Grafana melalui URL *http://localhost:3000*.
2. **Login ke Grafana:** Masukkan kredensial login Grafana. Terdiri dari username dan password.
3. **Konfigurasi Data Source:** Di halaman konfigurasi Data Source Prometheus, isi URL dengan URL Prometheus yang dijalankan dalam container Docker. URL ini adalah *http://localhost:9090*.
4. **Simpan dan Uji Koneksi:** Klik tombol "Save & Test" untuk menyimpan konfigurasi dan menguji koneksi ke Prometheus. Jika konfigurasi benar, Maka akan muncul pesan seperti pada **Gambar 9** yang menginformasikan bahwa koneksi berjalan dengan benar.

## 6.3 Dashboard Grafana



**Gambar 10. Dashboard Grafana**

Untuk membuat dashboard pada Grafana dapat melanjutkan langkah pada subbab 6.1. Setelah itu tinggal menambahkan data-data metrics yang ingin diambil. Seperti pada **Gambar 10** metrics yang diambil adalah sesuai seperti pada yang didefinisikan pada bab ruang lingkup. Semua grafik terlihat lurus dikarenakan state pada saat ini belum dilakukan hit API

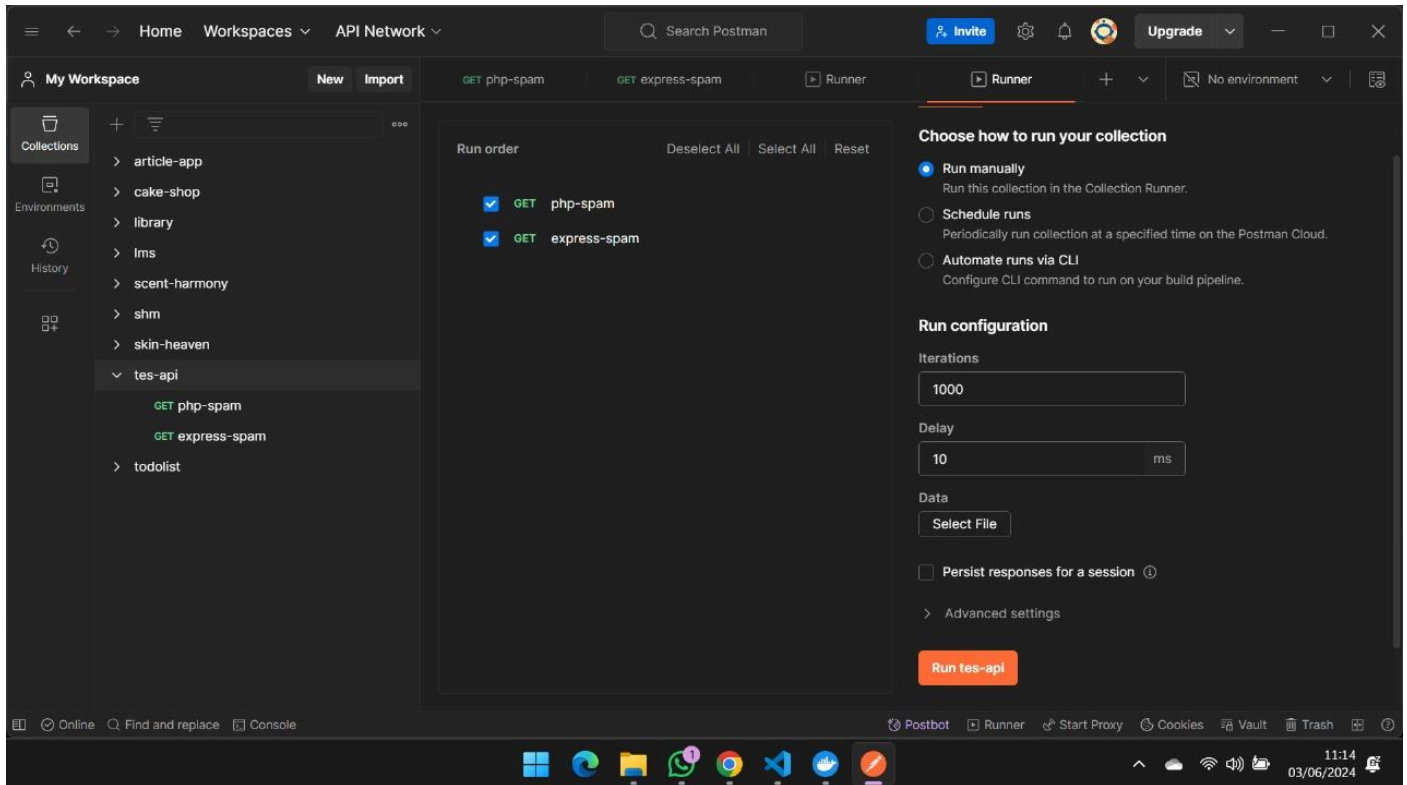


**Gambar 11. Dashboard Grafana**

Keadaan grafik monitoring setelah dilakukan ujicoba hit API.



## 6.4 Hit Test API Menggunakan Postman



**Gambar 12. Postman**

Ujicoba hit API dilakukan menggunakan Postman dengan frekuensi 1000 request yang diberi jeda 10 milisekon.

## BAB 7 KESIMPULAN

### 7.1 Analisa Hasil

Berdasarkan pengamatan pada sistem monitoring, ditemukan bahwa penggunaan virtual memory oleh Nginx cenderung stabil namun tinggi. Sebaliknya, arsitektur serverless menunjukkan alokasi virtual memory yang lebih rendah, tetapi tidak konstan dan bervariasi tergantung pada jumlah permintaan yang diterima. Dalam hal performa, arsitektur serverless terbukti lebih cepat dalam menangani permintaan dibandingkan dengan Nginx web server. Namun, arsitektur serverless juga cenderung menggunakan lebih banyak memori secara keseluruhan. Hal ini terjadi karena dalam arsitektur serverless, web server dan API diintegrasikan dalam satu entitas, sehingga tidak ada pemisahan beban kerja ke dalam container terpisah seperti pada konfigurasi Nginx dengan container terpisah untuk web server dan API. Analisis ini menunjukkan bahwa meskipun arsitektur serverless menawarkan keuntungan dalam hal kecepatan respons, pengelolaan memori menjadi lebih kompleks dan membutuhkan optimalisasi yang cermat untuk memastikan efisiensi.

## 7.2 Kesimpulan

Berdasarkan analisa hasil monitoring, dapat disimpulkan bahwa terdapat perbedaan signifikan dalam penggunaan sumber daya dan performa antara arsitektur Nginx dan serverless. Nginx menunjukkan penggunaan virtual memory yang stabil namun tinggi, sementara arsitektur serverless menggunakan virtual memory yang lebih rendah namun bervariasi sesuai dengan jumlah permintaan. Dalam hal kecepatan menangani permintaan, arsitektur serverless lebih unggul dibandingkan Nginx. Namun, serverless cenderung menggunakan lebih banyak memori secara keseluruhan, karena integrasi web server dan API dalam satu entitas tanpa pemisahan container. Oleh karena itu, meskipun arsitektur serverless menawarkan keuntungan dalam kecepatan, pemilihan antara Nginx dan serverless harus mempertimbangkan kebutuhan spesifik dari penggunaan memori dan performa aplikasi yang diinginkan. Optimalisasi lebih lanjut diperlukan untuk mencapai keseimbangan yang ideal antara efisiensi sumber daya dan kecepatan respons dalam arsitektur serverless.