Final Project Report

DevOps Pipeline for a Multi-Service Application

Giorgi Gagnidze

June 2025

Table of Contents

1. Introduction

This project demonstrates the design and implementation of a modern DevOps pipeline for a multi-service application. The main objective is to containerize and orchestrate multiple services using Docker and Docker Compose, while integrating monitoring, security, and automation practices throughout the application lifecycle.

Key components of the project include:

- **Containerization:** Both frontend and backend services are packaged into independent containers with custom Dockerfiles and served by NGINX.
- **Orchestration:** Docker Compose is used to define and run the entire stack, ensuring inter-service communication and simplified container management.
- **Monitoring & Visualization:** Prometheus collects real-time metrics from the backend service, and Grafana provides interactive dashboards to visualize application health and performance, alert manager is responsible for sending alert emails to maintainers in case of a failure.
- **Stress Testing:** Stress testing with Gatling, more than 1000 concurrent requests for CRUD operations, and on-purpose, increased CPU & memory usage within each request to better visualize utilization metrics data in Grafana.
- **Security:** The security of all Docker images is assessed using Trivy. Sensitive data is managed securely using .env files and GitHub Actions secrets, as well as Ansible is responsible for securely uploading .env files on the VM.
- **Automation:** Ansible is used to automate deployment and provisioning tasks (updating Linux repositories, creating folders, etc...) reducing manual intervention and ensuring smooth deployment process to have the infrastructure ready to go.

- **Infrastructure as Code (IaC):** Terraform provisions the cloud infrastructure, in this case GCP virtual machine and manages it automatically, ensuring reproducibility and ease of scaling.
- **Version Control:** The entire project is tracked with Git and hosted on GitHub, which also provides GitHub actions, enabling effective collaboration, transparency and automation. Deploy is done on a Terraform provided machine

2. General Project Structure

```
devops-final-project/
├── .github/                # GitHub actions
├── backend/                # Spring Boot backend service
│   ├── build/
│   ├── checkstyle.xml
│   ├── Dockerfile
├── frontend/               # React frontend service
│   ├── build/
│   ├── public/
│   ├── src/
│   ├── .env.development
│   ├── Dockerfile
│   ├── package.json
│   ├── package-lock.json
│   └── tsconfig.json
├── infra/                  # Infrastructure and deployment
│   ├── ansible/
│   ├── backend/      # ENV
│   ├── frontend/     # ENV
│   ├── gatling/      # Gatling results and simulation
│   ├── monitoring/   # Grafana, Prometheus and Alert manager
```

```
│   ├── nginx/          # Folder containing nginx.conf
│   ├── terraform/      # terraform files provisioning GCP Instance
│   ├─ compose.yml  # Docker compose file
│   └── scan-all.sh     # Trivy script that scans all Docker images
```

3. Application Overview

The application is a full-stack, cloud-deployed system, consisting of a Spring Boot backend REST API (with Swagger) and a ReactJS frontend served by NGINX. The backend exposes RESTful APIs for creating TODO-tasks which are updateable, deletable and savable.

The Backend comes with H2 embedded database and exposes health check endpoints for Prometheus and Grafana. It also comes with checktyle and JUnit integration tests.

```
spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:h2:mem:testdb}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:sa}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:}
spring.datasource.driverClassName=org.h2.Driver

management.endpoints.web.exposure.include=*
management.endpoint.prometheus.access=unrestricted
management.prometheus.metrics.export.enabled=true
```

Architecture Overview

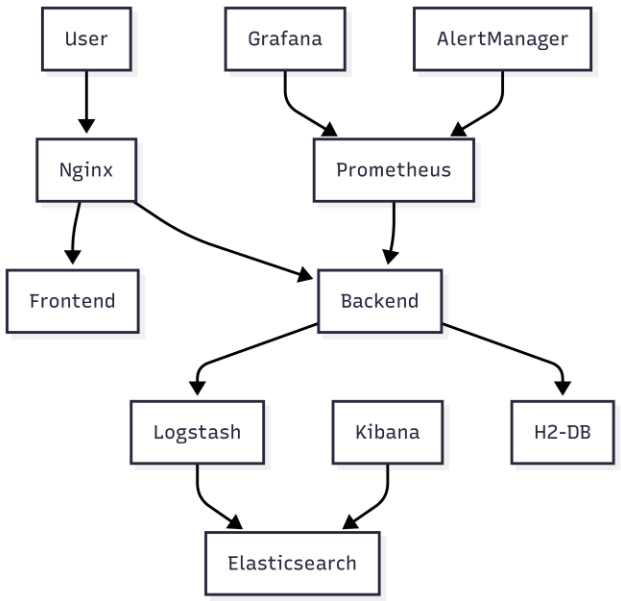
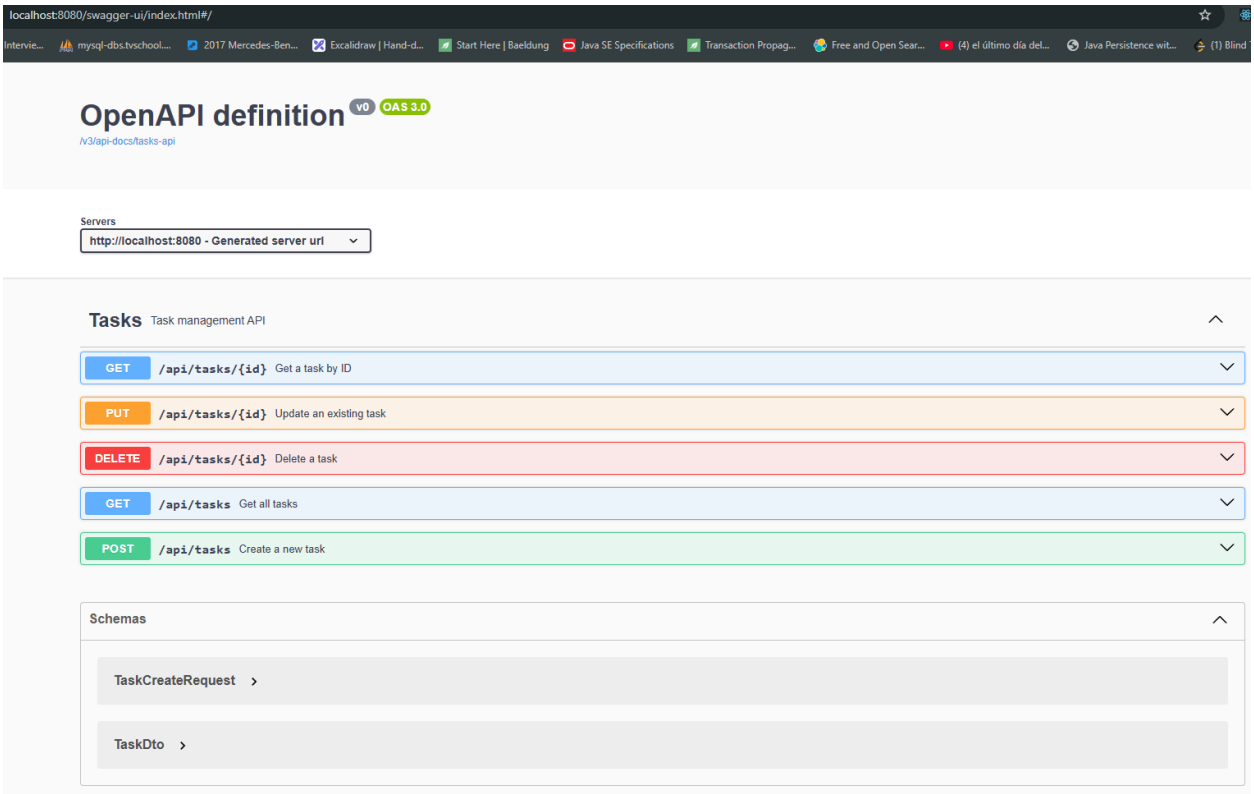


Image above displays the general flow, how the user is served by NGINX and the backend communication with Alerting and Monitoring services.

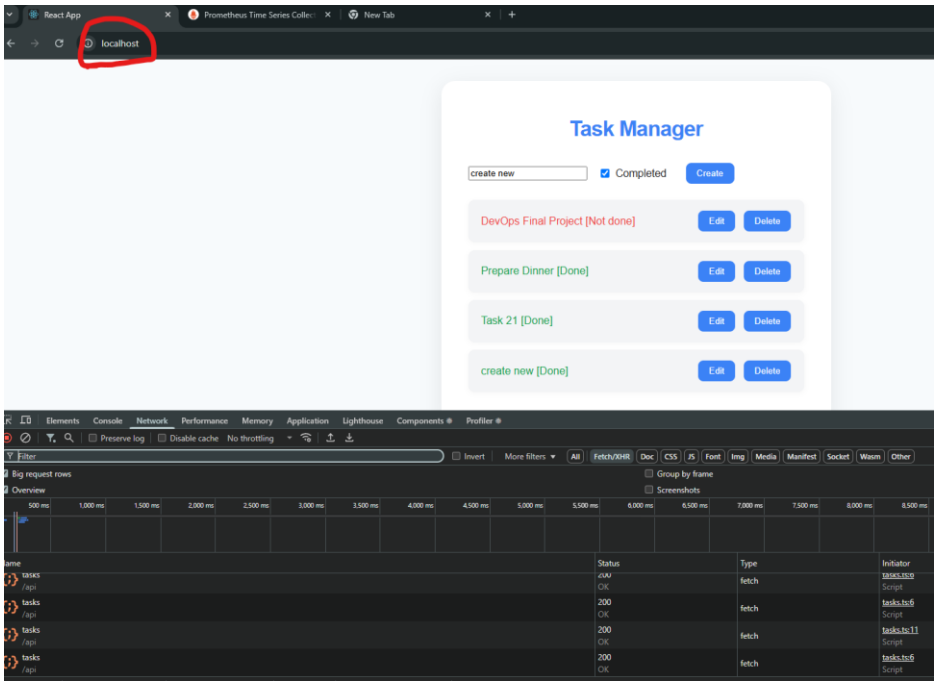
Demo

1) API:



2) UI served by NGINX:

The user can create, update, or delete tasks from the dashboard



4. Containerize Services

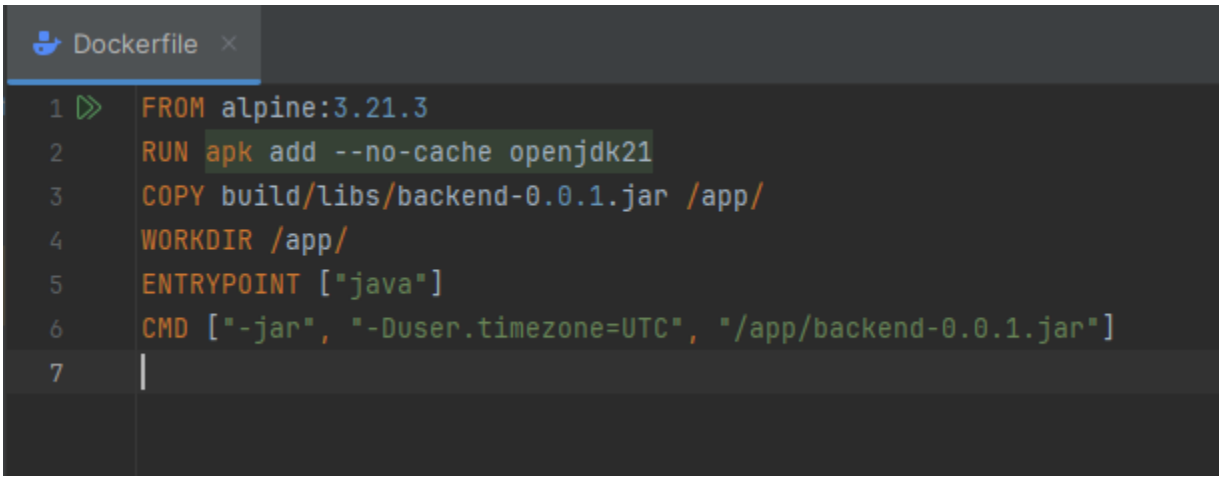
The task was to create at least two services (backend, frontend) and ensure services ran independently in the containers.

Under my root directory I've created 3 folders

- Backend – containing backend service with Dockerfile
- Frontend – containing front end service with its own Dockerfile as well
- Infra – containing all of the infrastructure related stuff, as well as Compose.yml file

Backend

This is the Dockerfile for my backend service, copying generated jar from the build folder to the app directory and providing a command for running the app



Frontend

This is the Dockerfile for my frontend service

Building Containers

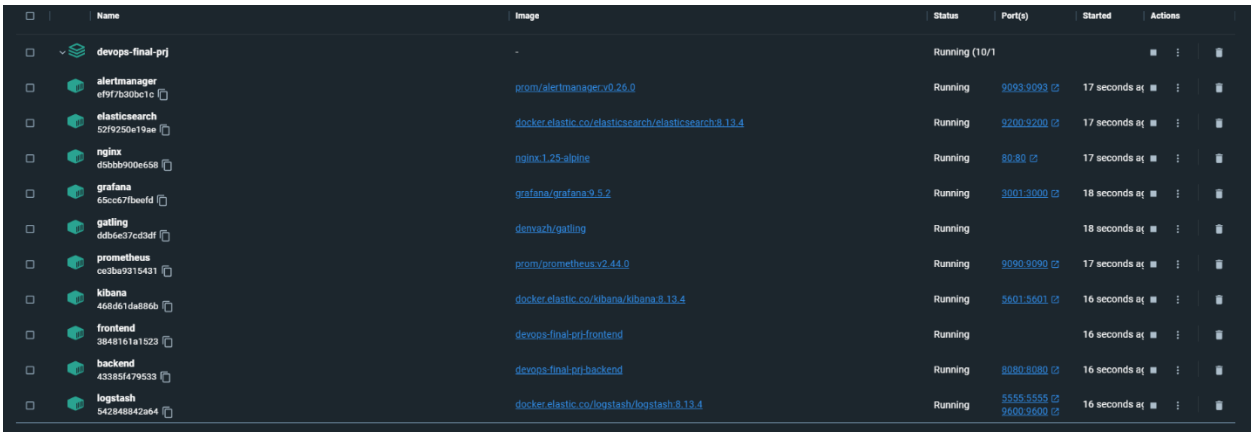
Here’s my compose.yml file snippet containing backend and front end images

Building backend with docker compose build --no-cache backend

```
PS C:\Users\giorg\Desktop\devops-final-project\infra> docker compose build --no-cache backend
[+] Building 37.2s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/alpine:3.21.3
 => [auth] library/alpine:pull token for registry-1.docker.io
 => CACHED [1/4] FROM docker.io/library/alpine:3.21.3@sha256:a8560b36e8b8210634f77d9f7f9efd7ffa463e380b75e2e74aff4511df3ef88c
```

Building frontend with docker compose build --no-cache frontend

```
Terminal   Local ×   Ubuntu-20.04 ×   Ubuntu-20.04 (2) ×   +
PS C:\Users\giorg\Desktop\devops-final-project\infra> docker compose build --no-cache frontend
[+] Building 92.2s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:14.9.0
 => [auth] library/node:pull token for registry-1.docker.io
 => [1/6] FROM docker.io/library/node:14.9.0@sha256:ce506ed8986a0c8a364757771679706ebd129fa466165fcc6e2c7dc449a0baac
 => [internal] load build context
```



NGINX Configuration

nginx.conf

```
worker_processes 1;
events { worker_connections 1024; }

http {
    include       mime.types;
    default_type  application/octet-stream;

    server {
        listen 80;

        location / {
            proxy_pass http://frontend:3000/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        location /api/ {
            proxy_pass http://backend:8080/api/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

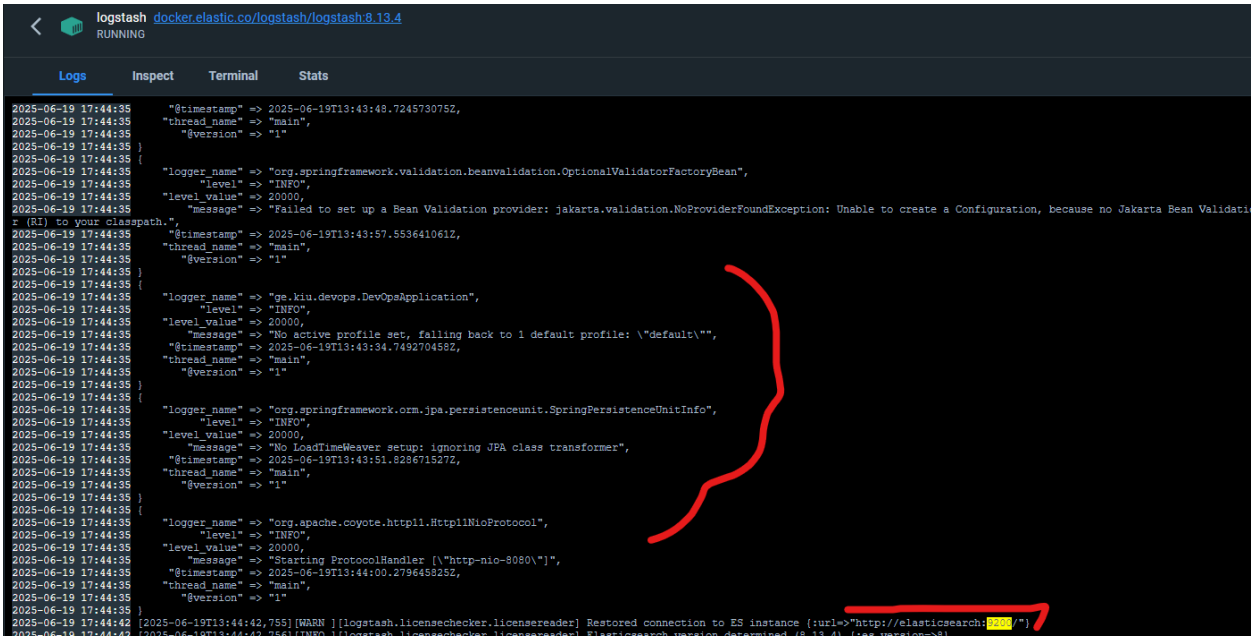
```
nginx:
  image: nginx:1.25-alpine
  container_name: nginx
  ports:
    - "80:80"
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
```

5. Compose the Stack

```
Terminal  Local  Ubuntu-20.04  Ubuntu-20.04 (2)
PS C:\Users\giorg\Desktop\devops-final-project\infra> docker compose up -d
[+] Running 11/11
 - Network devops-final-prj_default  Created                    0.0s
 - Container prometheus              Started                    2.2s
 - Container gatling                 Started                    1.1s
 - Container grafana                 Started                    1.4s
 - Container elasticsearch           Started                    2.0s
 - Container nginx                   Started                    2.3s
 - Container alertmanager            Started                    2.1s
 - Container logstash                Started                    3.2s
 - Container kibana                  Started                    3.0s
 - Container frontend                Started                    3.5s
 - Container backend                 Started                    3.5s
PS C:\Users\giorg\Desktop\devops-final-project\infra>
```

By default, all services defined in Compose.yml file are automatically connected to default network, if not defined custom one.

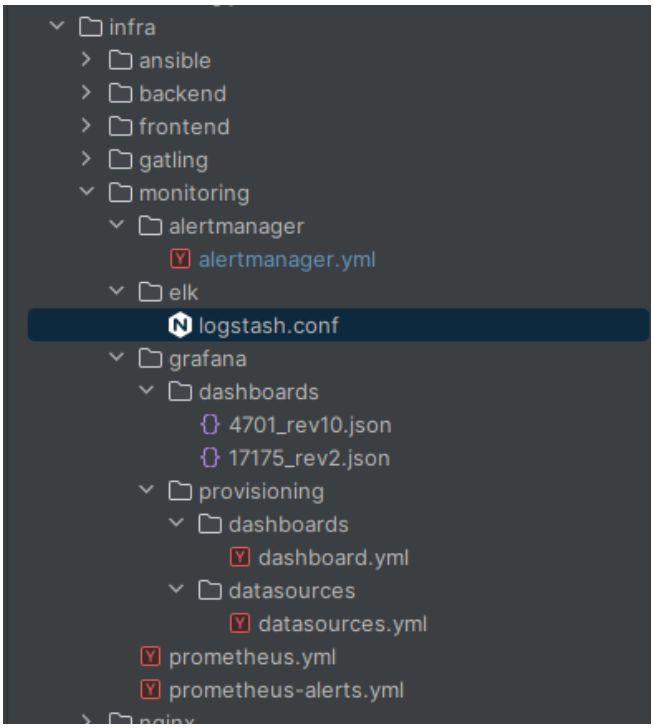
To see that the communication works between services, we can check logs, for example logstash, we see application logs coming from the backend service and connection to the elastic:



6. Monitoring & Visualization

For the Monitoring and Visualization part, I've set up everything: ELK Stack, Grafana and Prometheus with Alerts.

```yaml
prometheus:
  image: prom/prometheus:v2.44.0
  container_name: prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    - ./monitoring/prometheus-alerts.yml:/etc/prometheus/prometheus-alerts.yml

alertmanager:
  image: prom/alertmanager:v0.26.0
  container_name: alertmanager
  ports:
    - "9093:9093"
  volumes:
    - ./monitoring/alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml

grafana:
  image: grafana/grafana:9.5.2
  container_name: grafana
  ports:
    - "3001:3000"
  volumes:
    - ./monitoring/grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
    - ./monitoring/grafana/provisioning/dashboards:/etc/grafana/provisioning/dashboards
    - ./monitoring/grafana/dashboards:/var/lib/grafana/dashboards
```



The volumes are mounted to the corresponding directories in the project.

As seen here, I have two dashboards for Grafana, one visualizing Spring Boot metrics, the other visualizes JVM Specific metrics aka Micrometer.

Dashboards file tells Grafana which dashboards to use that are provided, datasources define and map prometheus variable as a source of data for the given dashboard templates.

Alertmanager yaml file contains smtp secrets needed for sending emails to the users.

prometheus.yml

```yaml
global:
  scrape_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - 'alertmanager:9093'

rule_files:
  - 'prometheus-alerts.yml'

scrape_configs:
  - job_name: 'backend-metrics'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'host.docker.internal:8080' ]
        labels:
          application: 'backend-metrics'
```

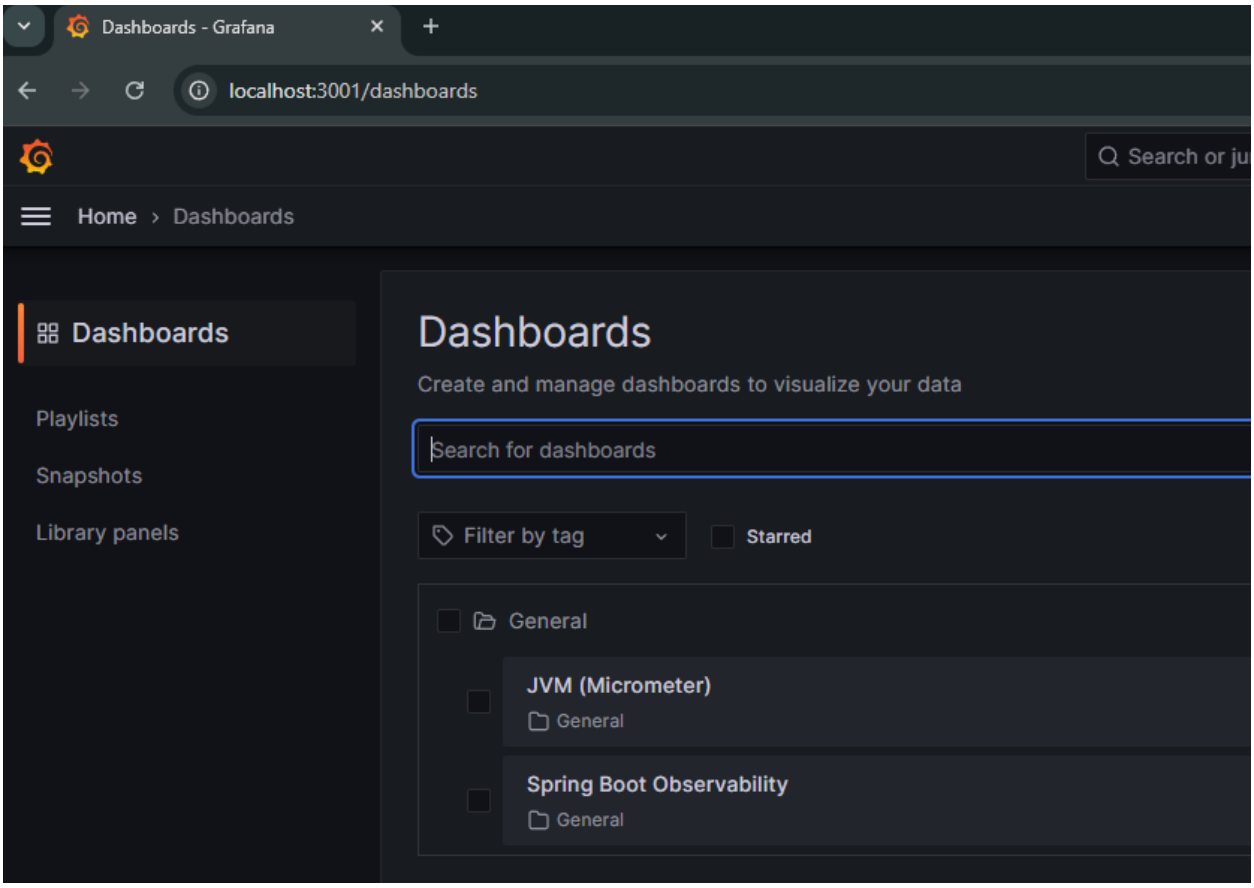
prometheus-alerts.yml

```yaml
groups:
  - name: service-alerts
    rules:
      - alert: BackendDown
        expr: up{job="backend-metrics"} == 0
        for: 30s
        labels:
          severity: critical
        annotations:
          summary: "Backend API is down"
          description: "The Spring Boot backend (job=

      - alert: HighBackendCPU
        expr: process_cpu_seconds_total{job="backend-
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "Backend CPU usage is high"
          description: "Backend process CPU usage has
```

These two files configure Prometheus to scrape the backend health from the actuator endpoint as well as configure communication between Prometheus and alert manager. The file on the right defines two tasks or the rulesets according to which alerts are sent in case of an expression logic is satisfied, for example when the backend service is down for more than 30 seconds and etc.

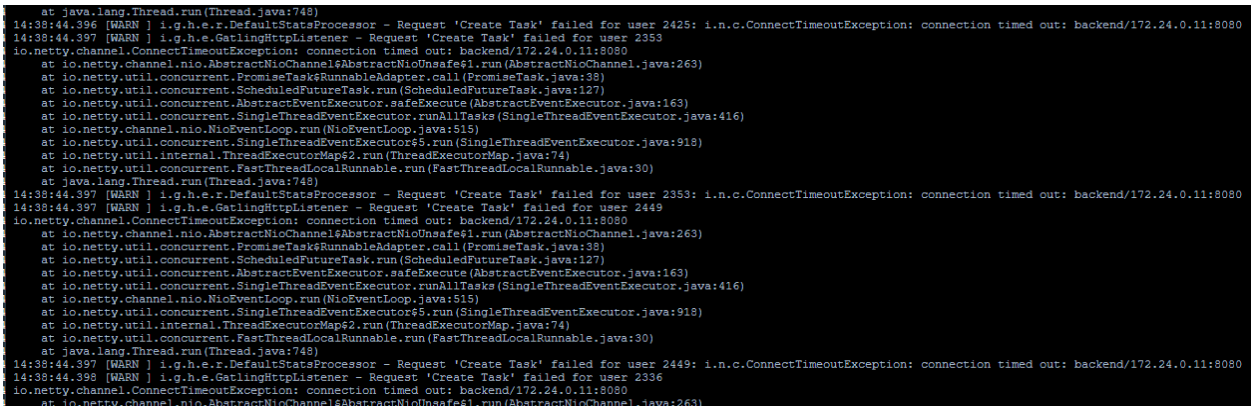
Visualization

We see the dashboards we defined earlier: Micrometer and Spring Boot related

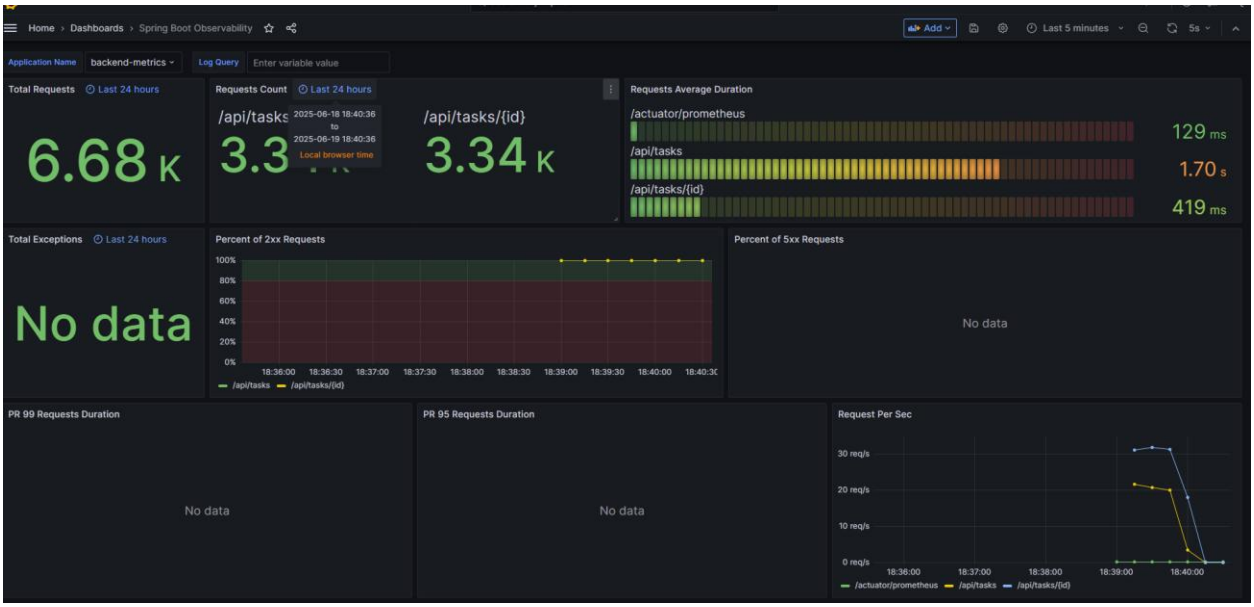


Now, let's observe how the metrics are going, considering Gatling is running after starting the Docker compose entirely.

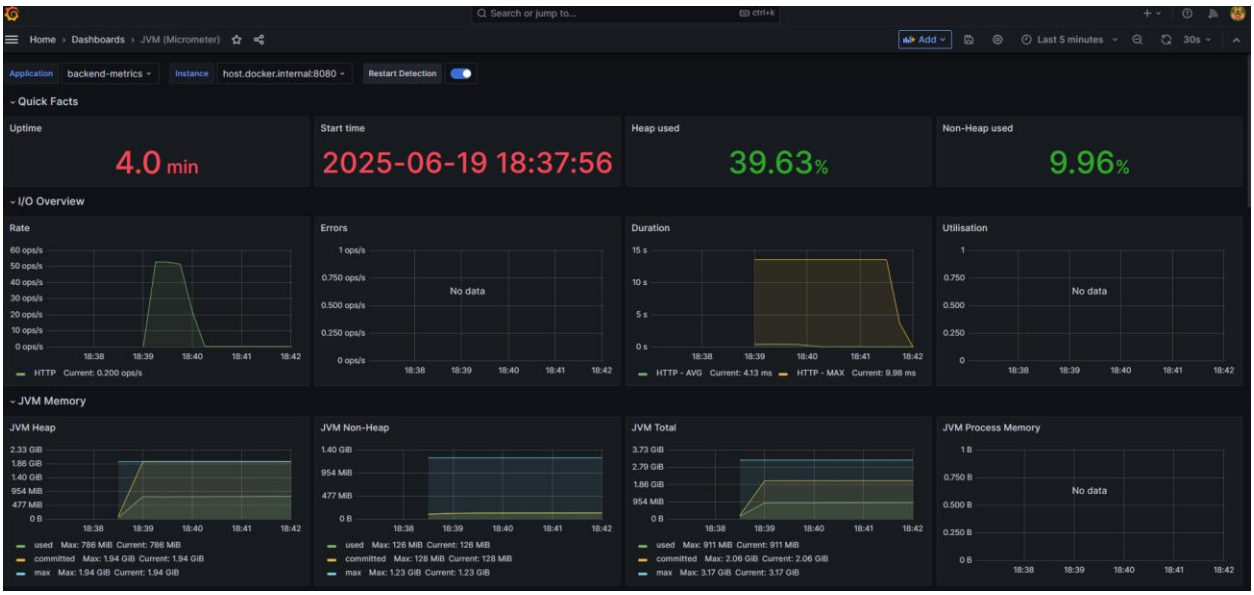
For the first test, I set 2500 concurrent requests for CRUD on the controller, seems its not able to handle it all together, considering there's timeout exceptions in Gatling logs



Here’s the visualization of Spring Boot backend metrics, as we see there’s lots of requests and mostly 2xx statuses.

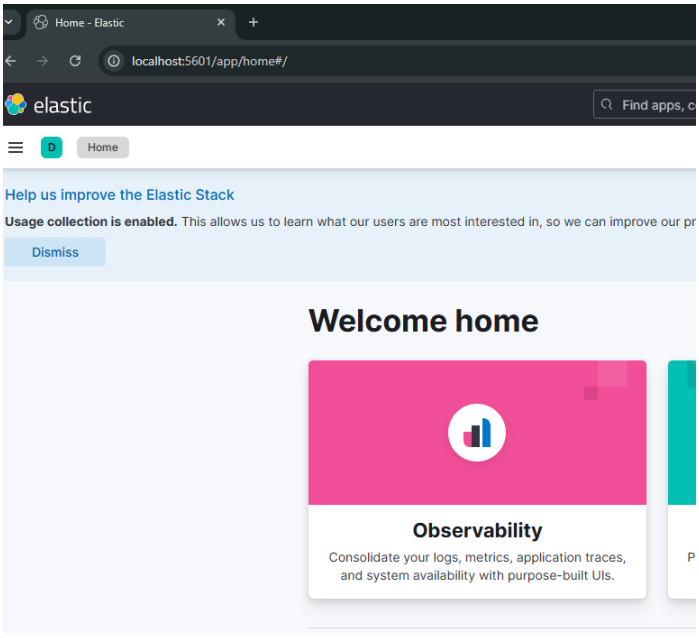


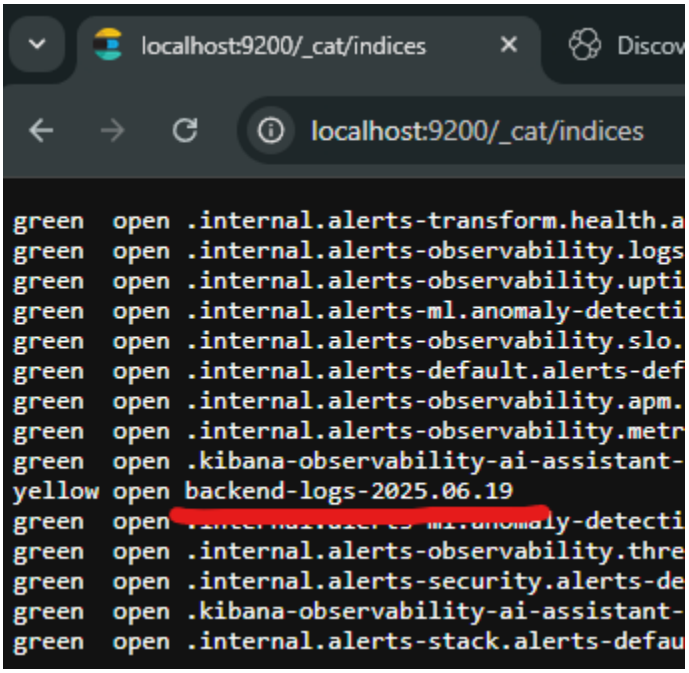
If we take a look at the JVM metrics now, we will see that the CPU and Memory utilization is quite high (or at least was quite high at its peak). The app has been running for the last 5 minutes and utilization is degrading as the requests are not loading the server anymore…



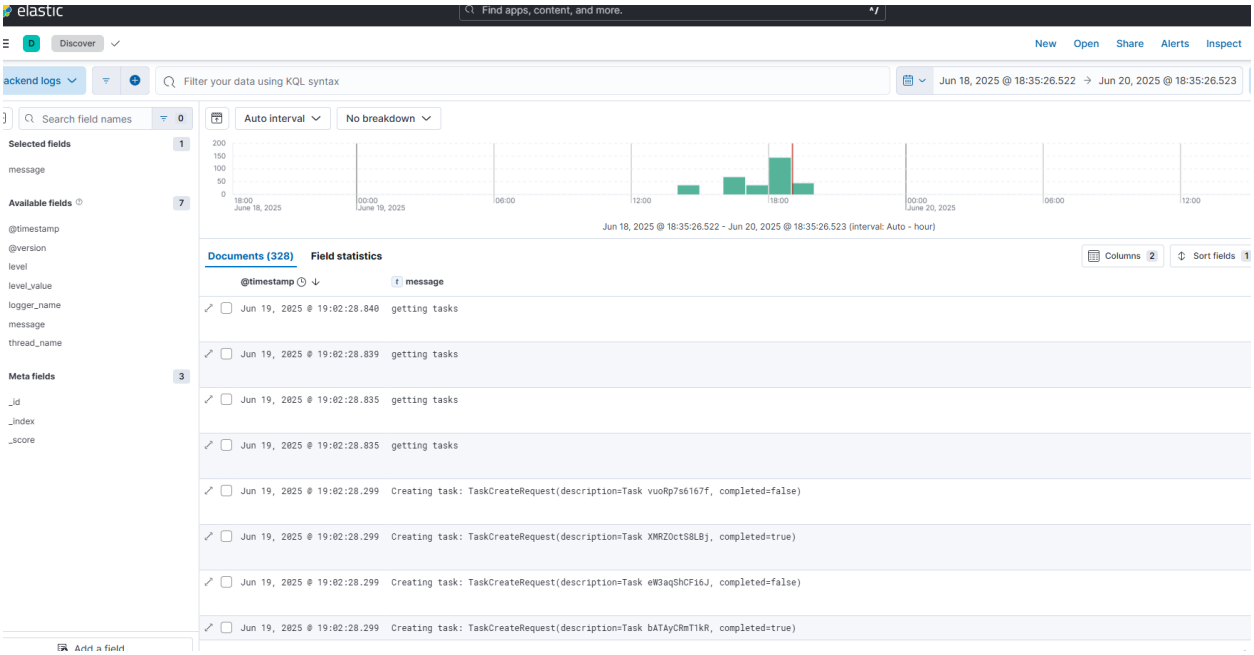
Logging

The Logging part is also interesting, where I defined ELK stack gathering logs from the backend to elastic with Logstash and visualizing them with Kibana. We need to go to the dashboards define the pattern and see the logs from there, currently my logs are under this elastic index as we see.



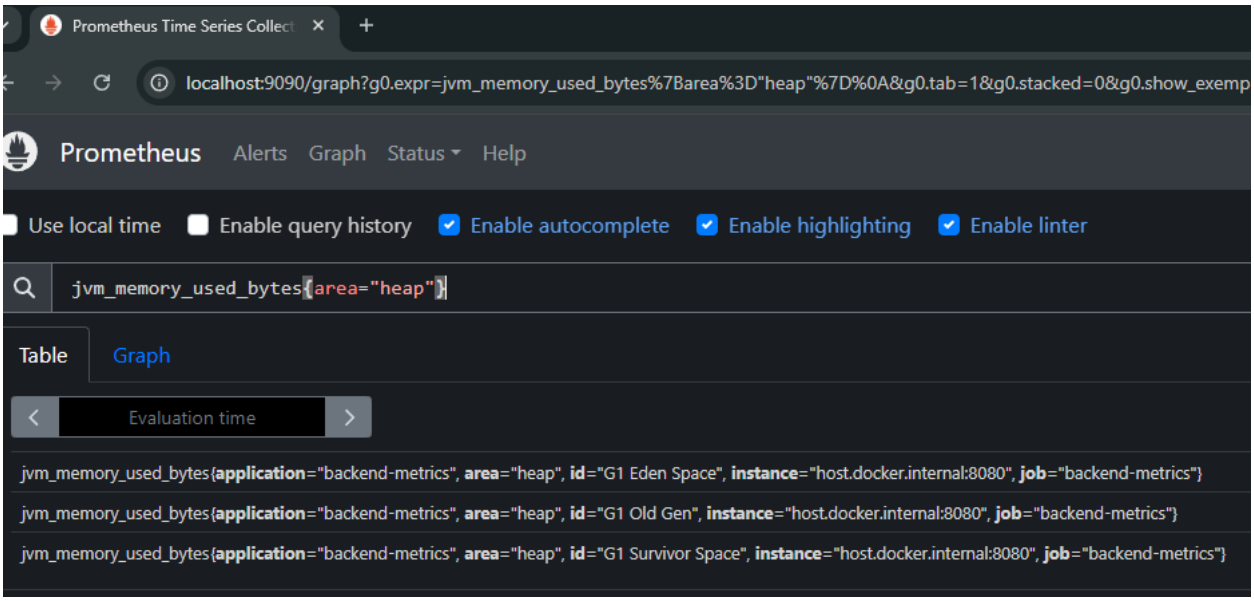


And if we go to discover page, select the message field from the left, we'll see the application logs as shown here.

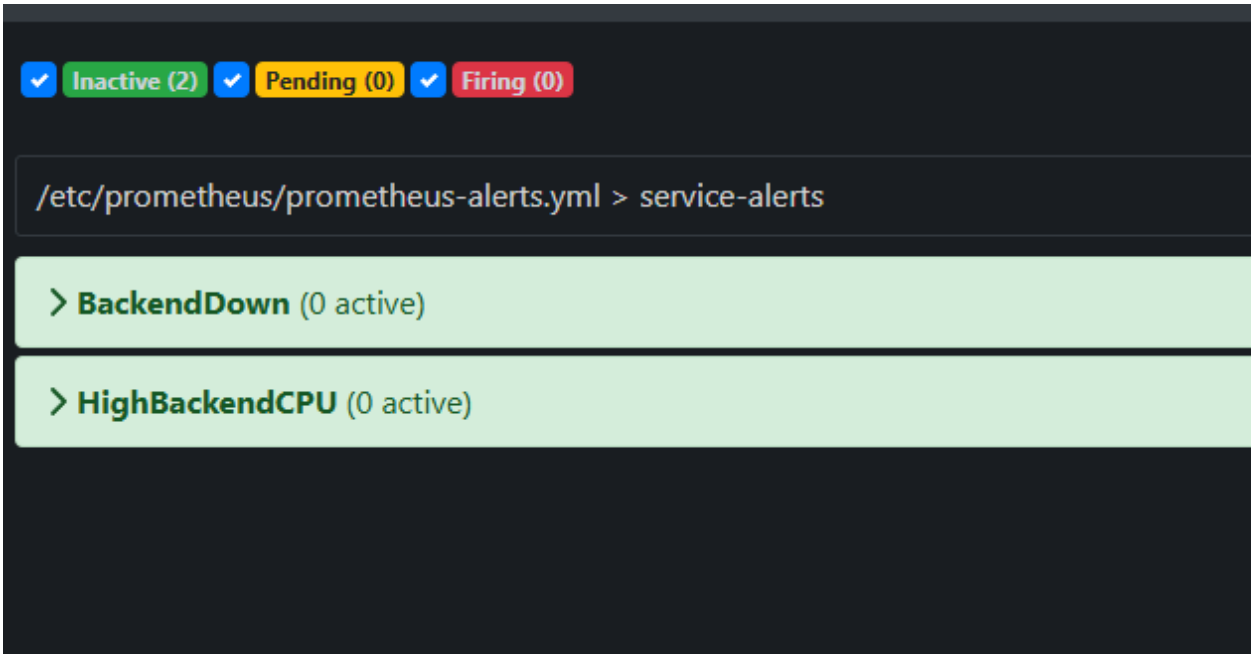


Prometheus

And here, Prometheus shows whatever metric query we give to it, in this case we see the consumption in the heap area of JVM selected by this query

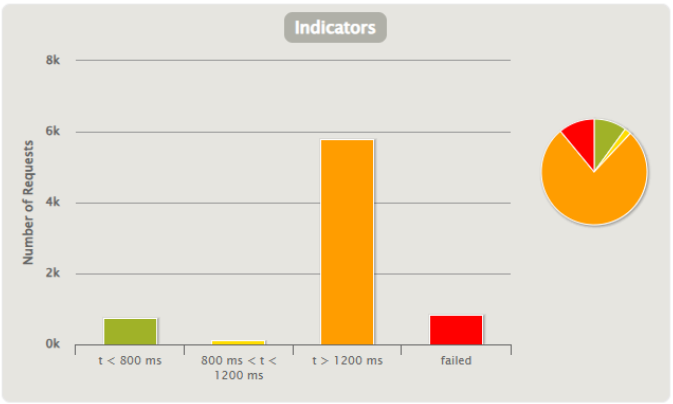


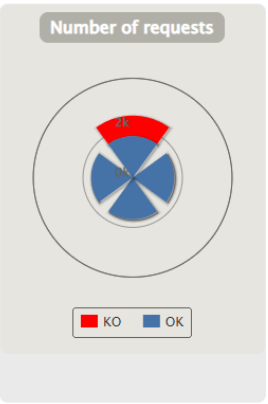
Alerting tasks which to we will come back later at **Incident Simulation**



Gatling Simulation Results

2000+ Concurrent Requests

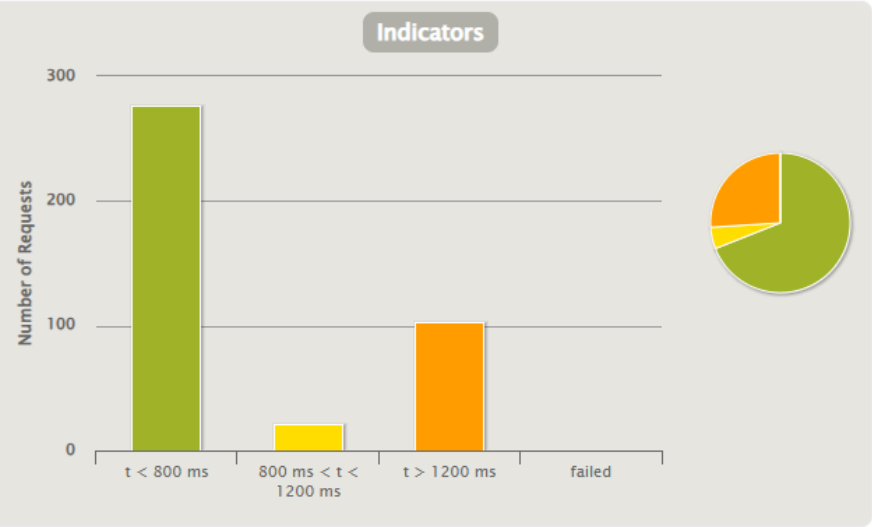


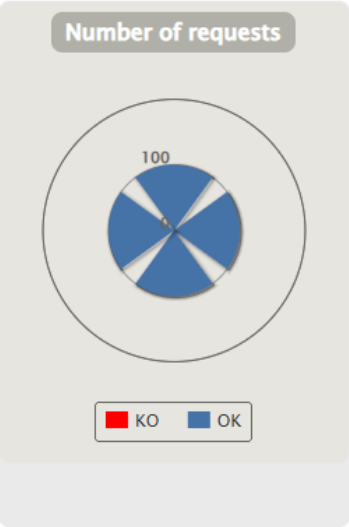


STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	7513	6684	829	11%	197.711	3	5709	10352	23687	25811	35067	8649	7437
Create Task	2500	1671	829	33%	65.789	3974	10395	23237	25367	34506	35067	15843	7123
Update Task	1671	1671	0	0%	43.974	7	4509	15016	18735	20236	21098	7632	5941
Get All Tasks	1671	1671	0	0%	43.974	7	5239	5875	6480	15457	16142	4703	2482
Delete Task	1671	1671	0	0%	43.974	3	2791	4272	6007	6446	7299	2851	1913

100 Concurrent Request





STATISTICS

Expand all groups | Collapse all groups

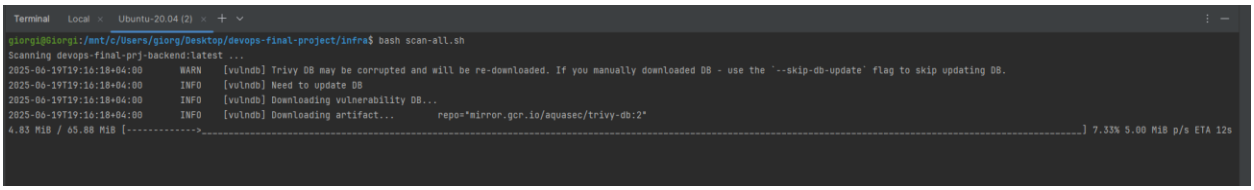
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	400	400	0	0%	100	11	523	1235	1812	2002	2227	709	586
Create Task	100	100	0	0%	25	1202	1609	1770	1982	2191	2227	1634	208
Update Task	100	100	0	0%	25	42	420	586	980	1281	1429	502	270
Get All Tasks	100	100	0	0%	25	26	535	625	812	974	1115	488	209
Delete Task	100	100	0	0%	25	11	235	305	381	609	672	214	141

7. Security

scan-all.sh

```
#!/bin/bash
set -e

IMAGES="
devops-final-prj-backend:latest
devops-final-prj-frontend:latest
nginx:1.25-alpine
prom/prometheus:v2.44.0
prom/alertmanager:v0.26.0
grafana/grafana:9.5.2
denvazh/gatling:latest
docker.elastic.co/elasticsearch/elasticsearch:8.13.4
docker.elastic.co/logstash/logstash:8.13.4
docker.elastic.co/kibana/kibana:8.13.4
"   Giorgi.Gagnidze, Yesterday • add trivy scan script

OUTPUT="trivy-report.txt"
# shellcheck disable=SC2188
> "$OUTPUT"

for image in $IMAGES; do
  echo "======================================" >> "$OUTPUT"
  echo "Scanning $image ..." | tee -a "$OUTPUT"
  trivy image "$image" | tee -a "$OUTPUT"
done
```

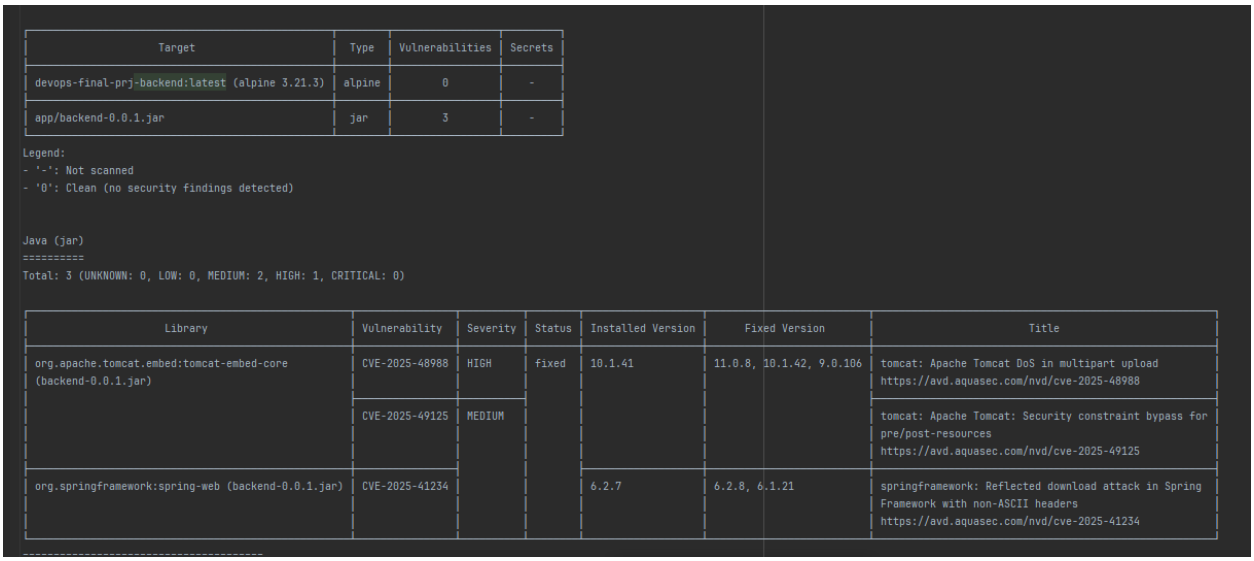
The requirement was to run **Trivy** scan on all Docker images, so I decided to write a shell script, in which there would be a hardcoded names of Docker images and it would scan everything automatically



This generated a huge file, or more than 20K lines, considering the fact that we have many images in our Docker compose.

Let’s try to find and address any of the important issues that Trivy found. Most of the issues are version related, so upgrading package and image versions should help.

In my case for example, I can just upgrade Spring Boot version to avoid this error.



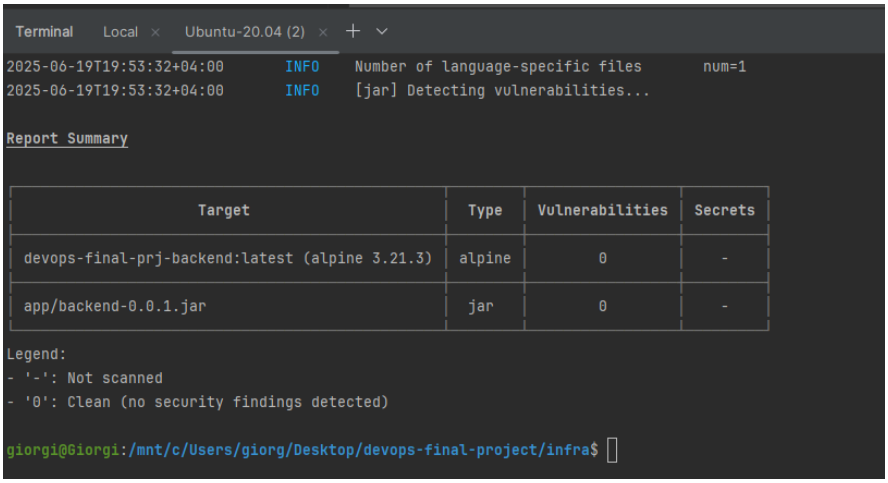
Target	Type	Vulnerabilities	Secrets
devops-final-prj-backend:latest (alpine 3.21.3)	alpine	0	-
app/backend-0.0.1.jar	jar	3	-

Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)

Java (jar)
==========
Total: 3 (UNKNOWN: 0, LOW: 0, MEDIUM: 2, HIGH: 1, CRITICAL: 0)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
org.apache.tomcat.embed:tomcat-embed-core (backend-0.0.1.jar)	CVE-2025-48988	HIGH	fixed	10.1.41	11.0.8, 10.1.42, 9.0.106	tomcat: Apache Tomcat DoS in multipart upload https://avd.aquasec.com/nvd/cve-2025-48988
	CVE-2025-49125	MEDIUM				tomcat: Apache Tomcat: Security constraint bypass for pre/post-resources https://avd.aquasec.com/nvd/cve-2025-49125
org.springframework:spring-web (backend-0.0.1.jar)	CVE-2025-41234			6.2.7	6.2.8, 6.1.21	springframework: Reflected download attack in Spring Framework with non-ASCII headers https://avd.aquasec.com/nvd/cve-2025-41234

After upgrading the versions the vulnerabilities are gone as we see:



Terminal Local × Ubuntu-20.04 (2) × +

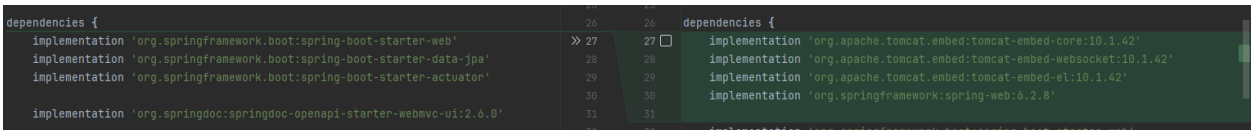
2025-06-19T19:53:32+04:00 INFO Number of language-specific files num=1
2025-06-19T19:53:32+04:00 INFO [jar] Detecting vulnerabilities...

Report Summary

Target	Type	Vulnerabilities	Secrets
devops-final-prj-backend:latest (alpine 3.21.3)	alpine	0	-
app/backend-0.0.1.jar	jar	0	-

Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)

giorgi@Giorgi:/mnt/c/Users/giorg/Desktop/devops-final-project/infra$



```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-actuator'

    implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.6.0'
```

```
dependencies {
    implementation 'org.apache.tomcat.embed:tomcat-embed-core:10.1.42'
    implementation 'org.apache.tomcat.embed:tomcat-embed-websocket:10.1.42'
    implementation 'org.apache.tomcat.embed:tomcat-embed-el:10.1.42'
    implementation 'org.springframework:spring-web:6.2.8'
```

This means we have successfully utilized Trivy to find vulnerabilities in our Docker images and fixed them accordingly.

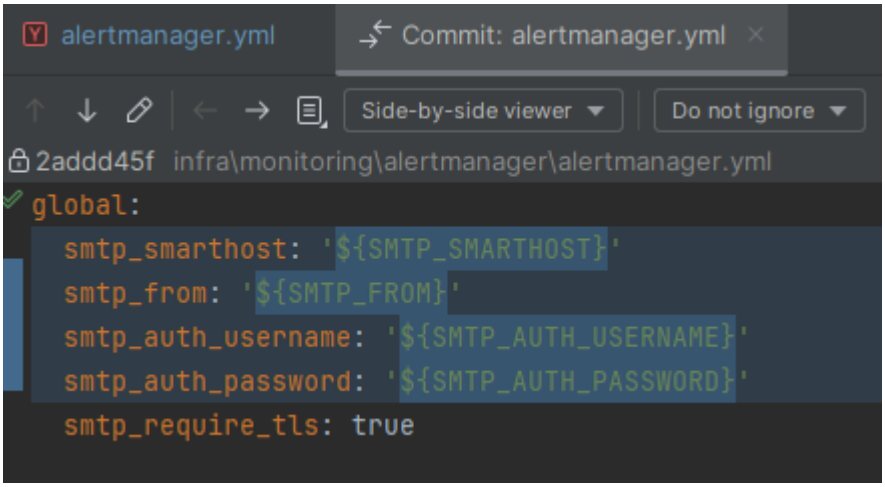
Managing Sensitive Data

As you might have noticed, in my compose.yml I have specified the .env files for the images, for example the backend image has its own env file which is located at infra\backend\prod.env

```yaml
services:
  backend:
    container_name: backend
    restart: always
    build:
      context: ../backend
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    depends_on:
      - grafana
      - prometheus
      - alertmanager
      - gatling
    env_file:
      - backend/prod.env
```

Then the upload of credentials are done via GitHub Actions using Ansible, copying the GitHub secrets and writing them to the VM we are deploying to.

Another example for this is Prometheus Alert Manager credentials

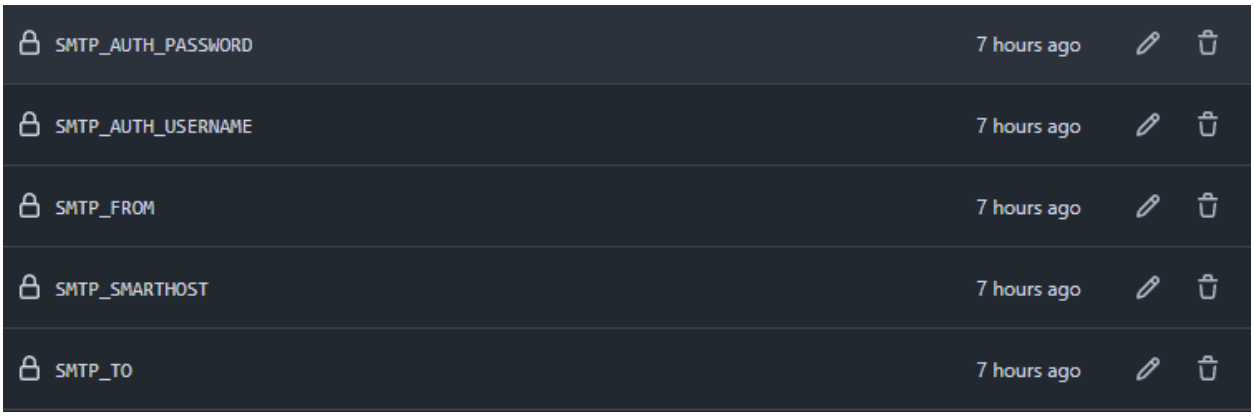


This is also managed by Ansible and GitHub actions secrets, this alertmanager.yml is so called env file for the alert manager.

As I already mentioned, infrastructure is automatically deployed by infra-deploy.yml in GitHub Actions, which also runs Ansible automatically and creates inventory file for it.

It passes secret arguments and Ansible deploys alertmanager.yml file on the server itself, without leaking the credentials.

```
- name: Install Ansible
  run: pip install ansible

- name: Create SSH key file
  run: |
    echo "${{ secrets.GCP_VM_SSH_KEY }}" > id_rsa
    chmod 600 id_rsa

- name: Create Ansible inventory file
  run: |
    echo "[gcp]" > inventory
    echo "${{ secrets.GCP_VM_HOST }} ansible_user=${{ secrets.GCP_VM_USER }} ansible_ssh_private_key_file=$(pwd)/id_rsa" >> inventory

- name: Run Ansible Playbook
  env:
    ANSIBLE_HOST_KEY_CHECKING: 'False'
  run: |
    ansible-playbook -i inventory infra/ansible/playbook.yml \
      -e "smtp_smarthost=${{ secrets.SMTP_SMARTHOST }}" \   Giorgi...gnidze, Today • upload smtp secret for alertmanager
      -e "smtp_from=${{ secrets.SMTP_FROM }}" \
      -e "smtp_auth_username=${{ secrets.SMTP_AUTH_USERNAME }}" \
      -e "smtp_auth_password=${{ secrets.SMTP_AUTH_PASSWORD }}" \
      -e "smtp_to=${{ secrets.SMTP_TO }}"
```



Therefore, the credentials and secrets are managed by .env files, GitHub Secrets and Ansible.

8. Incident Simulation + Post-Mortem

Post mortem file and discussions is attached with related screenshots.

For the incident simulation part I have taken down backend container manually, another option could've been stress loading backend with Gatling until it would shut down automatically, and could've had automatic restart policy for Docker but that would also kill my PC. By default I have restart: auto so if backend went down on itself it would've restarted

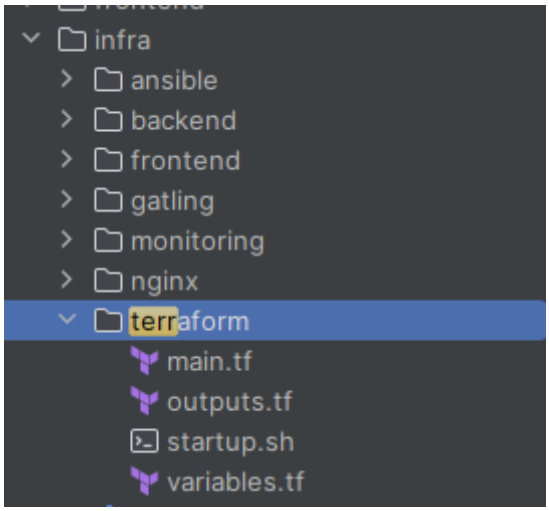
9. Automation (Ansible, Terraform, GitHub Actions)

As per bonus, it was suggested to use **Ansible**, which I have used in many ways. As I've discussed with the head of the class (course), I have deploy the entire application to the cloud environment, on GCP VM.

I have provisioned the infrastructure with **Terraform** and automated deployment with **Ansible** and **GitHub Actions**

Terraform

As I have already mentioned, under my /infra is everything that the local/VM infrastructure needs. Under Terraform folder I have defined the script for provisioning an instance on GCP and allowing multi-port traffic on this VM. Also I have multiple variables here, such as VM's name, machine type, Region, Zone, Service Account Key, etc…

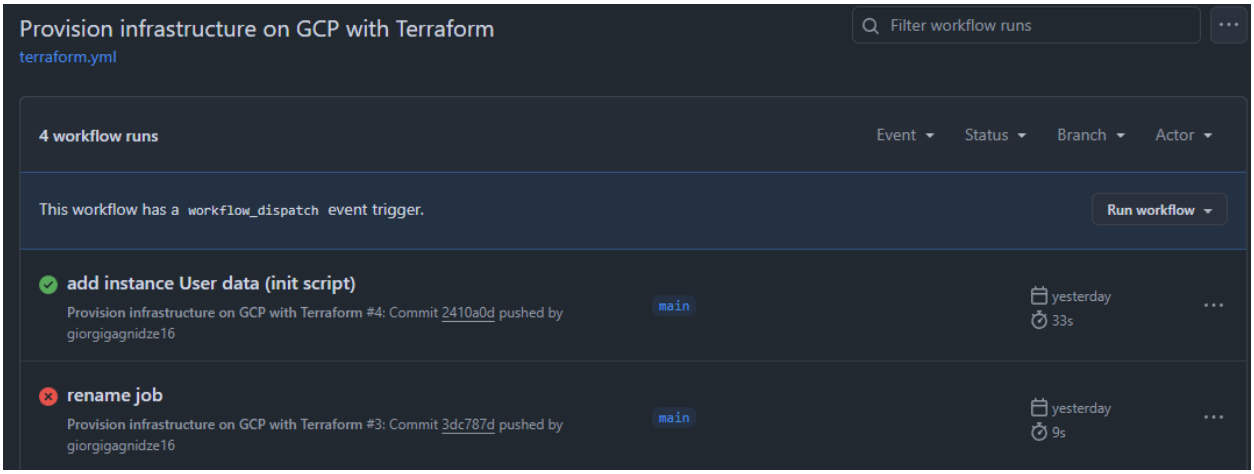



```
provider "google" {  Giorgi.Gagnidze
  credentials = file(var.gcp_credentials_file)
  project     = var.project_id
  region      = var.region
  zone        = var.zone
}

resource "google_compute_instance" "app" {  1 usage  Giorgi.Gagnidze
  name         = var.vm_name
  machine_type = var.machine_type
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = var.disk_image
    }
  }
```

This main.tf is run by GitHub Actions workflow, terraform.yml, which only runs whenever something changes under the following path “infra/terraform/” or it can also be run manually from GitHub. Here are the steps, they’re pretty straightforward:

```yaml
- name: Set up Terraform
  uses: hashicorp/setup-terraform@v3

- name: Write GCP credentials file
  run: |
    echo '${{ secrets.GCP_CREDENTIALS }}' > gcp-key.json

- name: Write SSH pubkey
  run: |
    echo '${{ secrets.TF_SSH_PUB_KEY }}' > id_rsa.pub

- name: Terraform Init
  run: terraform init

- name: Terraform Apply
  run: |
    terraform apply -auto-approve \
      -var="gcp_credentials_file=gcp-key.json" \
      -var="project_id=${{ secrets.GCP_PROJECT_ID }}" \
      -var="region=europe-west3" \
      -var="zone=europe-west3-a" \
      -var="ssh_user=deployer" \
      -var="ssh_pub_key=id_rsa.pub"

- name: Show Output IP
  run: terraform output instance_ip
```

As you see, most of the credentials are taken from GitHub secrets. A successful run for this action can be seen on my repository URL, under the actions section.

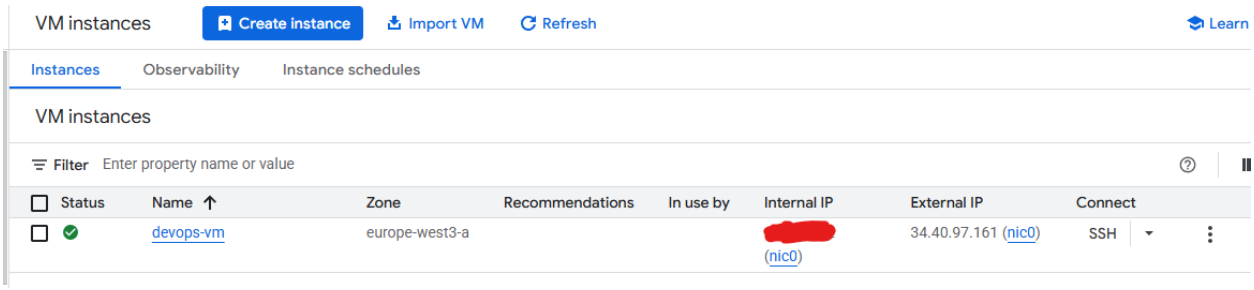


```
129
130 Plan: 2 to add, 0 to change, 0 to destroy.
131
132 Changes to Outputs:
133   + instance_ip = (known after apply)
134 google_compute_firewall.default-allow-ssh-http-https: Creating...
135 google_compute_instance.app: Creating...
136 google_compute_firewall.default-allow-ssh-http-https: Still creating... [00m10s elapsed]
137 google_compute_instance.app: Still creating... [00m10s elapsed]
138 google_compute_firewall.default-allow-ssh-http-https: Creation complete after 12s [id=projects/***/global/firewalls/default-allow-ssh-http-
    https]
139 google_compute_instance.app: Creation complete after 19s [id=projects/***/zones/europe-west3-a/instances/devops-vm]
140
141 Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
142
143 Outputs:
144
145 instance_ip = "34.40.124.152"
```

Show Output IP 0s

```
1 ▶ Run terraform output instance_ip
6 "34.40.124.152"
```

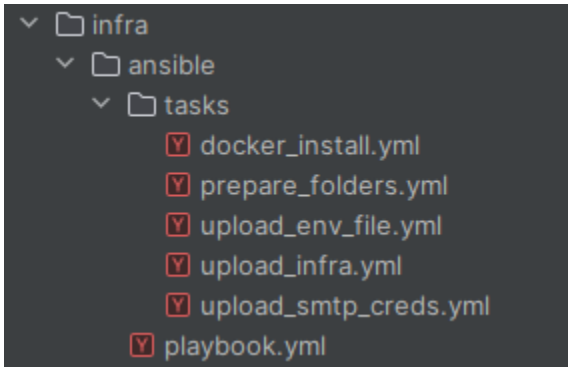
And there is my instance, the IP in the both images are different since the public IP is not static in this case and changes after every stop-start.



Now, we have seen I have used Terraform to provision the infra on GCP, Let's get back to Ansible and how I have used it.

Ansible

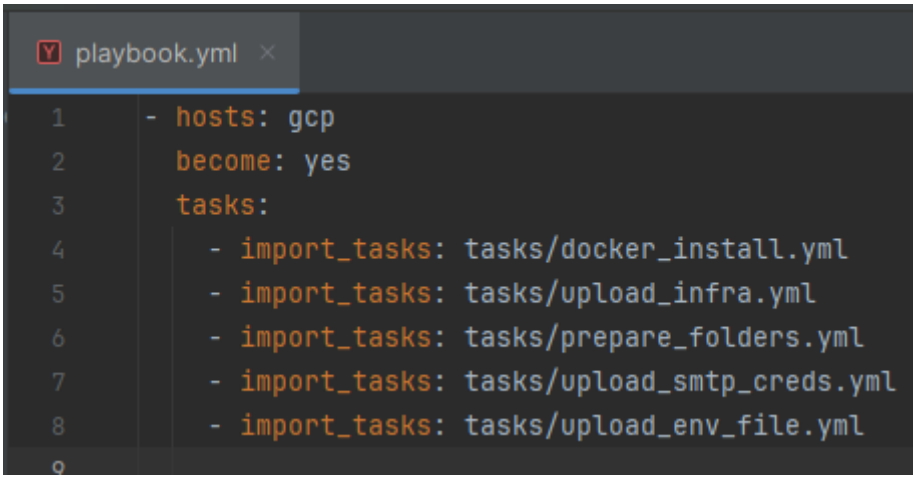
I have excluded Terraform folder from infra deployment on cloud, since the tf files are not really needed on the remote VM, we only need to provision Docker, make sure deployment folders exists and so on. That's what I have used Ansible for.



Under Ansible folder, I have main playbook.yml file for running Ansible tasks.

Tasks are for:

- Docker installation
- Preparing deployment folders
- Upload infra folder with compose to the VM
- Upload env files and credentials to the VM with help of GitHub secrets



Let’s skip Docker install part, since its most complicated one out of the 3 and more unrelated to our current domain, its Debian related.

Upload Infra:

Uploads everything under /infra folder to remote host, excluding terrafor, ansible and .git

My GCP deployment folder is /tmp/deployment

upload_infra.yml

```yaml
- name: Ensure rsync is installed
  ansible.builtin.apt:
    name: rsync
    state: present
    update_cache: yes

- name: delete infra directory
  ansible.builtin.file:
    path: /tmp/deployment/infra
    state: absent

- name: recreate an empty infra directory   Giorgi.Gagnidze,
  ansible.builtin.file:
    path: /tmp/deployment/infra
    state: directory
    mode: '0755'
    owner: "{{ ansible_user }}"

- name: Copy all folders except terraform
  ansible.builtin.synchronize:
    src: "{{ playbook_dir }}/.."
    dest: /tmp/deployment/infra/
    rsync_opts:
      - "--exclude=terraform"
      - "--exclude=.git"
      - "--exclude=ansible"
```

Prepare Folders:

This ensures that the backend and frontend directories exist, upon deployment the directories are emptied and created again to avoid any inconsistencies between versions and etc…

prepare_folders.yml

```yaml
- name: delete backend directory
  ansible.builtin.file:
    path: /tmp/deployment/backend
    state: absent
    Giorgi.Gagnidze, Today • create deployment folders on
- name: recreate an empty backend directory
  ansible.builtin.file:
    path: /tmp/deployment/backend
    state: directory
    mode: '0755'
    owner: "{{ ansible_user }}"

- name: delete frontend directory
  ansible.builtin.file:
    path: /tmp/deployment/frontend
    state: absent

- name: recreate an empty frontend directory
  ansible.builtin.file:
    path: /tmp/deployment/frontend
    state: directory
    mode: '0755'
    owner: "{{ ansible_user }}"
```

Upload SMTP Creds:

Uploads alertmanager.yml to the remote host after GitHub Actions passes the arguments down to the playbook.yml

upload_smtp_creds.yml

```yaml
- name: upload alert manager creds env file
  ansible.builtin.copy:
    dest: /tmp/deployment/infra/monitoring/alertmanager/alertmanager
    content: |
      global:
        smtp_smarthost: '{{ smtp_smarthost }}'
        smtp_from: '{{ smtp_from }}'   Giorgi.Gagnidze, Today • upl
        smtp_auth_username: '{{ smtp_auth_username }}'
        smtp_auth_password: '{{ smtp_auth_password }}'
        smtp_require_tls: true

      route:
        receiver: 'email-alerts'

      receivers:
        - name: 'email-alerts'
          email_configs:
            - to: '{{ smtp_to }}'
              send_resolved: true
    owner: "{{ ansible_user | default('deployer') }}"
    mode: '0600'
```

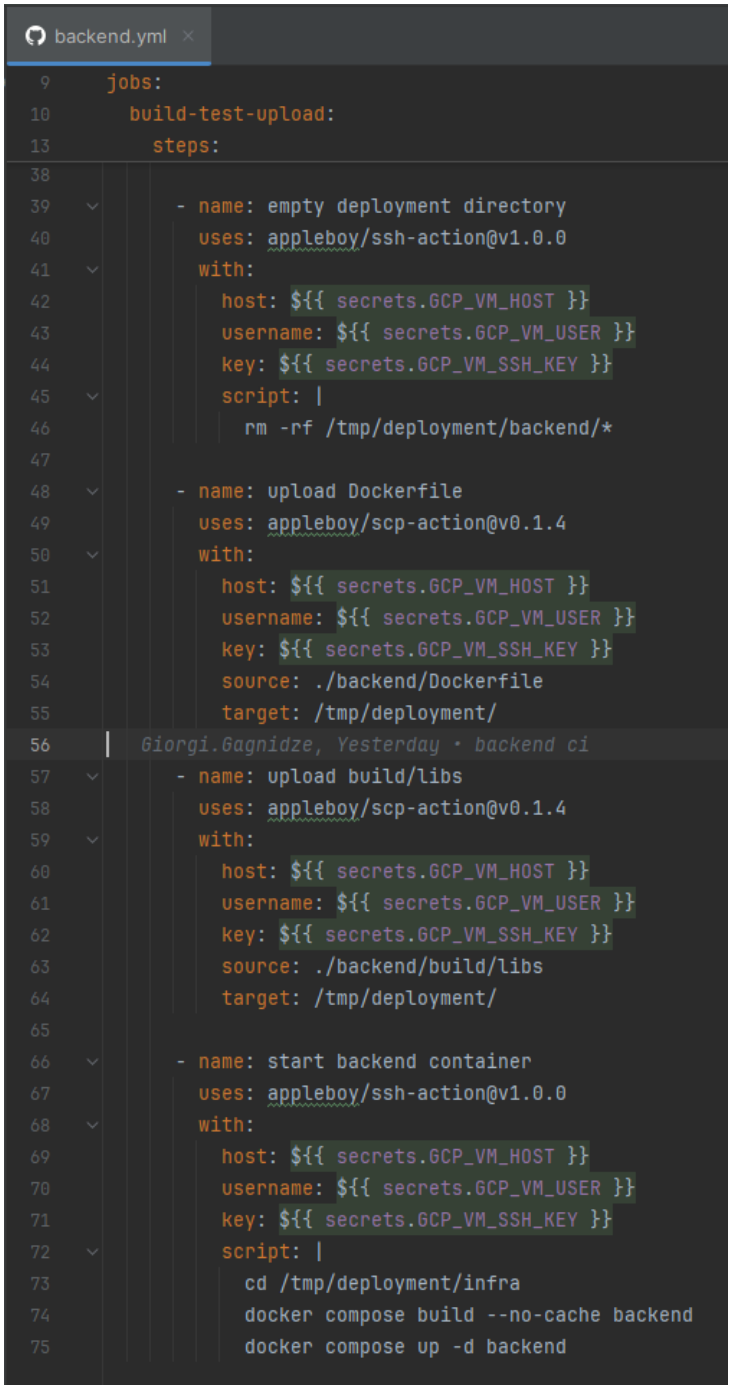
GitHub Actions + Ansible

Infra-deploy.yml is triggered when a new change is committed under infra/ansible/ directory. Ansible is installed on the runner, GCP key is written on the id_rsa file, passed down to inventory and finally Ansible Playbook is run with provided secret arguments.

```
      python-version: 3.12

    - name: Install Ansible
      run: pip install ansible

    - name: Create SSH key file
      run: |
        echo "${{ secrets.GCP_VM_SSH_KEY }}" > id_rsa
        chmod 600 id_rsa

    - name: Create Ansible inventory file
      run: |
        echo "[gcp]" > inventory
        echo "${{ secrets.GCP_VM_HOST }} ansible_user=${{ secrets.GCP_VM_USER }} ansible_ssh_private_key_file=$(pwd

    - name: Run Ansible Playbook
      env:
        ANSIBLE_HOST_KEY_CHECKING: 'False'
      run: |
        ansible-playbook -i inventory infra/ansible/playbook.yml \
          -e "smtp_smarthost=${{ secrets.SMTP_SMARTHOST }}" \
          -e "smtp_from=${{ secrets.SMTP_FROM }}" \
          -e "smtp_auth_username=${{ secrets.SMTP_AUTH_USERNAME }}" \
          -e "smtp_auth_password=${{ secrets.SMTP_AUTH_PASSWORD }}" \
          -e "smtp_to=${{ secrets.SMTP_TO }}"
```

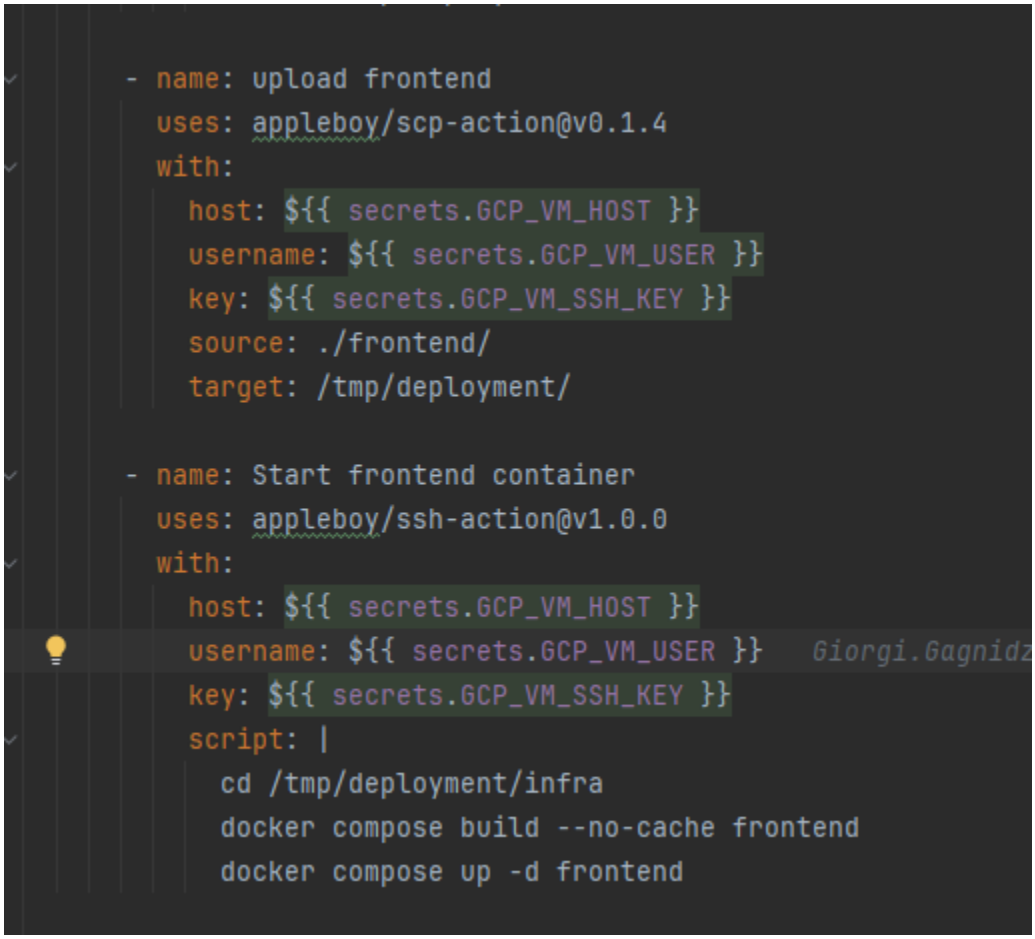
Backend Deploy

There’s an entire pipeline for backend deploy, Unit tests need to pass, linting checks and building jar. After that, remote backend deployment directory is emptied, Dockerfile uploaded as well as built jar and backend container is started



Frontend Deploy

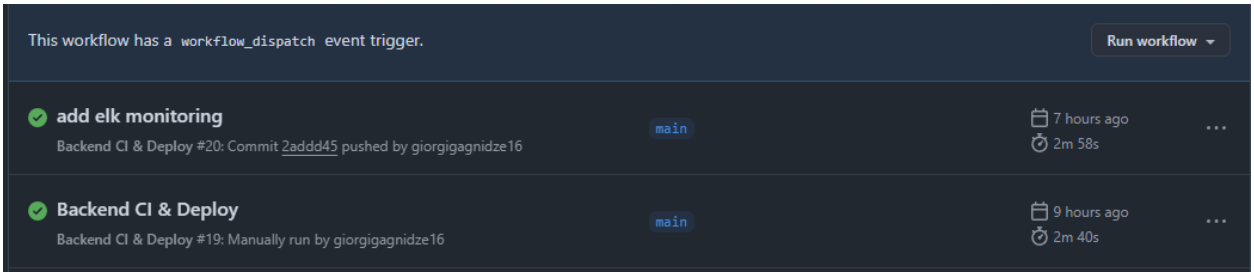
Front end folder uploaded to the remote host, image is build and container started!



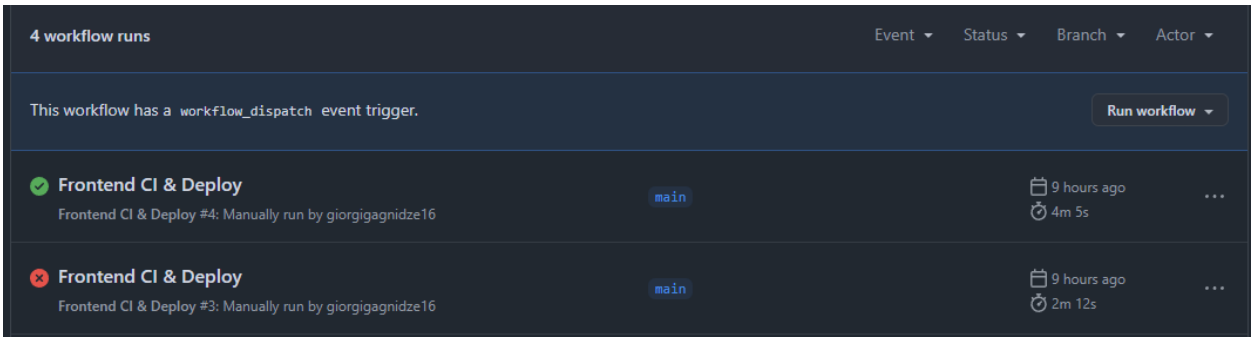
```yaml
- name: upload frontend
  uses: appleboy/scp-action@v0.1.4
  with:
    host: ${{ secrets.GCP_VM_HOST }}
    username: ${{ secrets.GCP_VM_USER }}
    key: ${{ secrets.GCP_VM_SSH_KEY }}
    source: ./frontend/
    target: /tmp/deployment/

- name: Start frontend container
  uses: appleboy/ssh-action@v1.0.0
  with:
    host: ${{ secrets.GCP_VM_HOST }}
    username: ${{ secrets.GCP_VM_USER }}   Giorgi.Gagnidz
    key: ${{ secrets.GCP_VM_SSH_KEY }}
    script: |
      cd /tmp/deployment/infra
      docker compose build --no-cache frontend
      docker compose up -d frontend
```

In compose file itself, backend and frontend are both dependent on other services, and their startup also triggers other services to start.

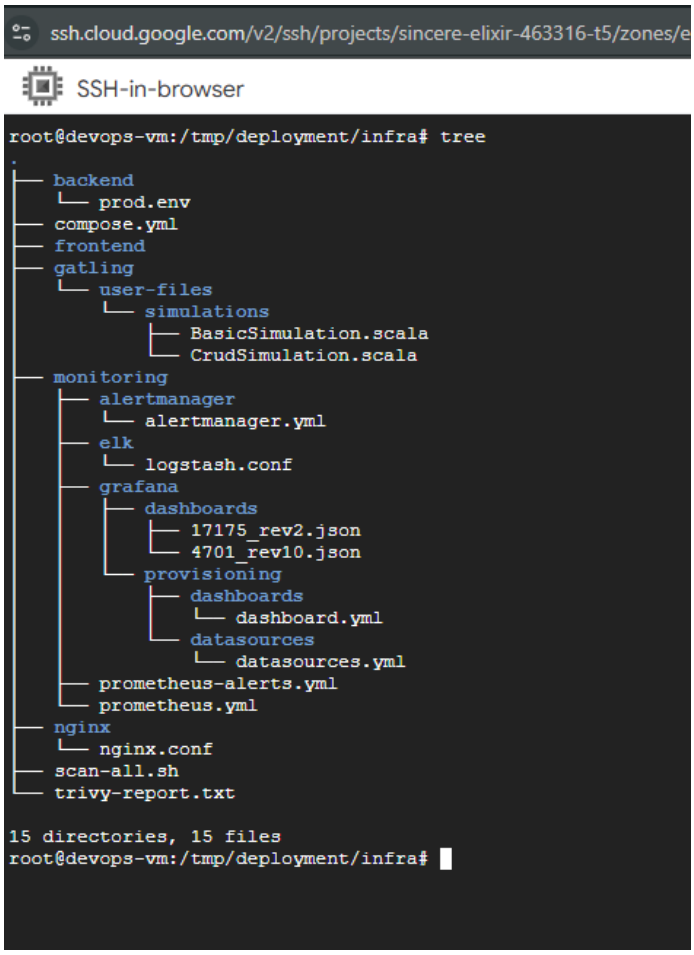
Backend Deployment:



Frontend Deployment:

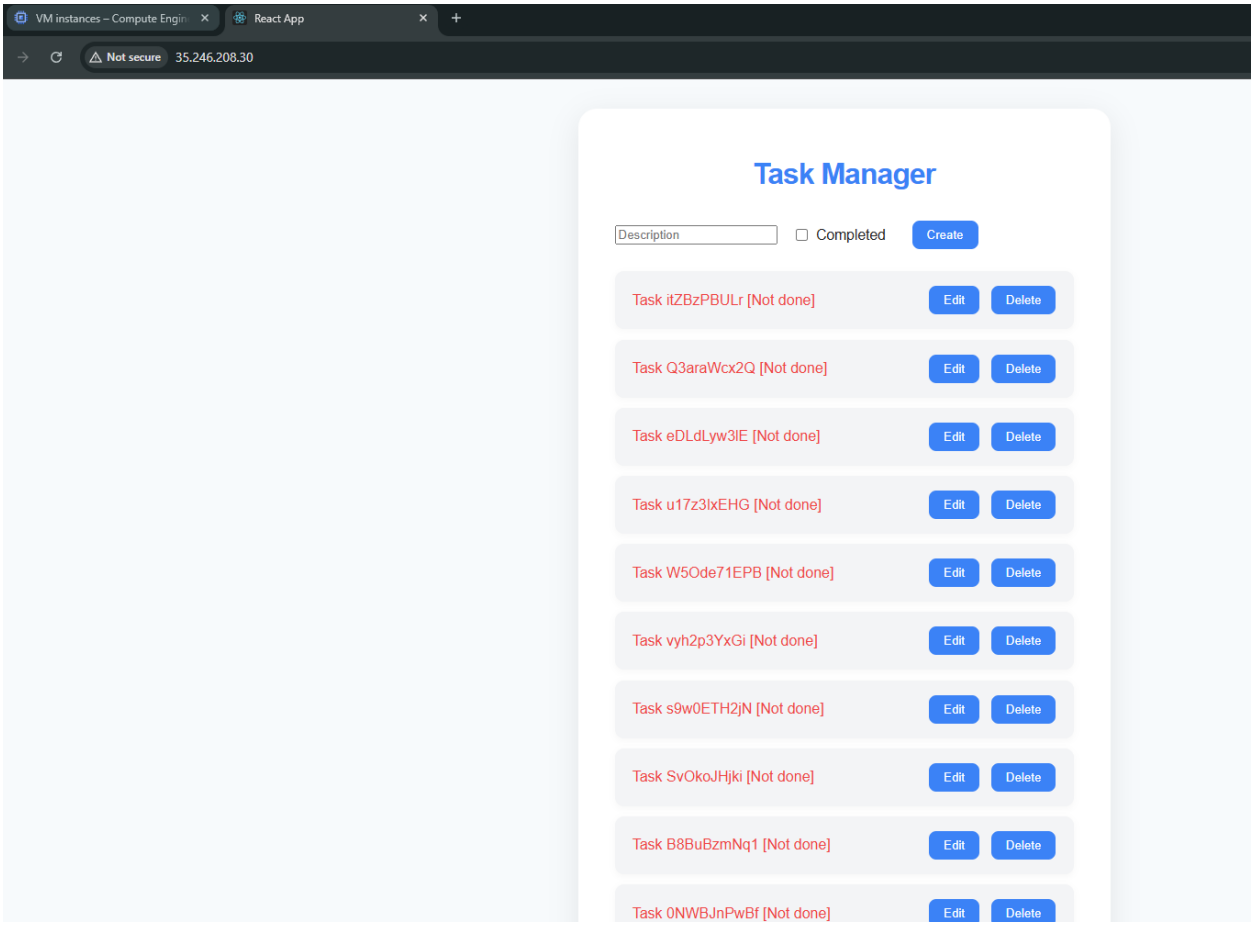


The remote host machine after deploying infra using Ansible: As we see all the infra folder is uploaded

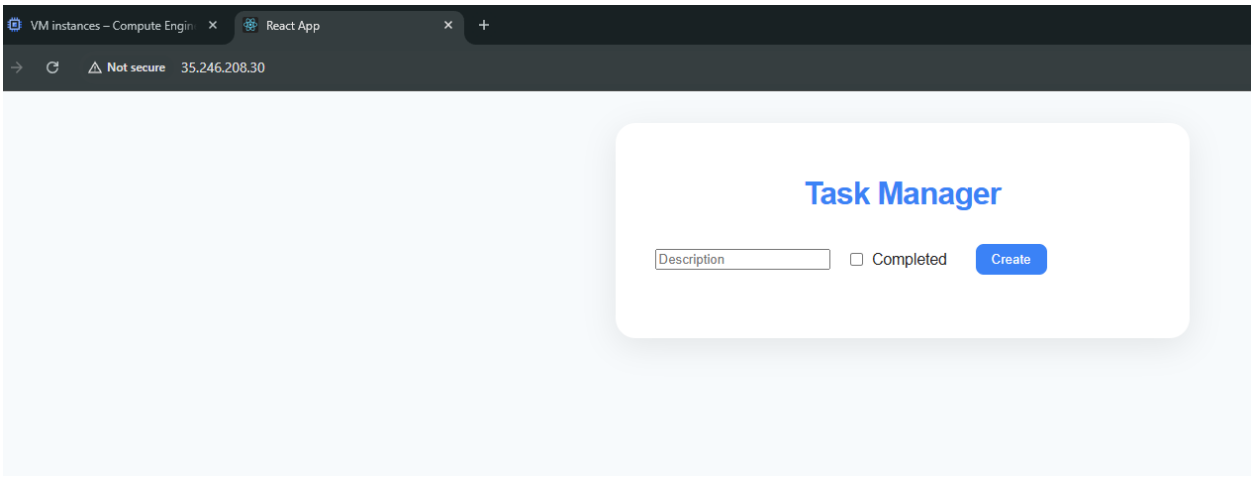


```
root@devops-vm:/tmp/deployment# ls
backend  frontend  infra
root@devops-vm:/tmp/deployment# ls backend/
Dockerfile  build
root@devops-vm:/tmp/deployment# ls frontend/
Dockerfile  README.md  build  node_modules  package-lock.json  package.json  public  src  tsconfig.json
root@devops-vm:/tmp/deployment#
```

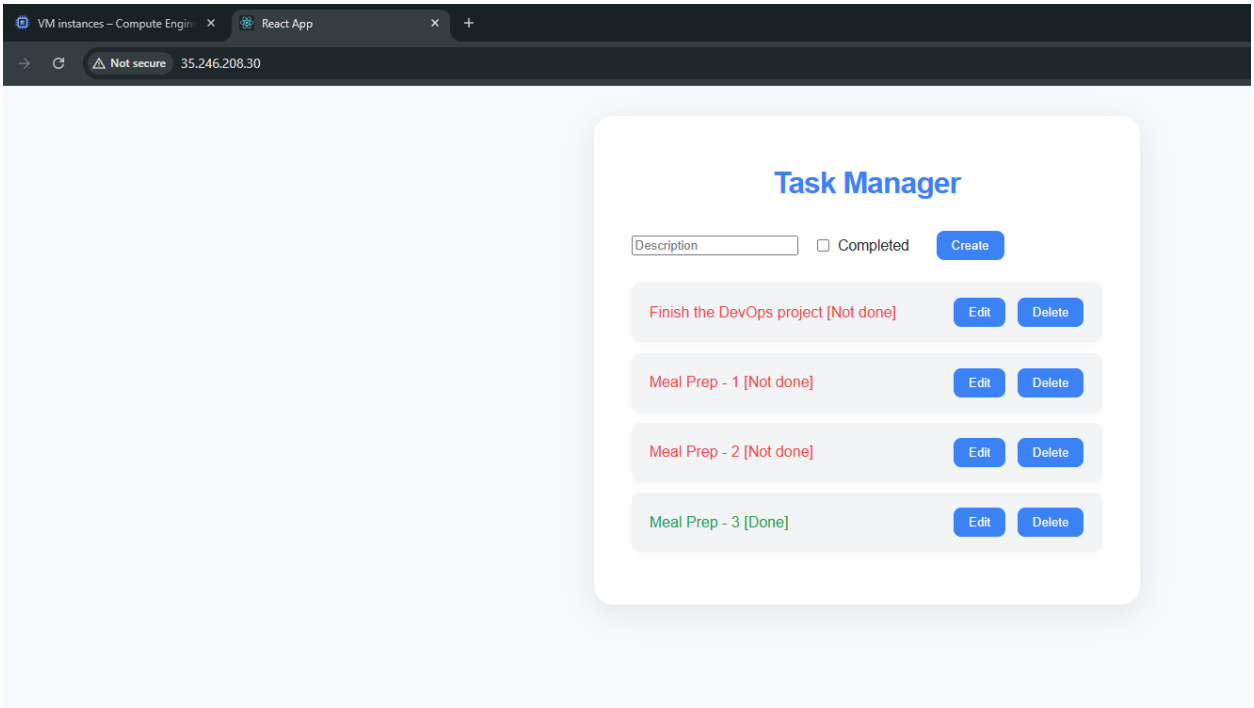
Let’s make sure instance is running all the services smoothly. Live Demo:



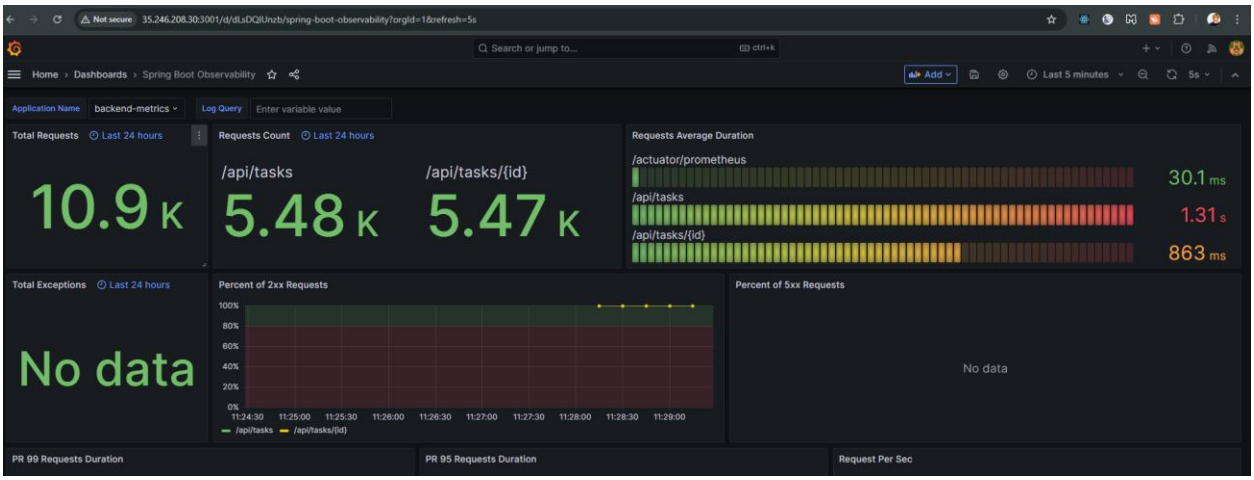
As we see these are tasks created by Gatling simulation, I was lucky I had the opportunity to see this actually since I simulated full CRUD and after deletions, we cant see the tasks obviously.



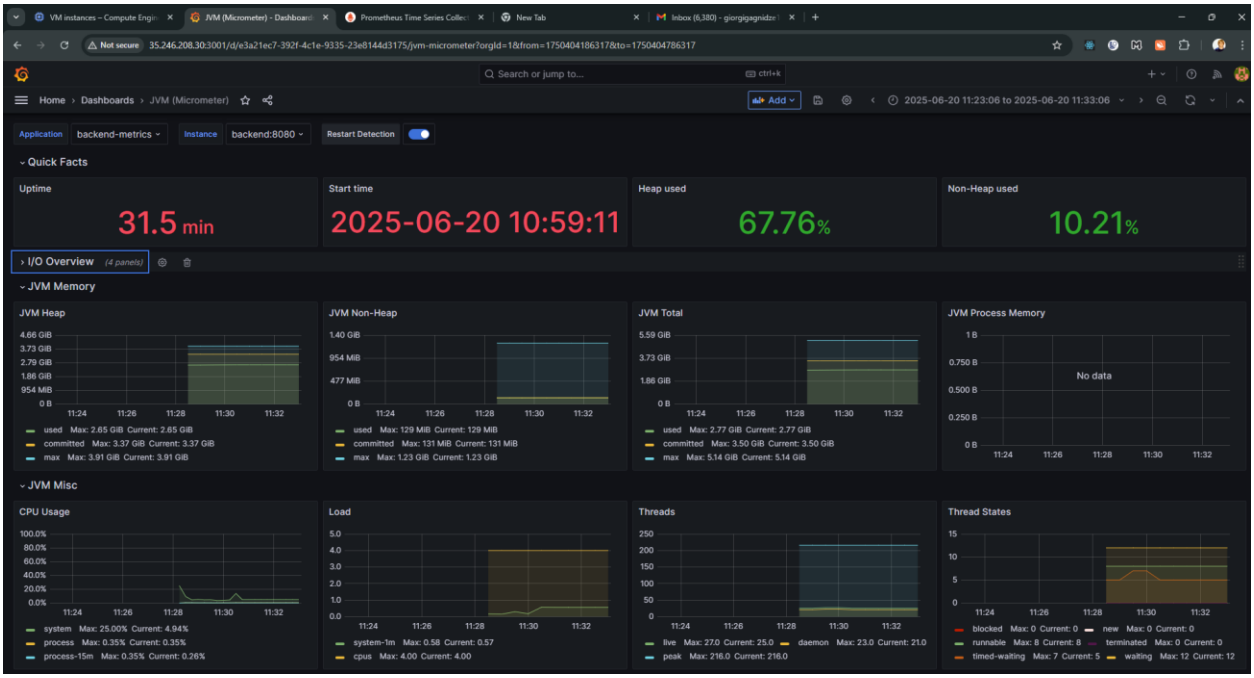
After refresh, they’re gone and we can see that this thing is fully functional too.



We can also check Grafana, ELK and everything

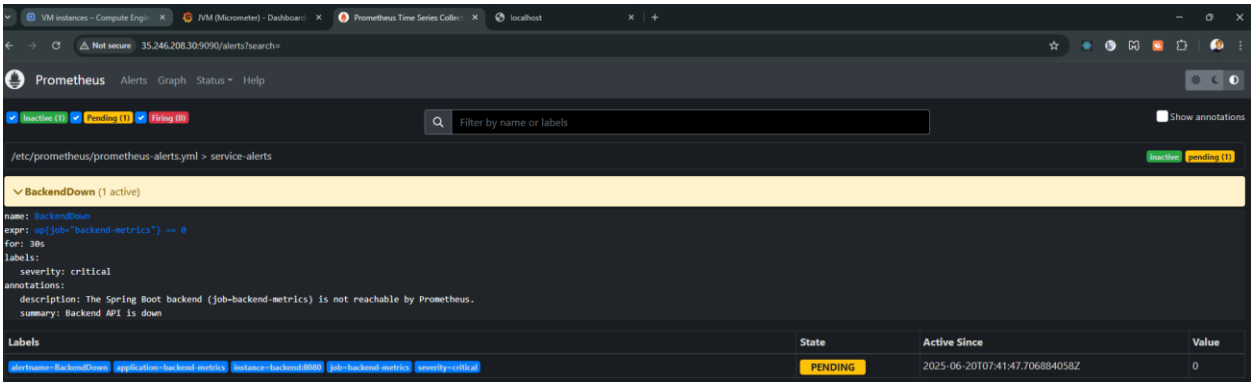


As you see there’s a lots of requests from Gatling due to me restarting Gatling manually.



Ok, now let's try to take down the backend service and we'll see that Prometheus responds accordingly

```
prometheus
root@devops-vm:/tmp/deployment/infra/monitoring# docker compose down backend
[+] Running 2/2
 ✓ Container backend                    Removed
 ! Network devops-final-prj_default     Resource is still in use
root@devops-vm:/tmp/deployment/infra/monitoring#
```



Well, that’s it. I have utilized every tool described in the task, applying the best practices possible. Additionally did the complete CI/CD deployment of an application to the GCP cloud for which I used Terraform, to get an instance VM, which would handle many Docker images, and load that Gatling was putting on the backend as well as 16GB or ram. Additionally I have used ELK & Prometheus Alerts for monitoring, GitHub Actions for automated deployment and Ansible for infrastructure provisioning (package installations, folder creation, updates, etc…)