GIC 2021-22

Projeto 3

Autores: **Bruno Bastos nºmec 93302,**
Mário Silva nºmec 93430,
Miguel Fazenda nºmec 110877

Date: **23/06/2022**

Index

1 Introduction

This report aims to describe the work done for the monitoring project on the subject of GIC. The objective of this work was to deploy an existing product in a scalable environment using Kubernetes while monitoring the system. The report describes the process of how it was deployed and the strategies used for monitoring.

2 Product

Misago is a fully-featured internet forum solution developed in accordance with practices and trends currently used in web software development.

The main goal of this product is to provide an open-source application, easily customizable and integrable, where users can discuss and follow topics of interest. It allows users to create categories with an unlimited number and depth of subcategories. Users can start their own threads and comment, make a post or create a pool in any of the other available threads. Mark post in question thread as the best answer, bringing basic QA functionality which can be helpful when integrating with an already existing product, especially in the software development area.

Admins manage and moderate the forum and have access to privileged functionalities for that.

3 Software stack

- **Django** - the Django web server handles the requests and serves the content to the user. It responds to the Rest HTTP requests and also serves the dynamic files using Django templates. The templates are rendered with the help of React compiled files.
- **PostgreSQL** - the main database of the application runs on PostgreSQL. The Django app connects to the database and performs queries, whether it is for data storage or retrieval.
- **Redis** - the Django app has some functionalities which are required to run in the background so that it doesn't slow down the application. Redis serves as a message broker between the Django app and the Celery workers that will perform the tasks specified by Django.
- **Celery** - the background tasks are performed by celery workers. Only a few sets of admin tasks depend on the celery worker.

Figure 1 shows the Software Stack of Misago application.

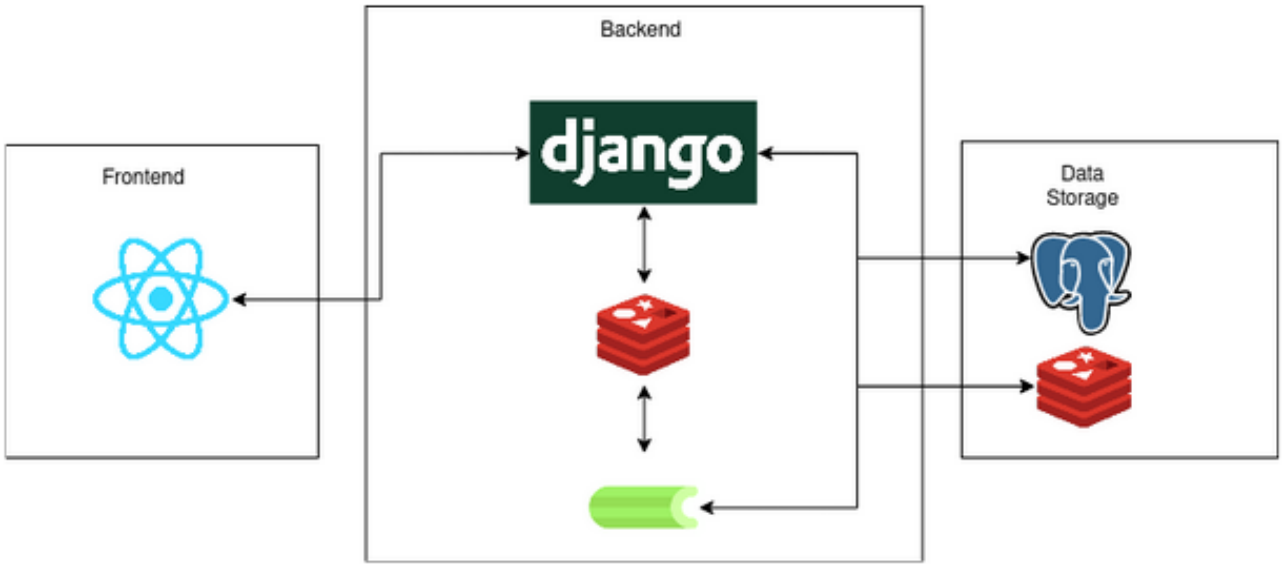

Figure 1: Misago Software Stack.

4 Deployment Strategy

The overall deployment strategy follows multiple steps to ensure the application runs as intended. Each step concerns the deployment of technology from the stack:

- Databases (Redis and PostgreSQL), which are initialized when Misago is deployed;
- Misago creates databases and serves the main app;
- Celery accepts tasks from Misago and executes them;
- Nginx, Load-Balancer to direct requests to the intended app/resource;

Figure 1 shows the Deployment Architecture.

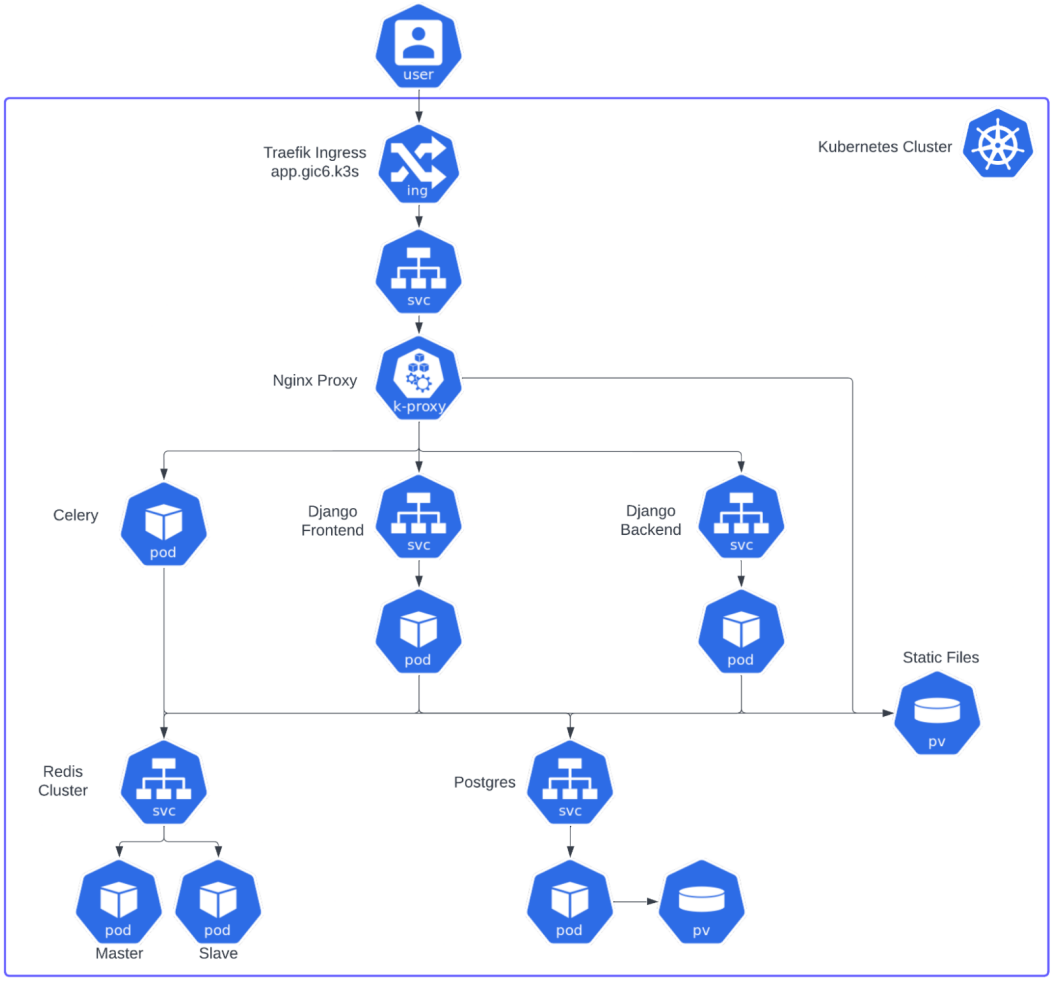

Figure 2: Deployment Architecture.

4.1 Ingress

The ingress uses Traefik and routes the requests to a Nginx service. It only accepts HTTP requests and runs on the host app.gic6.k3s.

Figure 3 is a code sample of the ingress configuration file.

```yaml
spec:
 rules:
 - host: app.gic6.k3s
   http:
     paths:
     - path: /
       pathType: Prefix
       backend:
         service:
           name: nginx
           port:
             number: 80
```

Figure 3: Ingress deployment file.

4.2 Django

The Django app was initially running using the development server, which is not scalable for bigger workloads. The first decision was to use **gunicorn** as a wsgi to handle the requests that are being made to the Django app. This has drawbacks because gunicorn cannot serve static files from Django.

The Django app also requires some **environment variables** in order to connect to the postgres database which is being provided using **secrets**.

Figure 4 is a contains the secrets for the django application.

```yaml
apiVersion: v1
kind: Secret
metadata:
 namespace: gic6
 name: postgres-secrets
type: Opaque
data:
 # Postgres
 POSTGRES_USER: bWlzYWdv
 POSTGRES_PASSWORD: bWlzYWdv
 POSTGRES_DB: bWlzYWdv
 POSTGRES_HOST: cG9zdGdyZXM=
 POSTGRES_TEST_DB: bWlzYWdvX3Rlc3Q=
 # Superuser
 SUPERUSER_USERNAME: QWRtaW4=
 SUPERUSER_EMAIL: YWRtaW5AZXhhbXBsZS5jb20=
 SUPERUSER_PASSWORD: cGFzc3dvcmQ=
```

Figure 4: Django Secrets file.

Some of the **dynamic** website pages are being sent by **Django** while the **static** contents are being provided by the **Nginx**.

The static files are stored in a folder. Both Django and the Nginx need to access this folder in order to store/retrieve the static files, so we create a shared volume that allows them both to do operations in the files. Whenever the first pod of the Django app is initialized, we enter the container and copy the files to the /static directory. This way, whenever Django adds a file to the /static directory, like a profile image, the Nginx has access to it and can serve it as a static file.

Since the directory with the volume mount path is deleted whenever it is initialized, some configurations had to be changed in the Django settings to point the static directory to the new path accordingly.

4.3 Nginx

The Nginx proxy will handle requests to the static files by accessing a shared volume that is initialized with the static files from Django.

Aside from the static files, the requests that reach the Nginx proxy will be forwarded to one of the two Django services and then to one of the pods. We decided to split the requests to the frontend and to the Rest API.

In the Nginx configurations, we set the location to change whenever there was a request made to /api, so that it would be redirected to the service that exposes the backend pods of Django. There is also a service that exposes the frontend pods of Django, that is only responsible for the template rendering.

The figure 5 presents the configuration used for the nginx deployment.

```
location /api/ {
  # proxy_pass http://app:8080/;
  proxy_set_header   X-Real-IP $remote_addr;
  proxy_set_header   Host      $http_host;
  proxy_http_version 1.1;
  proxy_set_header Connection "";
  proxy_pass http://misago-service:8000/api/;
}

location /nginx_status {
  stub_status on;
  access_log  on;
  allow all;
}

location / {
  # proxy_pass http://app:8080/;
  proxy_set_header   X-Real-IP $remote_addr;
  proxy_set_header   Host      $http_host;
  proxy_http_version 1.1;
  proxy_set_header Connection "";
  proxy_pass http://misago-frontend-service:8001/;
}

location /promopage {
  root /static/;
  index index.html;
}

location /static {
  root /;
}
```

Figure 5: Nginx Configuration file.

4.4 PostgreSQL

Django relies on a PostgreSQL database to store and retrieve data. PostgreSQL persists data by storing it in a volume. Unfortunately, we did not manage to create replicas for PostgreSQL, so if the pod goes down, the application will fail. The Postgres service will expose it to the Django pods so that they can manage the data stored.

4.5 Redis & Celery

There are some tasks that can be performed in the background by celery. The backend places the tasks in Redis and one of the celery workers will pick it up. For that, it is required to have a service to expose Redis and the Redis pods. In order to provide replication, we are using a Redis cluster with one master and two slaves. Celery does not need to be exposed by service because no other pod tries to access it, the workers are the ones accessing the Redis cluster and performing their tasks. The configurations for celery were changed to match the ones from the Redis cluster.

5 Scalability

Stateless components like the Django app and nginx can be scaled by specifying the number of pods to run in the kubernetes. There are some issues with the stateful components like redis and postgres.

As mentioned before, Redis is deployed in a cluster which allows it to run multiple slaves but since it only has one master, it creates a bottleneck in writing data to Redis.

Since in our Redis application, Redis is only used for admin tasks, it shouldn't be a problem unless there were many admins. Another bottleneck of Redis is the fact it doesn't have a sentinel, therefore, if the master fails or dies, there can be data loss and the master would have to be restarted or re-assigned.

As it was previously mentioned, we were not able to replicate Postgres, so there is only one pod running. If we managed to do that, we would have one master pod and one or two slaves. Just like Redis, that would be an issue if there were many writes to the database, but considering that the application focused on a forum, where most users are usually reading, then a slave would be very useful.

6 Automatic Deployment

The source code is stored in a Github repository. Every time there are changes made to the main branch, some workflows are executed using **Github actions**.

The first one uses the source code to build the images required for the application to run, the Django backend and the Django frontend.

If this workflow succeeds, the next workflow runs on the class VM and will execute the **run.sh** script which will update the kubernetes deployments, services and other configurations. Since the app requires the static files to be copied to the volume, the script waits for one of the backend pods to be running in order to copy the files to the shared volume location.

The next code sample has the contents of the **run.sh** script.

```
#!/bin/bash
# ingress
kubectl apply -f k3s/ingress/ingress.yaml -n gic6
# postgres
kubectl apply -f k3s/postgres/postgres-secrets.yaml -n gic6
kubectl apply -f k3s/postgres/postgres-pvc.yaml -n gic6
kubectl apply -f k3s/postgres/postgres-statefulset.yaml -n gic6
```

```
kubectl apply -f k3s/postgres/postgres-service.yaml -n gic6
# redis
kubectl apply -f k3s/redis/redis-config.yaml -n gic6
kubectl apply -f k3s/redis/redis-statefulset.yaml -n gic6
kubectl apply -f k3s/redis/redis-service.yaml -n gic6
# celery
kubectl apply -f k3s/celery/celery-deployment.yaml -gic6
# backend
kubectl apply -f k3s/misago/misago-config.yaml -n gic6
kubectl apply -f k3s/misago/misago-service.yaml -n gic6
kubectl apply -f k3s/misago/misago-deployment.yaml -n gic6
# frontend
kubectl apply -f k3s/frontend/misago-frontend-service.yaml -n gic6
kubectl apply -f k3s/frontend/misago-frontend-deployment.yaml -n gic6
# nginx
kubectl apply -f k3s/nginx/exporter-config.yaml -n gic6
kubectl apply -f k3s/nginx/nginx-config.yaml -n gic6
kubectl apply -f k3s/nginx/nginx-pvc.yaml -n gic6
kubectl apply -f k3s/nginx/nginx-service.yaml -n gic6
kubectl apply -f k3s/nginx/nginx-deployment.yaml -n gic6
GET_PODS_OPTIONS='--no-headers -o custom-columns=:metadata.name --field-selector=status.phase==
sleep 5
NGINX=
while [ -z "$NGINX" ]
do
    NGINX=$(kubectl get pods $GET_PODS_OPTIONS -n gic6 | grep -m1 nginx)
    echo -ne "waiting for nginx to start running..."\\r
    sleep 1
done
echo "NGINX=$NGINX"
MISAGO=
while [ -z "$MISAGO" ]
do
    MISAGO=$(kubectl get pods $GET_PODS_OPTIONS -n gic6 | grep -m1 misago)
    echo -ne "waiting for misago to start running..."\\r
    sleep 1
done
echo "MISAGO=$MISAGO"
kubectl exec -n gic6 -it $MISAGO -- /bin/bash -c "mkdir -p /srv/misago/static/static && cp -r /s

kubectl port-forward --namespace gic6 svc/nginx 9010:4040 &
kubectl port-forward --namespace gic6 svc/postgres 9187:9187 &
```

In figure 6 it is possible to see the continuous deployment running in github actions.

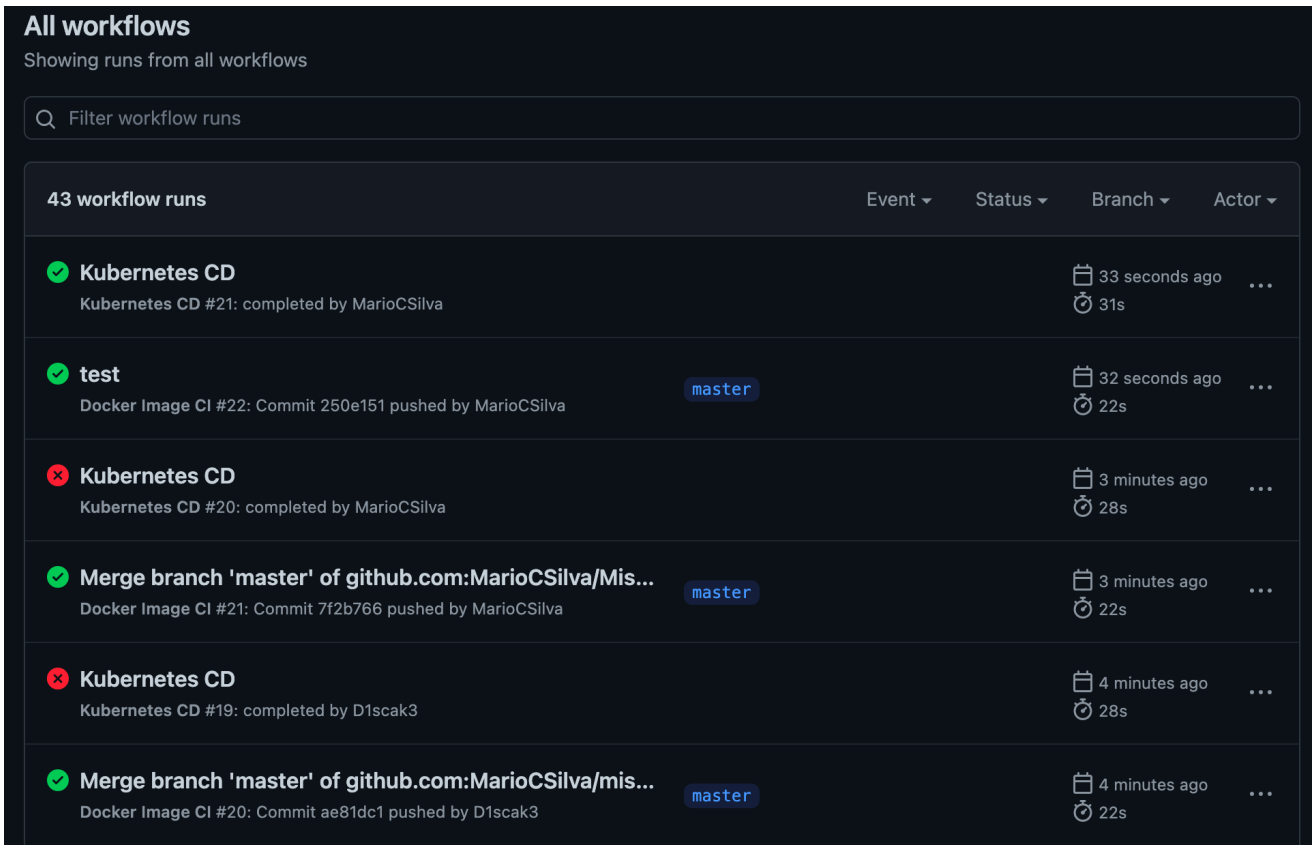

Figure 6: Run examples of the repository workflow.

7 Fault Tolerance

Kubernetes already provides some degree of fault tolerance. Most of our components can be scaled to have multiple instances running, which not only can handle more traffic, but also allows some components to crash without the application going down. The recovery of the components that crashed is also handled by the Kubernetes.

The major problems that cannot be addressed directly by Kubernetes are the data that can be lost when a volume is deleted/down and potential errors in the application itself. The first one can be addressed by making backups and duplicating the data of the database and media files. For the application errors, there can be a monitoring system that checks for errors in the requests, for example when there are a huge amount of responses with a status code of 500 it would send an alert to an email.

8 Monitoring

Monitoring is done through **Prometheus** and **Grafana**.

For **Prometheus**, there are 2 instances running:

- Prometheus-1, running in the cluster to collect metrics related to the cluster itself;
- Prometheus-2, running on Docker inside a virtual machine, that connects directly to the services of the intended component for monitoring. It connects to the following entities:
  - NGINX, made through port-forwarding a local port to the corresponding DNS service name of NGINX;
  - Postgres, made through port-forwarding a local port to the corresponding DNS service name of Postgres;

For **Grafana**, there is one instance running on Docker in a virtual machine, it establishes connections to the following entities:

- Prometheus-1, through the DNS name of the service. It's necessary to add the hostname "prometheus.deti" to the host's file;
- Prometheus-2, through localhost, since both docker containers are running inside the same virtual machine on the same network;

Figure 7 shows the architecture of the monitoring system.

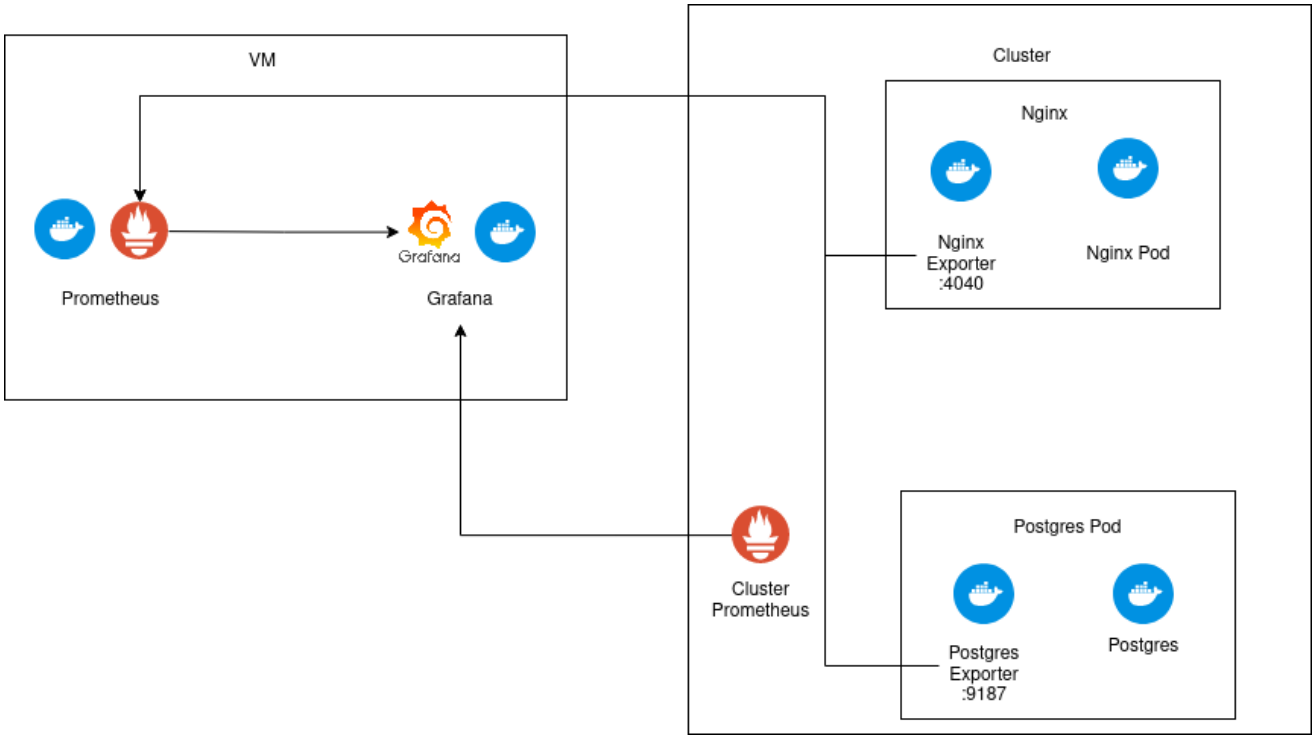

Figure 7: Monitoring architecture of the system.

8.1 Metrics

Grafana shows the collected metrics using dashboards. Following are examples of monitoring Postgres.

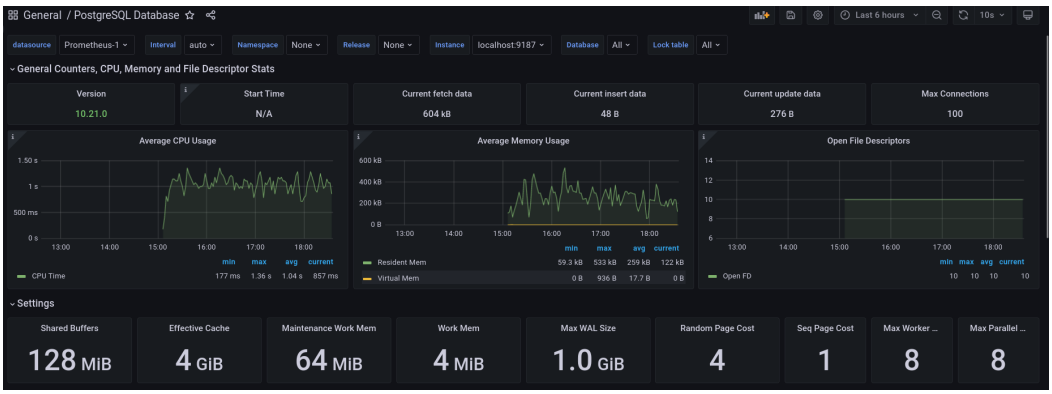

Figure 8: Postgres Dashboard 1

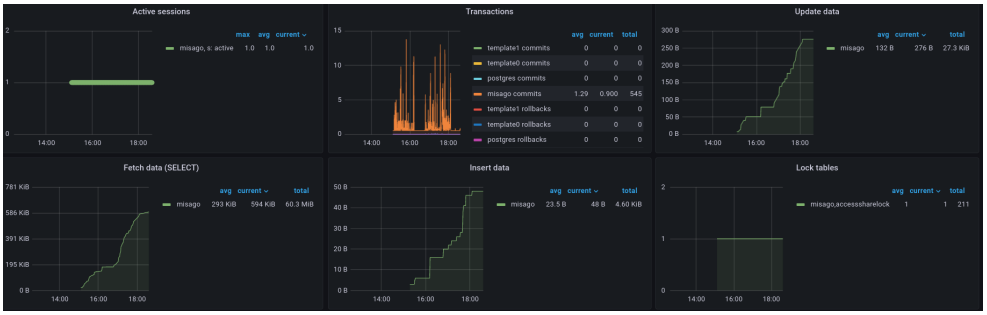

Figure 9: Postgres Dashboard 2

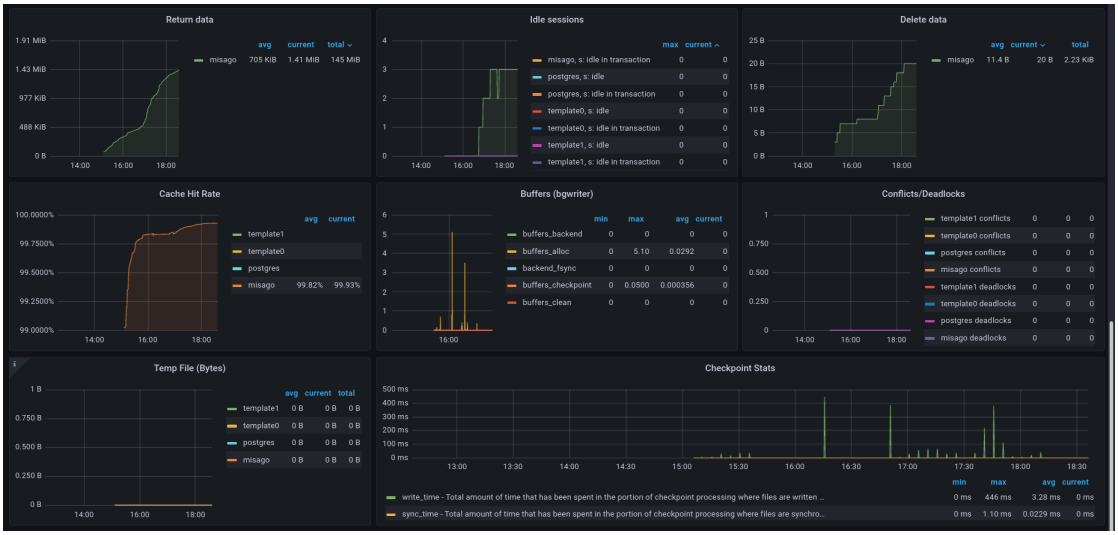

Figure 10: Postgres Dashboard 3

Besides this, we also have some specific information, such as the number of posts, users, and threads on the postgres, which allows for visualizing the evolution and growth of the database.

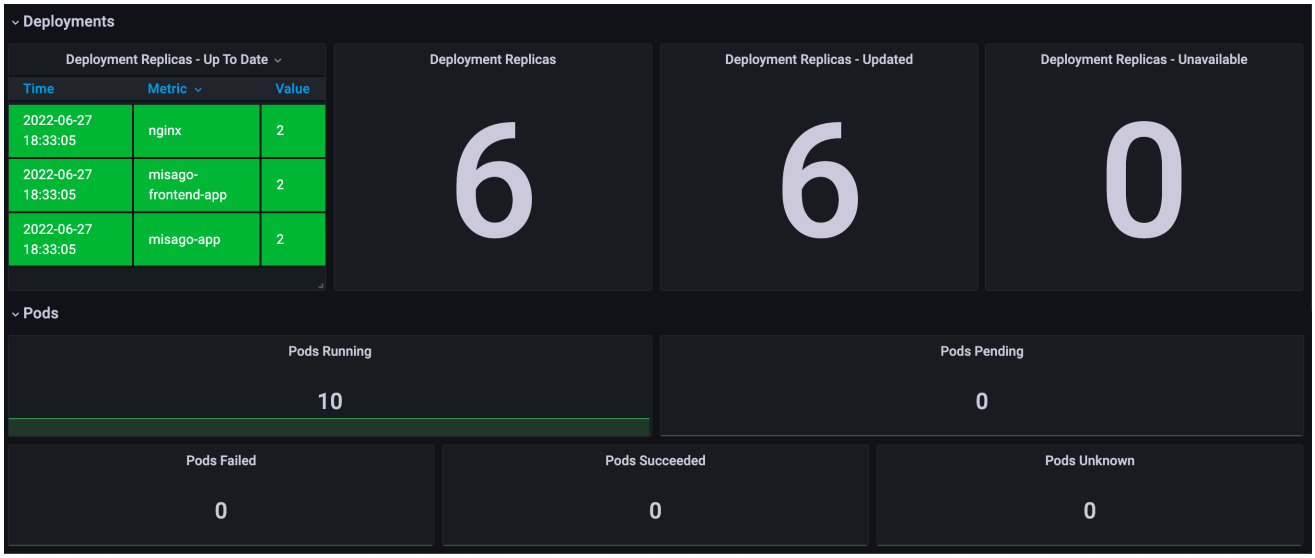

Figure 11: Postgres monitoring growth and activity

Figures 12, 13 and 14 show the dashboards from the metrics collected using the nginx exporter. The first dashboard collects metrics related to specific endpoints such as the number of failed logins, the rate of thread creation and errors in the application.

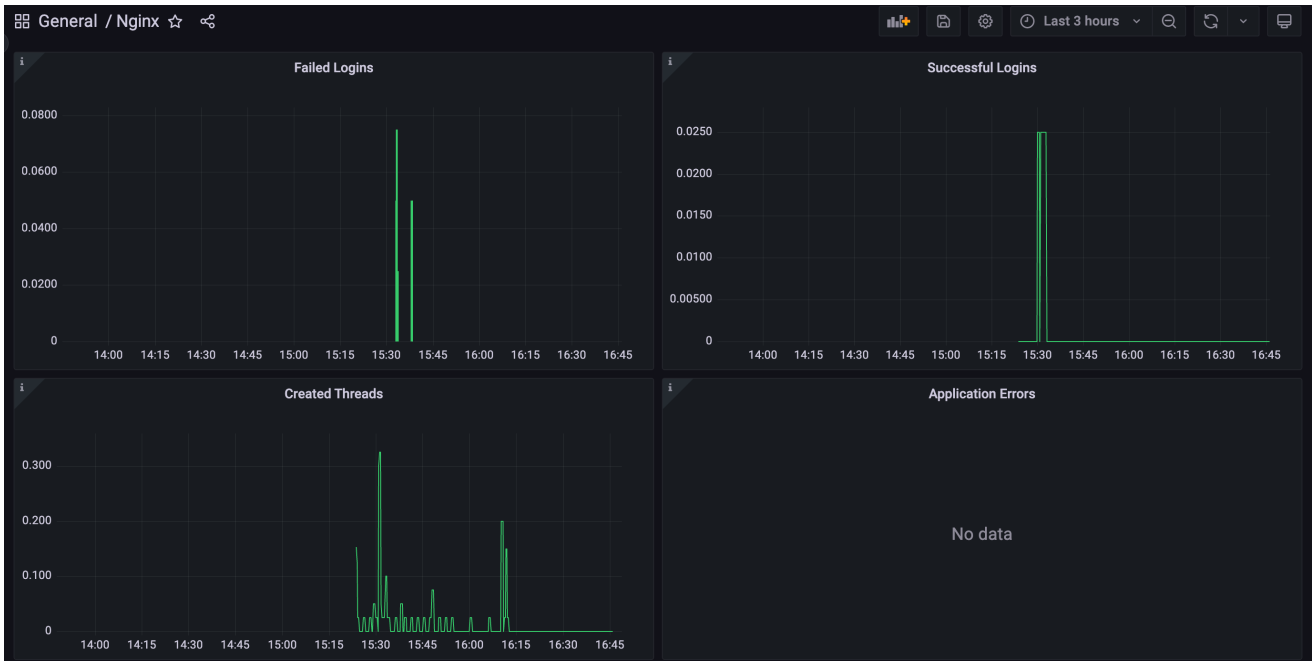

Figure 12: Nginx endpoint metrics

The second nginx dashboard displays the requests with the highest rate of usage on the application.

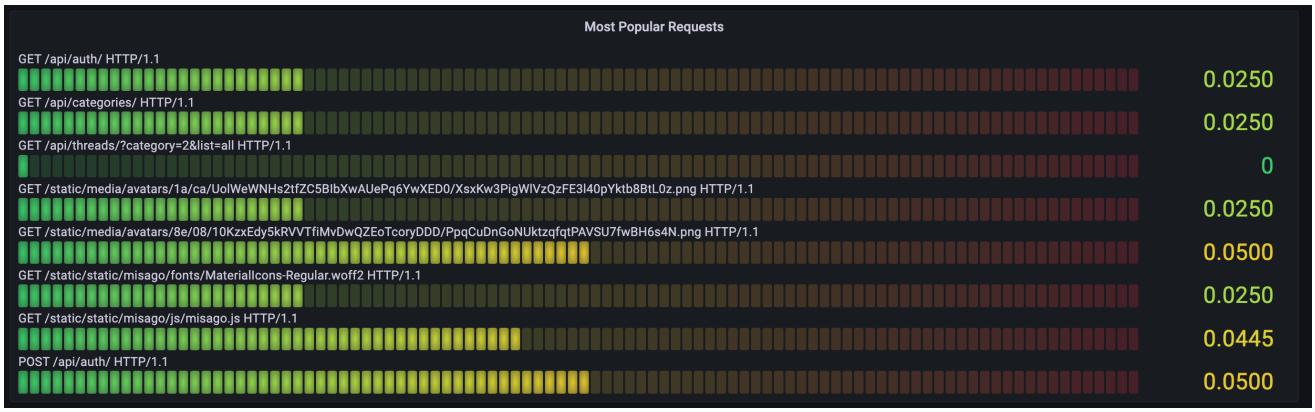

Figure 13: Nginx Most popular requests

The last dashboard uses metrics related to the traffic that goes through the application.

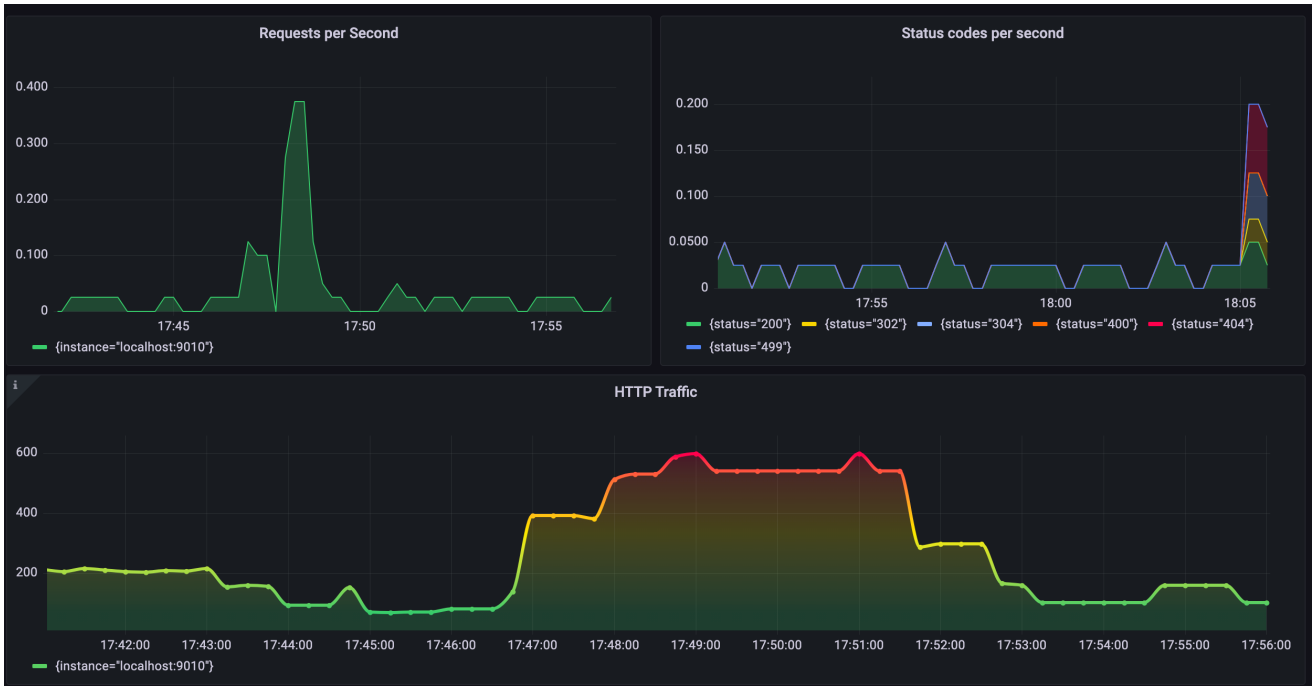

Figure 14: Nginx Traffic information

There are also some general metrics about the deployment to check how it is all operating.

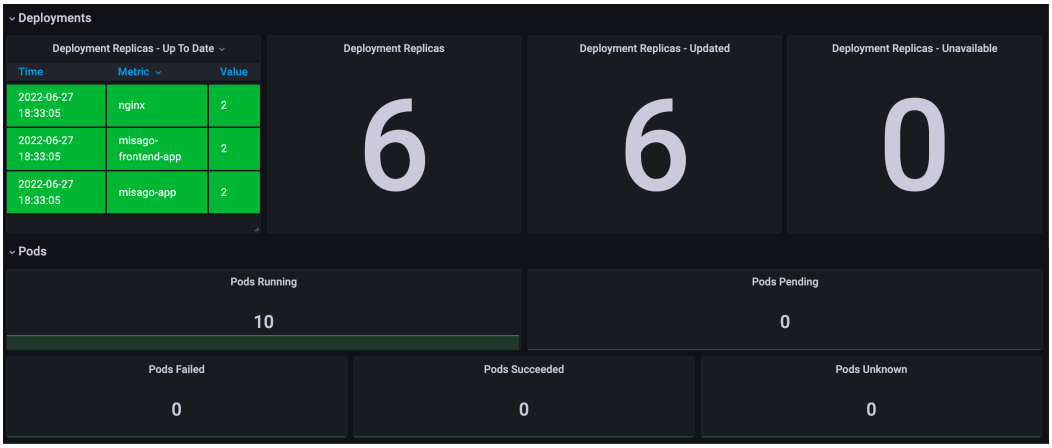

Figure 15: General Deployment Information

8.2 Alerts

In Grafana, it is possible to define alert rules to send notifications/emails to someone when there is an event that triggers that rule.

We defined rules to allow the developers to understand when there is something critical that requires human intervention. The alerts that we created send notifications to a discord server with the description of the rule.

When there is someone trying to access the admin page multiple times without success, an alert is sent to discord. Also, the application sometimes can have bugs that are not easy to detect, so there is an alarm that alerts the developers when there are many responses with an HTTP status of 500 internal server error.

Figure 16 shows an alert received in discord when someone tries to login to the admin page unsuccessfully for many tries.

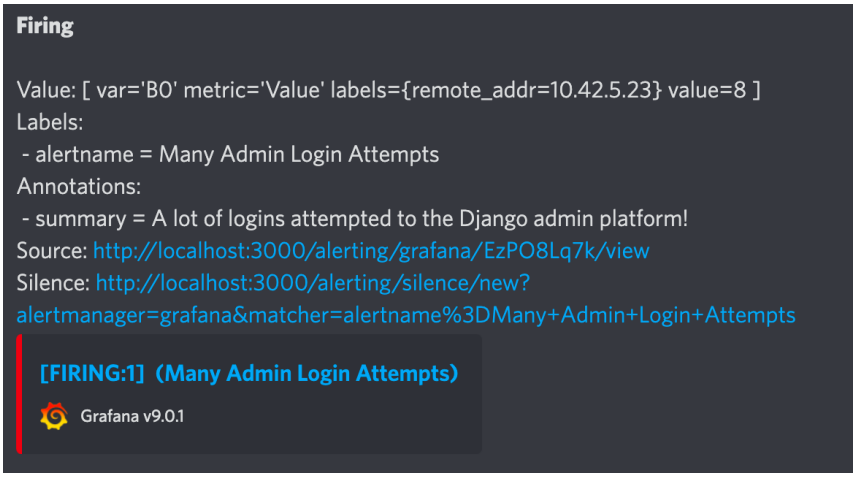

Figure 16: Alert example received in Discord

9 Issues

There were some issues during the deployment of this project, starting with the scalability of the database. PostgreSQL is a stateful component, which means that scaling requires some communication between the multiple replicas in order for data to stay consistent. We were not able to achieve the scalability of this component.

Another issue is that the application used was not developed while taking into consideration that it would be deployed in a Kubernetes cluster. This slowed us down because there were configurations that needed to be changed in order for the application to be able to work properly in the cluster.

Monitoring of the application was done using a Virtual Machine that had access to the same network where the cluster is hosted. The VM is running **Prometheus** and **Grafana** in docker containers. Since the Prometheus is not inside the cluster, we had to port-forward in order to access the services/pods that were running the exporters.

10 Conclusion

In conclusion, we learned about the process of deploying applications in orchestrated environments, taking into account the process of preparation/setup before the deployment, the deployment itself in the designated environment and the monitoring process after the deployment.

The preparation/setup phase allowed us to gain experience in working with other people's work and adapt ourselves/the program to reach the intended goal.

The deployment process allowed us to learn to allot about the Docker and Kubernetes environment and test our knowledge about networking and Linux systems.

The monitoring of the application can give the developers some insight into what is going well, and also about the errors. It does this while providing some information that allows making predictions about what is going to happen next and take some necessary actions for reacting and recovering from any system failure or security issue.