universidade de aveiro
theoria poiesis praxis

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Gestão de Infraestruturas de Computação (2021/22)

Assignment # 2: Initial Product Deployment

Bruno Bastos, Nmec: 93302
Mário Silva, Nmec: 93430
Miguel Fazenda, Nmec: 110877

Index

Misago

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 Q&A 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.

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.

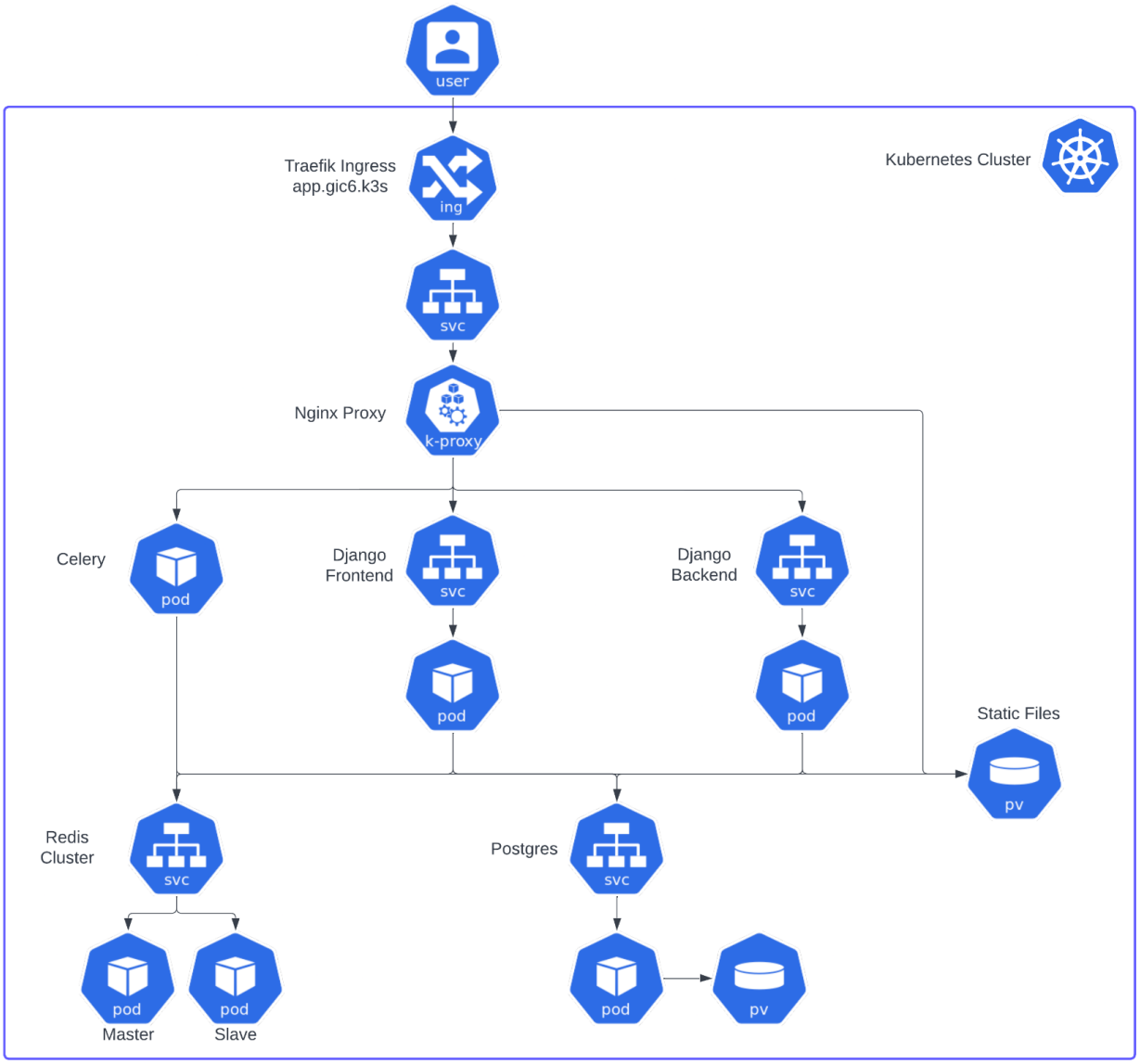
- Celery - the background tasks are performed by celery workers. Only a few sets of admin tasks depend on the celery worker.

Deployment Strategy

The overall deployment strategy follows multiple steps to ensure the application runs as intended.

Each step concerns with the deployment of technology from the stack, following the next order:

- 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;



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.

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

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 environment variables used to connect to the database and to create the initial admin account are being provided using a secret.

The figure below shows the file containing the secrets used by Django.

```
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=
```

Django can then load the secrets as environment variables. The next figure shows some secrets being loaded.

```
    env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgres-secrets
              key: POSTGRES_USER
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secrets
              key: POSTGRES_PASSWORD
```

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. 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.

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 below shows the routing configuration for the Nginx to perform this division of the received requests.

```
    location /api/ {
       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 / {
        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/;
      }
```

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.

Redis and 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.

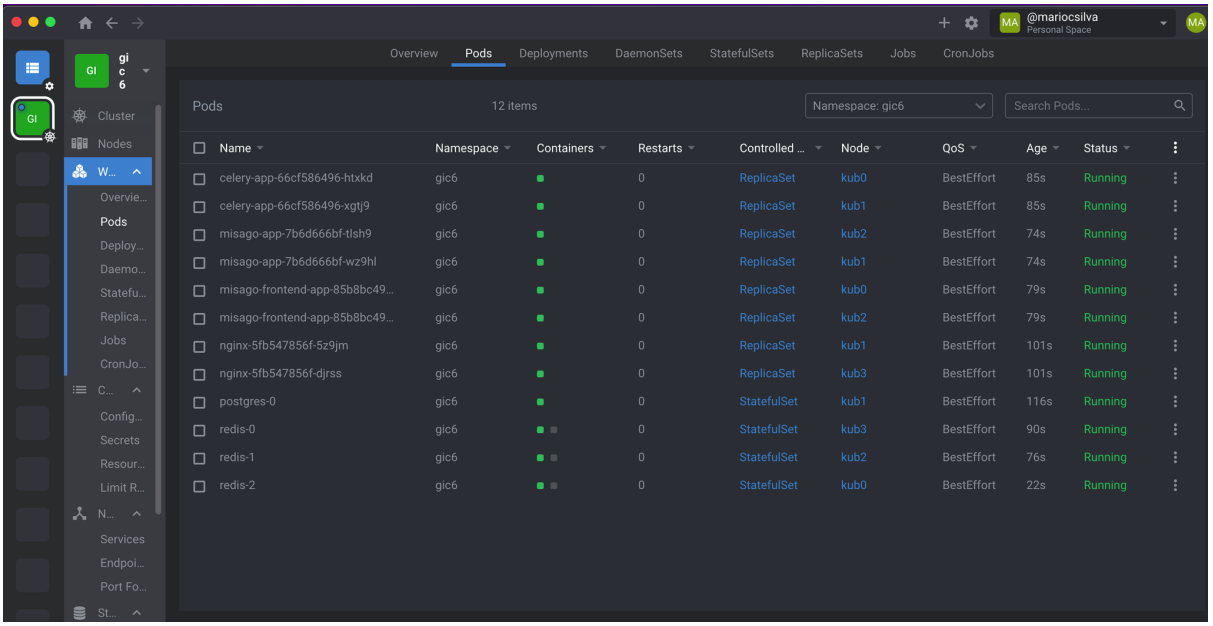
Scalability Bottlenecks

Although Redis is deployed with a cluster, it only has one master, so 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.

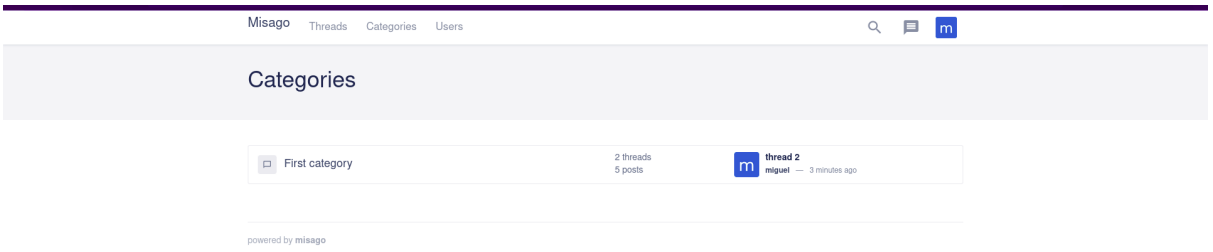
As it was previously mentioned, we were not able to replicate Postgres, so there is only one pod running. If we managed to do that, we would have one master pod and one or two slaves. Just like Redis, that would be an issue if there were many writes to the database, but considering that the application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

Demo

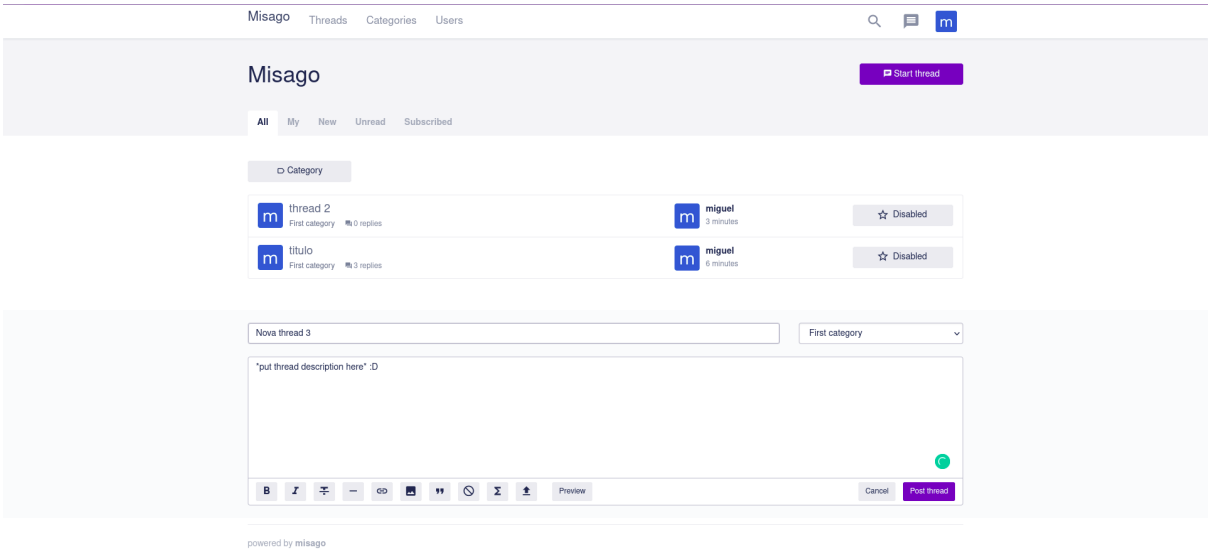
With the application deployed, we needed to test it to see if everything was working perfectly. Each component of the application was scaled to have 2 replicas, except for the postgres database. We will now show some of the basic functionalities.



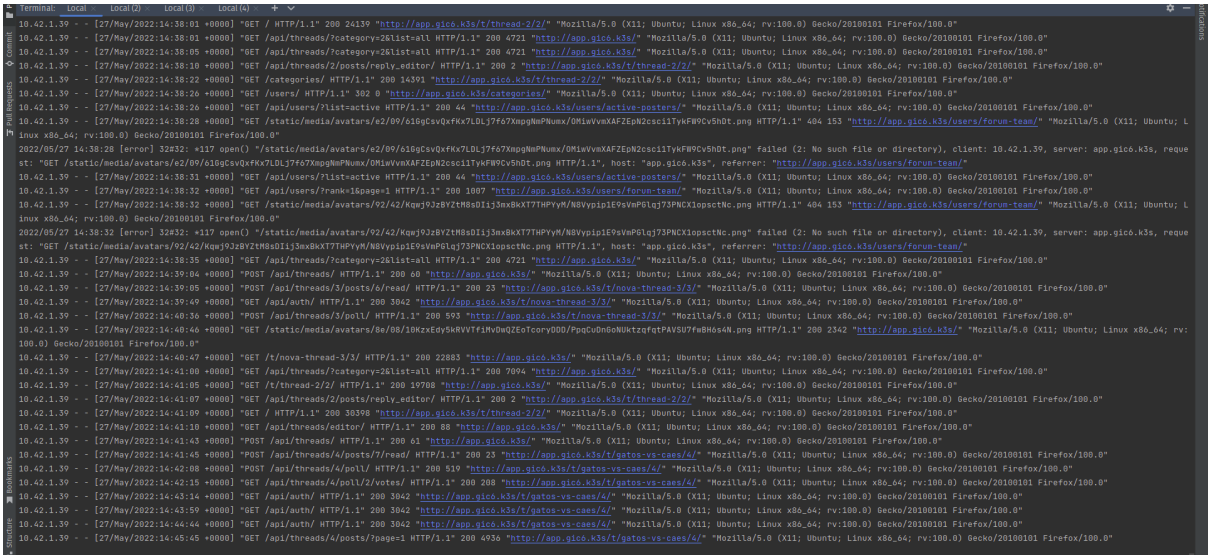
The user can register and login without any problem and then can proceed to one of the available pages. On the categories page, the user can see every category and the related information. The next figure shows an example of what the page can look like.



The next step can be to create a thread inside that category, which is easily achievable. In these interactions, the user can connect to different frontend/backend pods without noticing. The next figure illustrates an example of how to create a thread inside a category.



In the background, every request goes through the Nginx, which then forwards the requests to the services to handle which pod will process the requests. The next figure has the logs of the Nginx for some test requests that were made.



As it is possible to see, every request is correctly handled by the frontend and backend pods.

Since Redis requires data to be synchronized between replicas, we need to check if that is really what is happening. So we access both the master and the slaves and checked if the keys were the same. The next figure shows the logs from the master node.

```
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> auth a-very-complex-password-here
OK
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=10.42.2.67,port=6379,state=online,offset=62062821,lag=0
slave1:ip=10.42.0.194,port=6379,state=online,offset=62062821,lag=0
master_failover_state:no-failover
master_replid:e693b3255f5ec6fdfb4186fff14ce9b81cd341da
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:62063617
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:61015042
repl_backlog_histlen:1048576
127.0.0.1:6379> keys *
1) "emp3"
2) "x"
3) "_kombu.binding.celeryev"
4) "_kombu.binding.celery"
5) "_kombu.binding.celery.pidbox"
6) "emp2"
7) "emp1"
127.0.0.1:6379>
```

By comparing it with the logs from the slave, it is possible to see that every key is on both redis nodes.

```
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> auth a-very-complex-password-here
OK
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:redis-0.redis.gic6.svc.cluster.local
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_repl_offset:62112047
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:e693b3255f5ec6fdfb4186fff14ce9b81cd341da
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:62112047
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:61063472
repl_backlog_histlen:1048576
127.0.0.1:6379> keys *
1) "emp1"
2) "emp3"
3) "_kombu.binding.celery.pidbox"
4) "emp2"
5) "_kombu.binding.celery"
6) "_kombu.binding.celeryev"
7) "x"
127.0.0.1:6379>
```

Other deployments have a service that distributes the workload between all the replicas.