universidade de aveiro theoria poiesis praxis

Gestão de Infraestruturas de Computação 2021-22

Project Assignment 2

Theme: OpenPress Deployment

Authors: Leandro Silva nºmec 93446, Margarida Martins nºmec 93169 Daniel Gomes nºmec 93015

Date: 26/05/2022

Index

1 Introduction

This report aims to describe the work developed in the context of the *Gestão de Infraestruturas de Computação* Course 2021/22. For this assignment, the main goal was to "ship" an existent product to a *Kubernetes* Infrastructure. Further on this report, it will be described the product that we decided to deploy, its components as well as the deployment strategy chosen.

2 Our Product - OpenPress

Our Product, OpenPress, is a web platform which allows verified and independent journalists to publish media available for everyone. Thus, its goal is to allow users to watch media from independent sources, avoiding the spreading of misinformation and "fake news". The base software of our Product is MediaCMS, which is an open source web based platform that works out as a content management system. Its main features consist on publishing media content, watch and download the media, as well as creating playlists and share with others.

3 Individual Components

The technology stack that allows the functionalities that we provide is the following:

- NGINX, an HTTP and reverse proxy server, and a generic TCP/UDP proxy server. Regarding our application, not only it serves the static content necessary but also routes the HTTP requests to our Backend and Frontend, as we shall see nextly.
- React.js, a JavaScript library used in building user interfaces such as our Frontend.
- Django, a Python web framework. In our application it accepts and processes HTTP requests and is responsible to create celery tasks, manage cache and store data in the PostgreSQL database.
- Redis, a in-memory data store is used in our system for caching purposes and as a broker for Celery.
- Celery, a distributed task queue is responsible in our application for performing and scheduling tasks. The tasks are divided in two queues, long and short. Short tasks consists on simpler tasks such as clearing user sessions while long tasks deal mainly with media encoding, a more compute intensive task.
- PostgreSQL, an open source relational database system is our system database responsible for storing all the data.

4 Deployment Strategy

In the figure 9 below, it is presented an overall deployment architecture. In this section, we will explore the deployment approach for all the application's individual components.

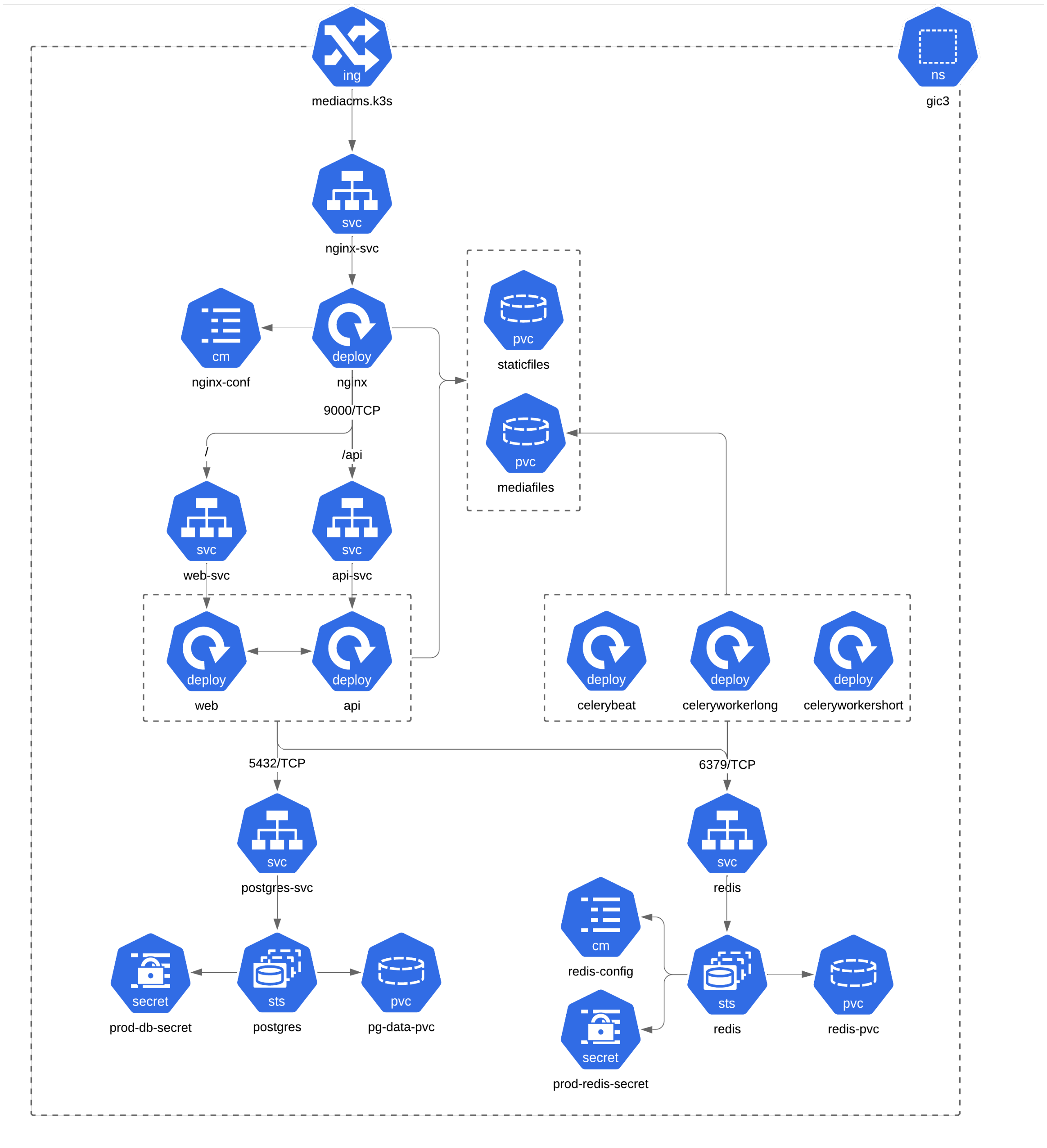


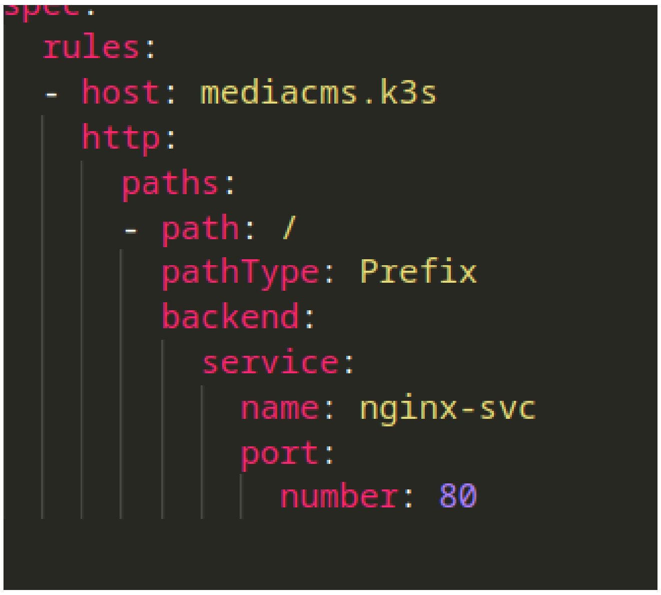
Figure 1: Deployment Architecture

4.1 NGINX

For NGINX, we decided to use Kubernetes Manifests of the following kinds:

- Persistent Volume and Persistent Volume Claims
- Deployment
- Service
- ConfigMap

As mentioned above, NGINX is responsible for serving static content and routing HTTP Requests. The Ingress that we created so that requests from outside the Cluster could reach our services, routes all requests to NGINX Service Created. The following figure contains the **Ingress** that we have done:



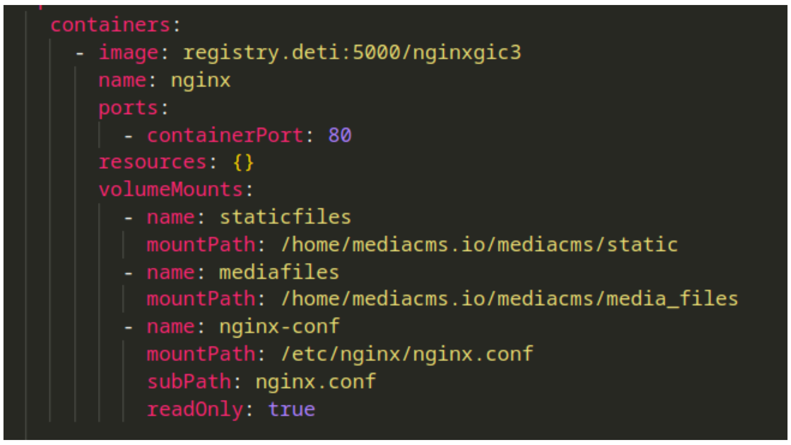
```yaml
spec:
  rules:
  - host: mediacms.k3s
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx-svc
            port:
              number: 80
```

Figure 2: Ingress Manifest

The only open port in the NGINX Service is the port 80, since is the port used for HTTP requests. Regarding the deployment of NGINX, three volumes have been created:

- A volume for the Media Files
- A volume for the Static Files
- A volume for the Nginx Configuration File

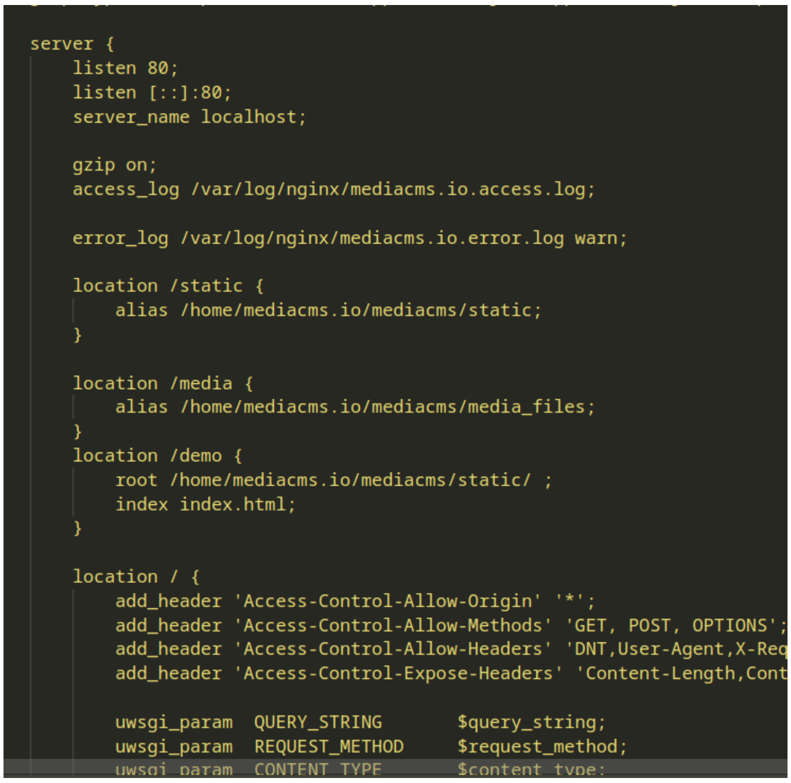
As the encoded videos were stored in the File System, this first volume mentioned is shared with the Celery Workers, the Django Service. The second Volume is shared with the Django Service so that the static content can be served by NGINX. Finally, the third one was necessary, as the Configuration File is Provided by a ConfigMap. The following figure shows an excerpt of the NGINX Deployment Manifest.



```yaml
containers:
  - image: registry.deti:5000/nginxgic3
    name: nginx
    ports:
      - containerPort: 80
    resources: {}
    volumeMounts:
      - name: staticfiles
        mountPath: /home/mediacms.io/mediacms/static
      - name: mediafiles
        mountPath: /home/mediacms.io/mediacms/media_files
      - name: nginx-conf
        mountPath: /etc/nginx/nginx.conf
        subPath: nginx.conf
        readOnly: true
```

Figure 3: Excerpt of the NGINX Deployment Manifest

Additionaly, the figure above contains the configuration provided in the Nginx ConfigMap.



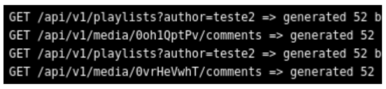
```nginx
server {
    listen 80;
    listen [::]:80;
    server_name localhost;

    gzip on;
    access_log /var/log/nginx/mediacms.io.access.log;

    error_log /var/log/nginx/mediacms.io.error.log warn;

    location /static {
        alias /home/mediacms.io/mediacms/static;
    }

    location /media {
        alias /home/mediacms.io/mediacms/media_files;
    }
    location /demo {
        root /home/mediacms.io/mediacms/static/ ;
        index index.html;
    }

    location / {
        add_header 'Access-Control-Allow-Origin' '*';
        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';
        add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-Req
        add_header 'Access-Control-Expose-Headers' 'Content-Length,Cont

        uwsgi_param  QUERY_STRING       $query_string;
        uwsgi_param  REQUEST_METHOD     $request_method;
        uwsgi_param  CONTENT_TYPE       $content_type;
```

Figure 4: Excerpt of the NGINX Deployment Manifest

As it may be seen, the locations */**static*** and */**media*** will retrieve files stored in the respective Volume Mount Paths. Besides this, the */**demo*** location is responsible for our promotional page, in which the necessary files are also stored in the volume for static content. Lastly, the Root Location will route requests to the Frontend Service. Another Location, is the */**api**/**v1***, which will be explained on the next subsection.

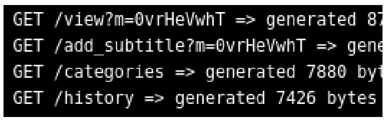
4.2 React and Django

In our System, Django uses React in its templating engine. Thus, in order to try to separate concerns in our Kubernetes Deployment, the API and Frontend Deployments use the same Docker Image, but these will be responsible for handling HTTP requests for the endpoints ***/api/v1*** and */*, respectively. As mentioned above, NGINX will do perform the routing of the requests to the correct service. The image 5 contains the logs of the API and Frontend, which confirms that each of these are receiving the correct requests.



```
GET /api/v1/playlists?author=teste2 => generated 52 b
GET /api/v1/media/0oh1QptPv/comments => generated 52
GET /api/v1/playlists?author=teste2 => generated 52 b
GET /api/v1/media/0vrHeVwhT/comments => generated 52
```

(a) Api deployment

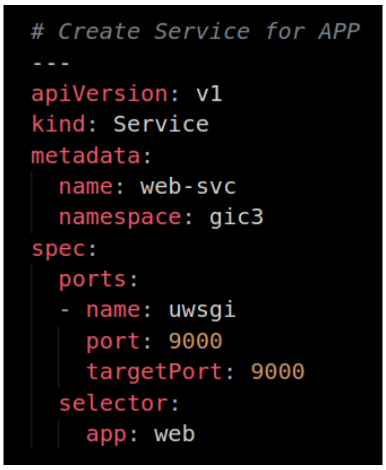


```
GET /view?m=0vrHeVwhT => generated 87
GET /add_subtitle?m=0vrHeVwhT => gene
GET /categories => generated 7880 byt
GET /history => generated 7426 bytes
```

(b) Frontend deployment

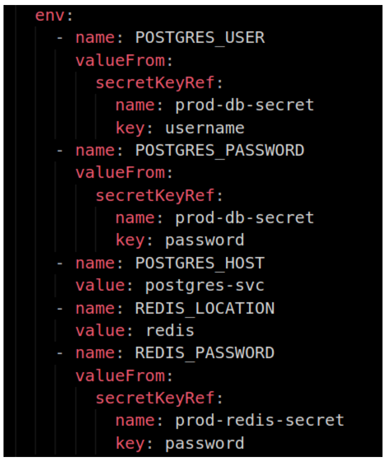
Figure 5: An excerpt of Api and Frontend deployment logs

Once again, both the deployments use the same docker image and have two volumes which are also shared with NGINX and Celery (Figure 6.c). In the deployments YAML files, some important configuration parameters such as the Redis and PostgreSQL passwords are passed as environment variables using generated Kubernetes secrets (Figure 6.b). This enables the Web and Api component to communicate with the cache and the database. Another manifest kind used for both Frontend and API were Services. These Services only exposed port 9000, since it was the only port used by the UWSGI Server in Django (Figure 6.a).



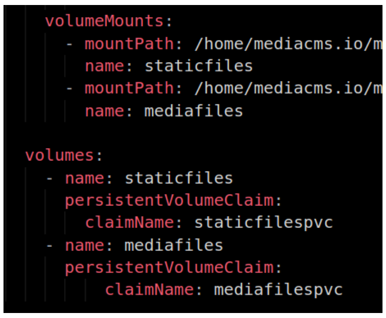
```yaml
# Create Service for APP
---
apiVersion: v1
kind: Service
metadata:
  name: web-svc
  namespace: gic3
spec:
  ports:
  - name: uwsgi
    port: 9000
    targetPort: 9000
  selector:
    app: web
```

(a) Service Configuration



```yaml
env:
  - name: POSTGRES_USER
    valueFrom:
      secretKeyRef:
        name: prod-db-secret
        key: username
  - name: POSTGRES_PASSWORD
    valueFrom:
      secretKeyRef:
        name: prod-db-secret
        key: password
  - name: POSTGRES_HOST
    value: postgres-svc
  - name: REDIS_LOCATION
    value: redis
  - name: REDIS_PASSWORD
    valueFrom:
      secretKeyRef:
        name: prod-redis-secret
        key: password
```

(b) Environment Variables



```yaml
  volumeMounts:
    - mountPath: /home/mediacms.io/m
      name: staticfiles
    - mountPath: /home/mediacms.io/m
      name: mediafiles

volumes:
  - name: staticfiles
    persistentVolumeClaim:
      claimName: staticfilespvc
  - name: mediafiles
    persistentVolumeClaim:
        claimName: mediafilespvc
```

(c) Volumes Configuration

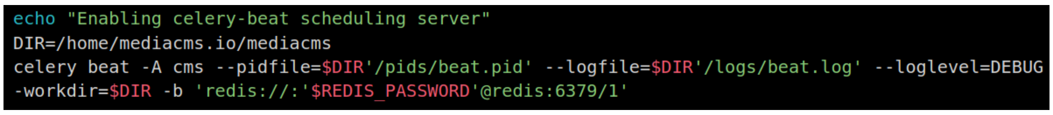
Figure 6: An excerpt of the Kubernetes entities for Api and Web components

4.3 Celery

For the Celery component, there are 3 deployments: one for the celery beat, and two for celery worker. One of the workers receives tasks from the short queue and the other from the long queue. Their tasks are many and diverse, but their core responsibility is to divide the uploaded videos in chunks and to encode them.

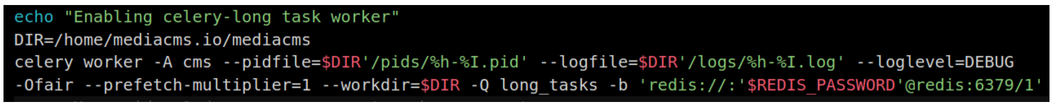
Their image is very similar from the Django and React component, meaning that they also need the Redis and

PostgreSQL environment variables to connect to them with the correct credentials. Besides, they also share the volume *mediafiles* to store the encoded video chunks. The only important exception is the command they finally execute in the *entrypoint.sh*, according to the job they are responsible for (Figure 7).



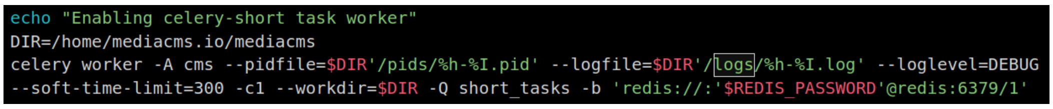
```
echo "Enabling celery-beat scheduling server"
DIR=/home/mediacms.io/mediacms
celery beat -A cms --pidfile=$DIR'/pids/beat.pid' --logfile=$DIR'/logs/beat.log' --loglevel=DEBUG
-workdir=$DIR -b 'redis://:'$REDIS_PASSWORD'@redis:6379/1'
```

(a) Celery Beat



```
echo "Enabling celery-long task worker"
DIR=/home/mediacms.io/mediacms
celery worker -A cms --pidfile=$DIR'/pids/%h-%I.pid' --logfile=$DIR'/logs/%h-%I.log' --loglevel=DEBUG
-Ofair --prefetch-multiplier=1 --workdir=$DIR -Q long_tasks -b 'redis://:'$REDIS_PASSWORD'@redis:6379/1'
```

(b) Celery Worker

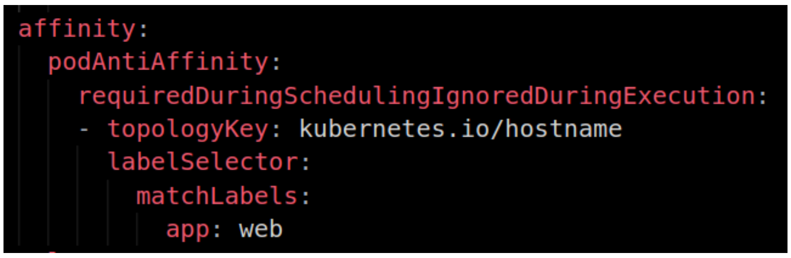


```
echo "Enabling celery-short task worker"
DIR=/home/mediacms.io/mediacms
celery worker -A cms --pidfile=$DIR'/pids/%h-%I.pid' --logfile=$DIR'/logs/%h-%I.log' --loglevel=DEBUG
--soft-time-limit=300 -c1 --workdir=$DIR -Q short_tasks -b 'redis://:'$REDIS_PASSWORD'@redis:6379/1'
```

(c) Celery Short Worker

Figure 7: Celery command entrypoint

Pod Affinity ensures two pods to be co-located in a single node To distribute the CPU load of our system, we defined a pod anti-affinity rule to separate the Web component from the Celery component so that we could ensure that the pods regarding these components were not in the same node. This configuration was added in the Celery deployment and can be seen in Figure 8.



```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - topologyKey: kubernetes.io/hostname
      labelSelector:
        matchLabels:
          app: web
```

Figure 8: Celery affinity with Web Component

4.4 Redis

To deploy Redis an Master-Slave Architecture has been used with two Slaves and one Master. The Master is responsible for performing the Write Operations and propagate those to the Slaves. Therefore, the Slaves only perform Read Operations. Regarding Redis we decided to use the following Kubernetes manifest kinds:

-

  Persistent Volume and Persistent Volume Claims
- **Service**
- **ConfigMap**

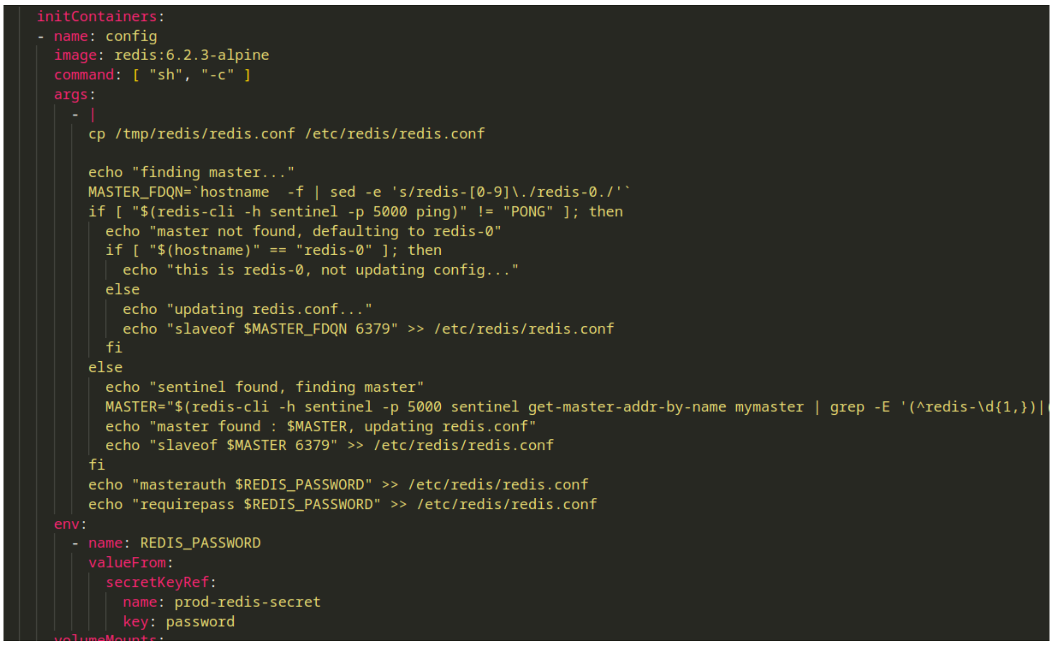
- **Secret**

Since Redis in our system requires data storing and works out as a Broker for Celery, it is a Stateful Service. Thus, the StatefulSet was the choosen workload since in these each pod is given a name and treated individually, in contrast to a Kubernetes Deployment, where pods are easily replaceable. Stateful applications require pods with unique identities.

To do so, initially an **initContainer** is used with some steps necessary to update the configuration file for slaves and master pods: The Master Pod will initially have the hostname *redis-0*, thus, if a Redis Pod has a hostname different from this one, the configuration file for Redis will be updated to identify the pod as a slave. Additionally, **two volumes** have been created, one responsible for the persistence of data and other to store the ConfigMap file, as it will be explained. Finally, the authentication for Redis will also be added to the configuration file used the Password defined in the **Secret** created for Redis. This secret is created using the following command:

kubectl create secret generic prod-redis-secret –from-literal=password=$(head -c 24 /dev/random — base64) -n gic3

The following figure shows the init container mentioned above:



```
initContainers:
- name: config
  image: redis:6.2.3-alpine
  command: [ "sh", "-c" ]
  args:
    - |
      cp /tmp/redis/redis.conf /etc/redis/redis.conf

      echo "finding master..."
      MASTER_FDQN=`hostname  -f | sed -e 's/redis-[0-9]\./redis-0./'`
      if [ "$(redis-cli -h sentinel -p 5000 ping)" != "PONG" ]; then
        echo "master not found, defaulting to redis-0"
        if [ "$(hostname)" == "redis-0" ]; then
          echo "this is redis-0, not updating config..."
        else
          echo "updating redis.conf..."
          echo "slaveof $MASTER_FDQN 6379" >> /etc/redis/redis.conf
        fi
      else
        echo "sentinel found, finding master"
        MASTER="$(redis-cli -h sentinel -p 5000 sentinel get-master-addr-by-name mymaster | grep -E '(^redis-\d{1,})|
        echo "master found : $MASTER, updating redis.conf"
        echo "slaveof $MASTER 6379" >> /etc/redis/redis.conf
      fi
      echo "masterauth $REDIS_PASSWORD" >> /etc/redis/redis.conf
      echo "requirepass $REDIS_PASSWORD" >> /etc/redis/redis.conf
  env:
    - name: REDIS_PASSWORD
      valueFrom:
        secretKeyRef:
          name: prod-redis-secret
          key: password
```

Figure 9: Init Container in Redis StatefulSet Manifest

Besides this, the Configuration file for Redis is provided using a ConfigMap. Finally, in order to other components of our system access Redis, we created a Service for it, allowing requests to **Port 6379**, the default one of Redis. This Service has to be headless so that the Network Identity of the Pods is provided, in other words, with no ClusterIP.

4.5 PostgreSQL

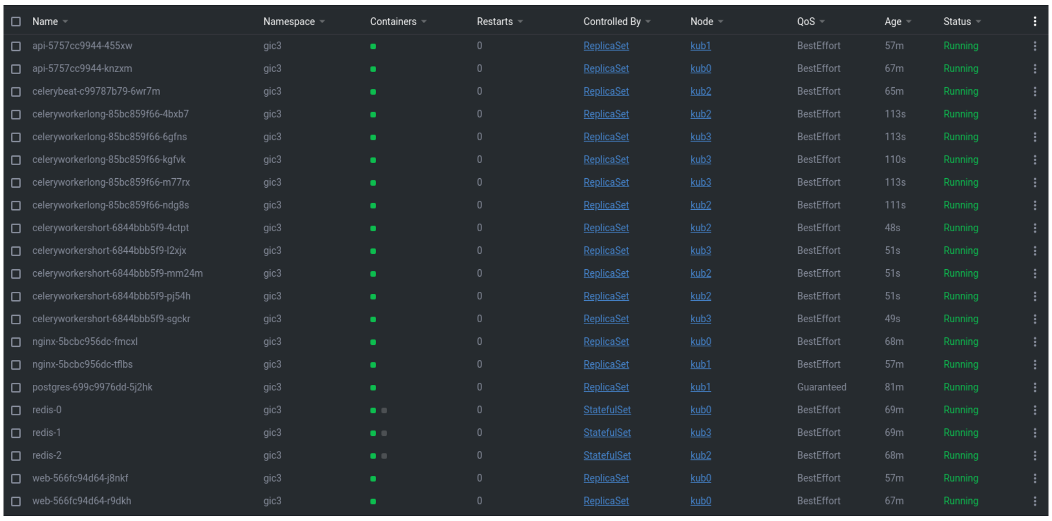
PostgreSQL's deployment consists of three different Kubernetes manifest kinds: a service which exposes port 5432, a persistent volume and a deployment. The deployment manifest kind uses a docker image created. It also sets some important environment variables, such as POSTGRES_USER and POSTGRES_PASSWORD. The variables values are obtained through Kubernetes secrets. The command used to generate the secret was the following: ***kubectl create secret generic prod-db-secret –from-literal=username=mediacms –from-literal=password=$(head -c 24 /dev/random — base64) -n gic3***

The deployment also creates a volume which will be used to store the data of the database.

5 Obtained Results

We successfully deployed our application which is available at: http://mediacms.k3s. A promotional page of our project is also available at: http://mediacms.k3s/demo/.

In figure 10 below we can see the final deployment status.

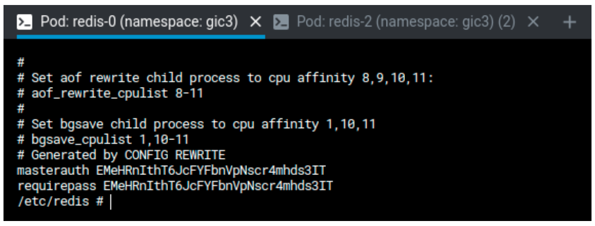


Name	Namespace	Containers	Restarts	Controlled By	Node	QoS	Age	Status
api-5757cc9944-455xw	gic3	🟩	0	ReplicaSet	kub1	BestEffort	57m	Running
api-5757cc9944-knzxm	gic3	🟩	0	ReplicaSet	kub0	BestEffort	67m	Running
celerybeat-c99787b79-6wr7m	gic3	🟩	0	ReplicaSet	kub2	BestEffort	65m	Running
celeryworkerlong-85bc859f66-4bxb7	gic3	🟩	0	ReplicaSet	kub2	BestEffort	113s	Running
celeryworkerlong-85bc859f66-6gfns	gic3	🟩	0	ReplicaSet	kub3	BestEffort	113s	Running
celeryworkerlong-85bc859f66-kgfvk	gic3	🟩	0	ReplicaSet	kub3	BestEffort	110s	Running
celeryworkerlong-85bc859f66-m77rx	gic3	🟩	0	ReplicaSet	kub3	BestEffort	113s	Running
celeryworkerlong-85bc859f66-ndg8s	gic3	🟩	0	ReplicaSet	kub2	BestEffort	111s	Running
celeryworkershort-6844bbb5f9-4ctpt	gic3	🟩	0	ReplicaSet	kub2	BestEffort	48s	Running
celeryworkershort-6844bbb5f9-l2xjx	gic3	🟩	0	ReplicaSet	kub3	BestEffort	51s	Running
celeryworkershort-6844bbb5f9-mm24m	gic3	🟩	0	ReplicaSet	kub2	BestEffort	51s	Running
celeryworkershort-6844bbb5f9-pj54h	gic3	🟩	0	ReplicaSet	kub2	BestEffort	51s	Running
celeryworkershort-6844bbb5f9-sgckr	gic3	🟩	0	ReplicaSet	kub3	BestEffort	49s	Running
nginx-5bcbc956dc-fmcxl	gic3	🟩	0	ReplicaSet	kub0	BestEffort	68m	Running
nginx-5bcbc956dc-tflbs	gic3	🟩	0	ReplicaSet	kub1	BestEffort	57m	Running
postgres-699c9976dd-5j2hk	gic3	🟩	0	ReplicaSet	kub1	Guaranteed	81m	Running
redis-0	gic3	🟩 ⬛	0	StatefulSet	kub0	BestEffort	69m	Running
redis-1	gic3	🟩 ⬛	0	StatefulSet	kub3	BestEffort	69m	Running
redis-2	gic3	🟩 ⬛	0	StatefulSet	kub2	BestEffort	68m	Running
web-566fc94d64-j8nkf	gic3	🟩	0	ReplicaSet	kub0	BestEffort	57m	Running
web-566fc94d64-r9dkh	gic3	🟩	0	ReplicaSet	kub0	BestEffort	67m	Running

Figure 10: Pods running in final deployment

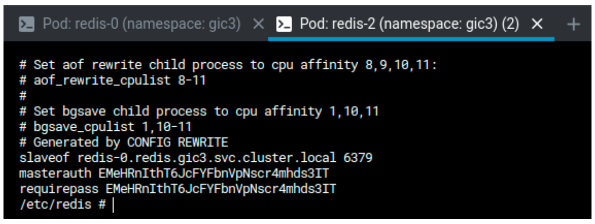
To respond to the bottlenecks of our application, we had to apply strategies to these points of failure. In our last presentation, we defined that horizontal scaling was of extreme importance in React, Django and Celery Worker components, since these were likely to receive more CPU load. Thus, we configured them such that their deployment could create replicas dynamically.

Despite not having in consideration the replication on the Redis component, we later notice that, for being the broker between Django and Celery, it would also suffer some of their load. For this reason, we built Redis as a cluster, increasing its performance and availability.


Pod: redis-0 (namespace: gic3)
Pod: redis-2 (namespace: gic3) (2)

```
#
# Set aof rewrite child process to cpu affinity 8,9,10,11:
# aof_rewrite_cpulist 8-11
#
# Set bgsave child process to cpu affinity 1,10,11
# bgsave_cpulist 1,10-11
# Generated by CONFIG REWRITE
masterauth EMeHRnIthT6JcFYFbnVpNscr4mhds3IT
requirepass EMeHRnIthT6JcFYFbnVpNscr4mhds3IT
/etc/redis # |
```

(a) Redis Pod-0 (Master)

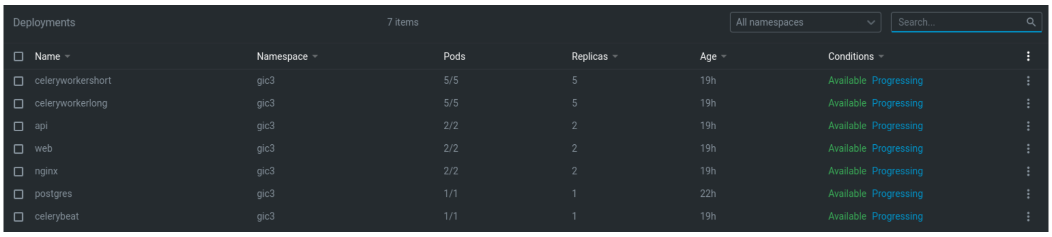

Pod: redis-0 (namespace: gic3)
Pod: redis-2 (namespace: gic3) (2)

```
# Set aof rewrite child process to cpu affinity 8,9,10,11:
# aof_rewrite_cpulist 8-11
#
# Set bgsave child process to cpu affinity 1,10,11
# bgsave_cpulist 1,10-11
# Generated by CONFIG REWRITE
slaveof redis-0.redis.gic3.svc.cluster.local 6379
masterauth EMeHRnIthT6JcFYFbnVpNscr4mhds3IT
requirepass EMeHRnIthT6JcFYFbnVpNscr4mhds3IT
/etc/redis # |
```

(b) Redis Pod-2 (Slave)

Figure 11: Redis cluster instances configurations

From the figure 11 above, it is shown how the configuration is adapted between master and slave redis nodes. Here, the Redis Pod-2 is configured to be a slave of Redis Pod-0 (named redis-0).



Name	Namespace	Pods	Replicas	Age	Conditions
celeryworkershort	gic3	5/5	5	19h	Available Progressing
celeryworkerlong	gic3	5/5	5	19h	Available Progressing
api	gic3	2/2	2	19h	Available Progressing
web	gic3	2/2	2	19h	Available Progressing
nginx	gic3	2/2	2	19h	Available Progressing
postgres	gic3	1/1	1	22h	Available Progressing
celerybeat	gic3	1/1	1	19h	Available Progressing

Figure 12: Pods and number of replicas in each deployment

As we can see in figure 12, there are 10 celery worker replicas, 5 for the long tasks and 5 for the short tasks. Two replicas were created for NGINX, Frontend(web) and API. The redis cluster contains 3 pods, one for the master redis-data-0 and the other two for slaves. Meanwhile postgreSQL and the celery beat only possess one replica.

It is important to notice that the celery worker replicas are in different nodes than the Frontend and web pods.

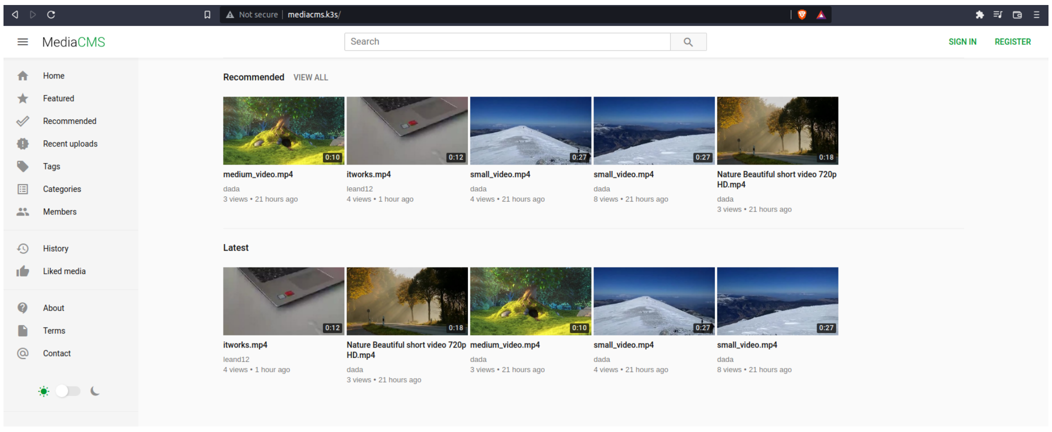
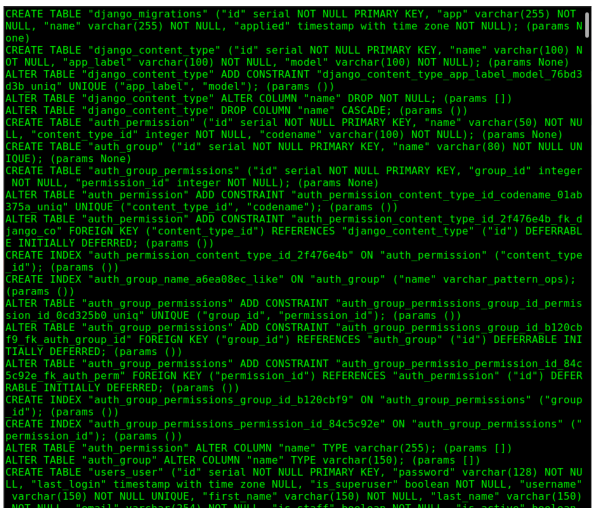
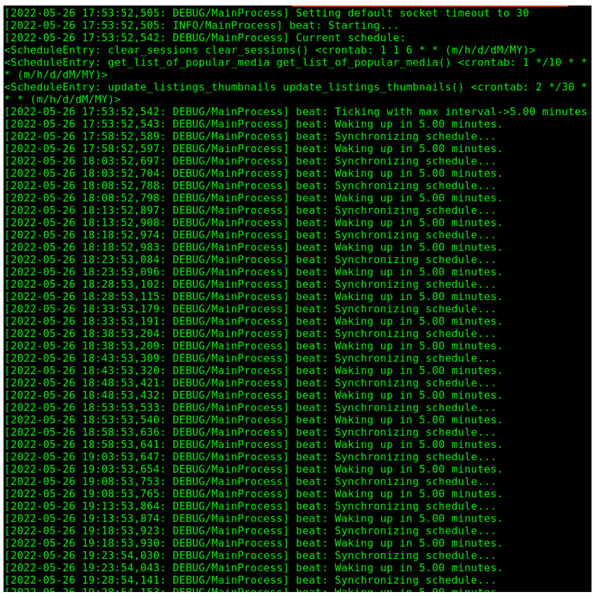


Figure 13: Website homepage in production



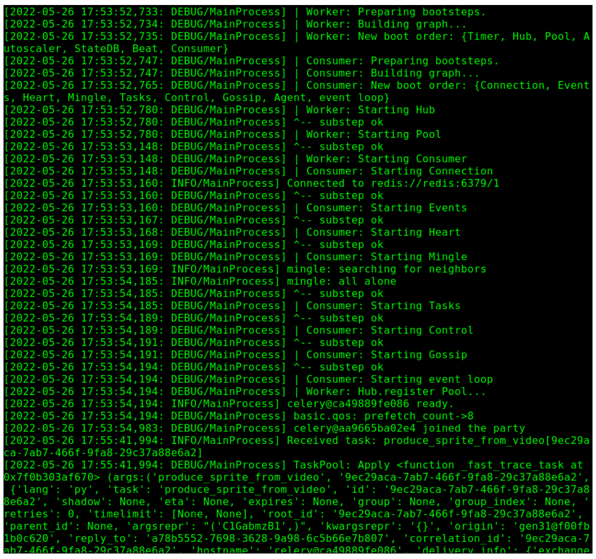
```
CREATE TABLE "django_migrations" ("id" serial NOT NULL PRIMARY KEY, "app" varchar(255) NOT
NULL, "name" varchar(255) NOT NULL, "applied" timestamp with time zone NOT NULL); (params N
one)
CREATE TABLE "django_content_type" ("id" serial NOT NULL PRIMARY KEY, "name" varchar(100) N
OT NULL, "app_label" varchar(100) NOT NULL, "model" varchar(100) NOT NULL); (params None)
ALTER TABLE "django_content_type" ADD CONSTRAINT "django_content_type_app_label_model_76bd3
d3b_uniq" UNIQUE ("app_label", "model"); (params ())
ALTER TABLE "django_content_type" ALTER COLUMN "name" DROP NOT NULL; (params [])
ALTER TABLE "django_content_type" DROP COLUMN "name" CASCADE; (params ())
CREATE TABLE "auth_permission" ("id" serial NOT NULL PRIMARY KEY, "name" varchar(50) NOT NU
LL, "content_type_id" integer NOT NULL, "codename" varchar(100) NOT NULL); (params None)
CREATE TABLE "auth_group" ("id" serial NOT NULL PRIMARY KEY, "name" varchar(80) NOT NULL UN
IQUE); (params None)
CREATE TABLE "auth_group_permissions" ("id" serial NOT NULL PRIMARY KEY, "group_id" integer
 NOT NULL, "permission_id" integer NOT NULL); (params None)
ALTER TABLE "auth_permission" ADD CONSTRAINT "auth_permission_content_type_id_codename_01ab
375a_uniq" UNIQUE ("content_type_id", "codename"); (params ())
ALTER TABLE "auth_permission" ADD CONSTRAINT "auth_permission_content_type_id_2f476e4b_fk_d
jango_co" FOREIGN KEY ("content_type_id") REFERENCES "django_content_type" ("id") DEFERRABL
E INITIALLY DEFERRED; (params ())
CREATE INDEX "auth_permission_content_type_id_2f476e4b" ON "auth_permission" ("content_type
_id"); (params ())
CREATE INDEX "auth_group_name_a6ea08ec_like" ON "auth_group" ("name" varchar_pattern_ops);
(params ())
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT "auth_group_permissions_group_id_permis
sion_id_0cd325b0_uniq" UNIQUE ("group_id", "permission_id"); (params ())
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT "auth_group_permissions_group_id_b120cb
f9_fk_auth_group_id" FOREIGN KEY ("group_id") REFERENCES "auth_group" ("id") DEFERRABLE INI
TIALLY DEFERRED; (params ())
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT "auth_group_permissio_permission_id_84c
5c92e_fk_auth_perm" FOREIGN KEY ("permission_id") REFERENCES "auth_permission" ("id") DEFER
RABLE INITIALLY DEFERRED; (params ())
CREATE INDEX "auth_group_permissions_group_id_b120cbf9" ON "auth_group_permissions" ("group
_id"); (params ())
CREATE INDEX "auth_group_permissions_permission_id_84c5c92e" ON "auth_group_permissions" ("
permission_id"); (params ())
ALTER TABLE "auth_permission" ALTER COLUMN "name" TYPE varchar(255); (params [])
ALTER TABLE "auth_group" ALTER COLUMN "name" TYPE varchar(150); (params [])
CREATE TABLE "users_user" ("id" serial NOT NULL PRIMARY KEY, "password" varchar(128) NOT NU
LL, "last_login" timestamp with time zone NULL, "is_superuser" boolean NOT NULL, "username"
 varchar(150) NOT NULL UNIQUE, "first_name" varchar(150) NOT NULL, "last_name" varchar(150)
```

(a) Web Logs



```
[2022-05-26 17:53:52,505: DEBUG/MainProcess] Setting default socket timeout to 30
[2022-05-26 17:53:52,505: INFO/MainProcess] beat: Starting...
[2022-05-26 17:53:52,542: DEBUG/MainProcess] Current schedule:
<ScheduleEntry: clear_sessions clear_sessions() <crontab: 1 1 6 * * (m/h/d/dM/MY)>
<ScheduleEntry: get_list_of_popular_media get_list_of_popular_media() <crontab: 1 */10 * *
* (m/h/d/dM/MY)>
<ScheduleEntry: update_listings_thumbnails update_listings_thumbnails() <crontab: 2 */30 *
* * (m/h/d/dM/MY)>
[2022-05-26 17:53:52,542: DEBUG/MainProcess] beat: Ticking with max interval->5.00 minutes
[2022-05-26 17:53:52,543: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 17:58:52,589: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 17:58:52,597: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:03:52,697: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:03:52,704: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:08:52,788: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:08:52,798: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:13:52,897: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:13:52,908: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:18:52,974: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:18:52,983: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:23:53,084: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:23:53,096: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:28:53,102: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:28:53,115: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:33:53,179: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:33:53,191: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:38:53,204: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:38:53,209: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:43:53,309: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:43:53,320: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:48:53,421: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:48:53,432: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:53:53,533: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:53:53,540: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 18:58:53,636: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 18:58:53,641: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:03:53,647: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 19:03:53,654: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:08:53,753: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 19:08:53,765: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:13:53,864: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 19:13:53,874: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:18:53,923: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 19:18:53,930: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:23:54,030: DEBUG/MainProcess] beat: Synchronizing schedule...
[2022-05-26 19:23:54,043: DEBUG/MainProcess] beat: Waking up in 5.00 minutes.
[2022-05-26 19:28:54,141: DEBUG/MainProcess] beat: Synchronizing schedule...
```

(b) Celery Beat Logs



```
[2022-05-26 17:53:52,733: DEBUG/MainProcess] | Worker: Preparing bootsteps.
[2022-05-26 17:53:52,734: DEBUG/MainProcess] | Worker: Building graph...
[2022-05-26 17:53:52,735: DEBUG/MainProcess] | Worker: New boot order: {Timer, Hub, Pool, A
utoscaler, StateDB, Beat, Consumer}
[2022-05-26 17:53:52,747: DEBUG/MainProcess] | Consumer: Preparing bootsteps.
[2022-05-26 17:53:52,747: DEBUG/MainProcess] | Consumer: Building graph...
[2022-05-26 17:53:52,765: DEBUG/MainProcess] | Consumer: New boot order: {Connection, Event
s, Heart, Mingle, Tasks, Control, Gossip, Agent, event loop}
[2022-05-26 17:53:52,780: DEBUG/MainProcess] | Worker: Starting Hub
[2022-05-26 17:53:52,780: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:52,780: DEBUG/MainProcess] | Worker: Starting Pool
[2022-05-26 17:53:53,148: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:53,148: DEBUG/MainProcess] | Worker: Starting Consumer
[2022-05-26 17:53:53,148: DEBUG/MainProcess] | Consumer: Starting Connection
[2022-05-26 17:53:53,160: INFO/MainProcess] Connected to redis://redis:6379/1
[2022-05-26 17:53:53,160: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:53,160: DEBUG/MainProcess] | Consumer: Starting Events
[2022-05-26 17:53:53,167: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:53,168: DEBUG/MainProcess] | Consumer: Starting Heart
[2022-05-26 17:53:53,169: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:53,169: DEBUG/MainProcess] | Consumer: Starting Mingle
[2022-05-26 17:53:53,169: INFO/MainProcess] mingle: searching for neighbors
[2022-05-26 17:53:54,185: INFO/MainProcess] mingle: all alone
[2022-05-26 17:53:54,185: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:54,185: DEBUG/MainProcess] | Consumer: Starting Tasks
[2022-05-26 17:53:54,189: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:54,189: DEBUG/MainProcess] | Consumer: Starting Control
[2022-05-26 17:53:54,191: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:54,191: DEBUG/MainProcess] | Consumer: Starting Gossip
[2022-05-26 17:53:54,194: DEBUG/MainProcess] ^-- substep ok
[2022-05-26 17:53:54,194: DEBUG/MainProcess] | Consumer: Starting event loop
[2022-05-26 17:53:54,194: DEBUG/MainProcess] | Worker: Hub.register Pool...
[2022-05-26 17:53:54,194: INFO/MainProcess] celery@ca49889fe086 ready.
[2022-05-26 17:53:54,194: DEBUG/MainProcess] basic.qos: prefetch_count->8
[2022-05-26 17:53:54,983: DEBUG/MainProcess] celery@aa9665ba02e4 joined the party
[2022-05-26 17:55:41,994: INFO/MainProcess] Received task: produce_sprite_from_video[9ec29a
ca-7ab7-466f-9fa8-29c37a88e6a2]
[2022-05-26 17:55:41,994: DEBUG/MainProcess] TaskPool: Apply <function _fast_trace_task at
0x7f0b303af670> (args:('produce_sprite_from_video', '9ec29aca-7ab7-466f-9fa8-29c37a88e6a2',
 {'lang': 'py', 'task': 'produce_sprite_from_video', 'id': '9ec29aca-7ab7-466f-9fa8-29c37a8
8e6a2', 'shadow': None, 'eta': None, 'expires': None, 'group': None, 'group_index': None, '
retries': 0, 'timelimit': [None, None], 'root_id': '9ec29aca-7ab7-466f-9fa8-29c37a88e6a2',
'parent_id': None, 'argsrepr': "('C1GabmzB1',)", 'kwargsrepr': '{}', 'origin': 'gen31@f00fb
1b0c620', 'reply_to': 'a78b5552-7698-3628-9a98-6c5b66e7b807', 'correlation_id': '9ec29aca-7
```

(c) Celery Worker Logs

Figure 14: Redis cluster instances configurations

In figure 14, we can see the logs from some components. In (a), the migrations on the database were applied; in (b) the celery beat was up, and its schedule was running normally; and in (c), the celery worker was up, and receiving tasks from the beat and the web component.

6 Scallability Bottlenecks

In this section it will be discussed some possible scallability bottlenecks and issues.

6.1 Celery Beat

In our deployment there can only be on instance of the celery beat. While in terms of workload, for our use case, this should not be problematic in case of the instance failure the celery beat will shutdown and would not be replace as there is no other replica.

This would have implications on the use of the application as the celery beat is responsible for scheduling celery tasks.

6.2 PostgreSQL

In our current deployment PostgreSQL scallability is not possible.

Just like Redis an master-slave cluster for PostgreSQL could have been implemented. This could have been done using Kubernetes StatefulSets. StatefulSets would allow for stable, persistent storage while guaranteeing higher availability.

6.3 Automation

In the process of deploying our application there are some manual tasks that must be done which might make it harder to automate the deployment process.

The tasks consist on copying the *static* folder after the nginx, Frontend and API deployments and creating the *hls* directory, inside the Media Folder: As Kubernetes Volume mount paths will delete any existent content in that path, the *hls* subdirectory would be deleted. Besides this, an existent issue is that sometimes NGINX has to be reloaded when deploying it for the first time, to renderize the Static Content.

7 Conclusion

Kubernetes was fundamental to ease the process of deploying a full stack application that consisted of multiple services running in different containers. Besides, it has shown the low effort that it requires to scale up components that present risks of being bottlenecks in our infrastructure, achieving high availability and performance in our application. It was definitely an interesting assignment since we have faced multiple challenges to reach our goals, where we managed to understand better in practice the mechanisms behind Kubernetes and how we could ”ship” a product to an Infrastructure in a scalable and manageable way.