Departamento de Eletrónica Telecomunicações e Informática

8240 - Mestrado Integrado em Engenharia de Computadores e Telemática

Engenharia e Gestão de Serviços

Year 2021/2022

EGS Project

Photography Course Application

Authors:
Lúcia Sousa 93086
Raquel Pinto 92948
Rafael Dias 95284
Sérgio Gasalho 84760
Repository:
https://github.com/psagasalho/EGS-Projeto

June, 2022

Contents

1 Introduction

In this project, as an initial phase, it was proposed the idea of a photography course application. Developing this concept further, the application would allow users to choose a subscription to access photography-related videos.

Based on this application the following micro services were raised. The authentication service, to ensure user authentication. The payment service, so that the user can pay the desired subscription and have access to the content of the application. And finally, the video file manager service that will make the videos available in the application. The implementation of each micro service will be covered in the following chapters.

The link for the repository is: https://github.com/psagasalho/EGS-Projeto.

2 Architecture

As said before, in this project there are three services (the authentication service, the payment service and the video file manager) and an aggregator.

It can be seen in Figure 1:

- The authentication service has two deployments, one deployment for the database (Deployment Auth DB) and another deployment for authentication (auth deployment) that communicates with the authapi-service through port 6000. This service communicates with the auth-db service (database) through port 3306 and with the aggregator service (django-service) which will be explained next. Both services have a pod.
- The payment service, it also has two deployments, one deployment for the database (Deployment Payment DB) and one deployment for the payment service named payment that communicates with the payment-service through port 6001. This service communicates with the database service (payment-db) through port 3306, with the authentication service (authapi-service) in order to validate the token and with the aggregator service (django-service) which will be explained next. Both payment services have a pod.
- The file management service as in the previous services has two deployments, one deployment for the database (Deployment File Manager DB) and one deployment for the file management with the name Deployment File Manager that communicates with the filemanagerapi-service through port 6000. In turn this service communicates with the database service named filemanager-db through port 3306, with the authentication service (authapi-service) to validate the token and with the aggregator service (django-service). Both payment services have a pod.
- Finally, the aggregator also has two deployments, the deployment postgres for the database and another deployment to aggregate all the services with the name deployment aggregator. The aggregator deployment communicates with a service (service django-service) through port 8000. This service in turn communicates with the database service (postgres-service). Both payment services have a pod.

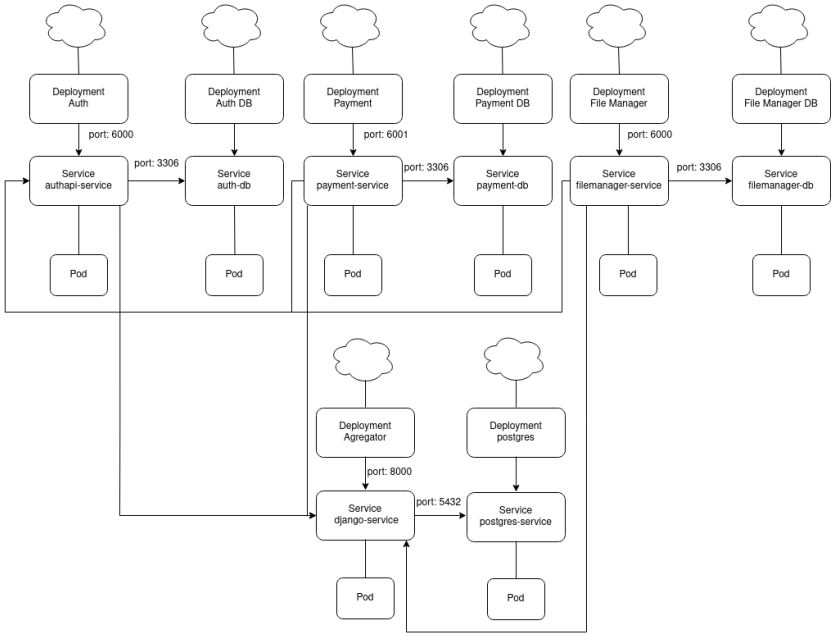

Figure 1: Architecture of Kubernetes.

3 Authentication Service

The authentication service allows, as the name indicates, the user to authenticate himself in the application. This micro service was implemented using Flask (python language) and has a MySQL database.

3.1 Authentication API

The API is necessary for the micro service architecture to function, the API is the communication tool between its services. To achieve this, five endpoints were created for this API.

In Figure 2 represents the endpoint of creating an account where the user enters his data (username, nMec, email, password, confirmPassword). When creating the account, these data will be saved in the database and is returning a json {"message" : "Success"} if there is no user with the same data and if the password is the same as the confirmPassword or return a json {"message" : "Bad Request"} if not.

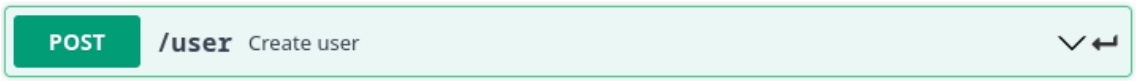

Figure 2: Endpoint to create user.

The next endpoint is the login endpoint (Figure 3), where a user logs in by entering their username and password. Before the user logs in it is verified if the password is correct. Returning a json {"token" : "tokenUser", "message" : "Success"} if password is correct or a json {"message" : "Bad Request"} if password is not correct.

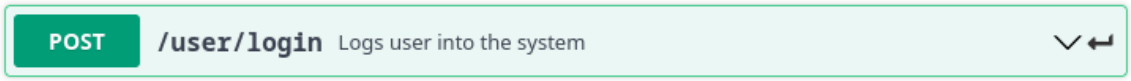

Figure 3: Endpoint to login an user.

The endpoint of the Figure 4 allows to know if a certain user (using his token) is authenticated, returning the json {"message" : "Success", "is_logged" : str(user.is_logged)} where is_logged is a boolean which informs whether it is authenticated or not.

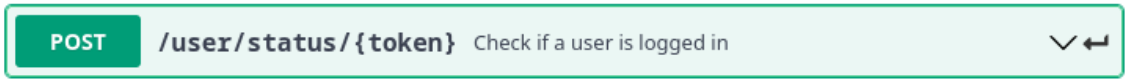

Figure 4: Endpoint to verify if user is authenticated.

As for the endpoint in Figure 5, it receives a token and logs out that user. If this operation is successful it returns a json {'message':'Success'}.

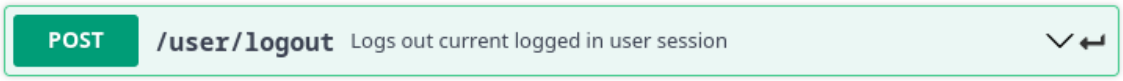

Figure 5: Endpoint to logout users.

The last endpoint (Figure 6) allows the other services to confirm the veracity of the authenticated user. It only checks if a token is valid or not, returning a json token {"response" : "invalid"} if invalid and a json token {"response" : "valid", "username" : usernameUser}.

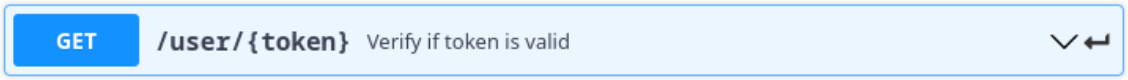

Figure 6: Endpoint to verify if token is valid.

To store the authentication service data we used a MySQL database that contains the username, nMec, email, a hash of a string composed by the password and by a predefined string, if the user is authenticated or not and finally the user token that is randomly generated for each user at the moment of the registration.

3.2 Create a Dockerfile for Python

In the Dockerfile for the authentication service, Figure 7 the first line has been added that tells Docker which base image will be used for the application. It will use the official Python image (python:3) which already has all the tools and packages needed to run a Python application.

```
1  FROM python:3
2
3  WORKDIR /usr/src/app
4
5  COPY requirements.txt ./
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY /app /usr/src/app/
9
10 EXPOSE 6000
11
12 CMD ["python3", "./app.py"]
```

Figure 7: Authentication service Dockerfile.

A work directory is created and this instructs Docker to use this path as the default location for all subsequent commands. Then, the very first thing to do once is downloaded a project written in Python is to install pip packages. This ensures that the application has all its dependencies installed.

Before run pip install, it is necessary to get the requirements.txt file into the image. For that is used the COPY command, this command takes two parameters. The first parameter tells Docker what file(s) to copy into the image. The second parameter tells Docker where that file(s) should be copied to.

Once the requirements.txt file is inside the image, the RUN command executes the command pip3 install and the modules are installed into the image.

Then it is executed the COPY command that takes all the files located in the directory /app and copies them into the image.

Expose instruction informs Docker that the container listens on the specified network ports (in our case 6000) at runtime.

Now, the next step is to tell Docker what command to run when the image is executed inside a container by using the CMD command.

3.3 Kubernetes

An Application can be run by creating a Kubernetes Deployment object, and it is possible to describe a Deployment in a YAML file.

The API authentication deployment, Figure 8, contains, for the Deployment object:

- the apiVersion, which indicates the version of the Kubernetes API used to create this object, is apps/v1;
- the kind, which indicates what kind of object is going to be created, is a Deployment object;
- metadata, the data that helps uniquely identify the object, including a name string, UID, and optional namespace, in this case, the name is authapi-deployment, the namespace is egs10 and app label is authapi;
- spec, which indicates the state for the object.

For the Service object, the apiVersion is v1, the kind that is a Service object, metadata, in this case, the name is authapi-service and the namespace is egs10 and spec that is detailed in Figure 8.

The deployment file, Figure 9, contains, the deployment object the apiVersion that is apps/v1, the kind is a Deployment object, the metadata the name is auth-db, the namespace is egs10 and app label is db and spec is detailed in Figure 9

In the spec.containers section, it is specified the MySQL image and assigned the value of the MYSQL ROOT PASSWORD environment variable to the password specified. Then it is connected the PVC to the deployment. And finally, in the separate section of the file, it is defined the service name and port. The storage configuration file, Figure 10, consists of two parts:

- First part defines the Persistent Volume. Customize the amount of allocated storage in spec.capacity.storage. In spec.hostPath specify the volume mount point;
- Second part of the file defines the PVC, Persistent Volume Claim.

In Kubernetes a secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. The YAML file, Figure 11, defines a Secret with one key in the data field, a password for the database.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: authapi-deployment
  namespace: egs10
  labels:
    app: authapi
spec:
  replicas: 1
  selector:
    matchLabels:
      app: authapi
  template:
    metadata:
      labels:
        app: authapi
    spec:
      containers:
        - name: authapi
          image: registry.deti:5000/egs10/api-auth:2022062401
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6000
          env:
            - name: db_root_password
              valueFrom:
                secretKeyRef:
                  name: flaskapi-secrets
                  key: db_root_password
            - name: db_name
              value: flaskapi

---
apiVersion: v1
kind: Service
metadata:
  name: authapi-service
  namespace: egs10
spec:
  ports:
  - port: 6000
    protocol: TCP
    targetPort: 5000
  selector:
    app: authapi
```

Figure 8: Authentication API deployment.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-db
  namespace: egs10
  labels:
    app: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: auth-db
        image: mysql
        imagePullPolicy: IfNotPresent
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: flaskapi-secrets
              key: db_root_password
        ports:
        - containerPort: 3306
          name: db-container
        volumeMounts:
          - name: mysql-persistent-storage
            mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv3-claim


---
apiVersion: v1
kind: Service
metadata:
  name: auth-db
  namespace: egs10
  labels:
    app: db
spec:
  ports:
  - port: 3306
    protocol: TCP
    name: auth-db
  selector:
    app: db
```

Figure 9: Authentication Database deployment.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv3-volume
  namespace: egs10
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv3-claim
  namespace: egs10
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Figure 10: Authentication Persistent Volume.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: flaskapi-secrets
  namespace: egs10
type: Opaque
data:
  db_root_password: ZXhhbXBsZQ==
```

Figure 11: Authentication API Secrets.

4 Payment Service

The payment service was designed and developed so that the user pays for the desired subscription. This microservice was implemented using Flask and MySQL.

4.1 Payment API

The API is necessary for the microservice architecture to function, the API is the communication tool between its services. To achieve this, three fundamental endpoints were created for this API.

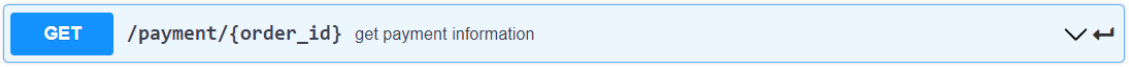

Figure 12: Endpoint to get payment information.

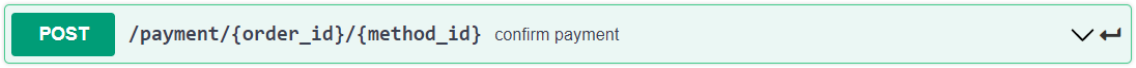

Figure 13: Endpoint to post information and confirm payment.

Figure 14: Endpoint to complete transaction.

The first endpoint, Figure 12, is a get method that will return all the information regarding the payment made by the user. For this to happen a database was created that contains the chosen subscription, the payment method, the user name, the price and the status of the purchase, whether it was paid or not.

The second endpoint, Figure 13, is a post method, here the user will fill in the subscription or purchase he wants to pay for, choose the payment method, confirm that he wants to pay, and be forwarded to a page to complete the payment. Here a transaction is created in the database.

In this endpoint the price will be introduced in the body so as the username. Then a json is sent {"message" : "Confirm Payment"}.

The third endpoint, Figure 14, is a post method, here the user will have access to a summary of his purchase in order to then be able to conclude the payment. The order status field in the database is updated to paid.

In this endpoint the order ID will be introduced in the body so as the username. Then a request is made to the authentication service, with this command requests.get('http://authapi-service:6000/user/'+token,headers=headers), to check if the token is valid. If the token is valid it returns a json, {"message": "Transaction Successful"}, otherwise returns {"message": "Transaction not Successful"} .

4.2 Create a Dockerfile for Python

A Docker image is a binary file. It is made up of multiple layers and is used to run code in a Docker container. Images are built from instructions in Dockerfiles to create a containerized version of the application.

A Dockerfile is a collection of instructions for building a Docker image that can then be run as a container. As each instruction is run in a Dockerfile, a new Docker layer is created. These

layers, which are known as intermediate images, are created when a change is made to your Docker image. Every Dockerfile begins with a parent or base image over which various commands are run.

```
1  FROM python:3
2
3  WORKDIR /usr/src/app
4
5  COPY requirements.txt ./
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY /app /usr/src/app/
9
10 CMD [ "python", "./app.py" ]
```

Figure 15: Dockerfile.

A Dockerfile is a text document that contains the instructions to assemble a Docker image. When we tell Docker to build our image by executing the docker build command, Docker reads these instructions, executes them, and creates a Docker image as a result.

In the Dockerfile for the payment service, Figure 15, it was added the first line that tells Docker which base image will be used for the application. It will use the official Python image that already has all the tools and packages needed to run a Python application.

A work directory is created and this instructs Docker to use this path as the default location for all subsequent commands. Then, the very first thing to do once is downloaded a project written in Python is to install pip packages. This ensures that the application has all its dependencies installed.

Before run pip install, it is necessary to get the requirements.txt file into the image. For that is used the COPY command, this command takes two parameters. The first parameter tells Docker what file(s) to copy into the image. The second parameter tells Docker where that file(s) should be copied to.

Once the requirements.txt file is inside the image, the RUN command executes the command pip3 install and the modules are installed into the image.

Then it is executed the COPY command that takes all the files located in the directory /app and copies them into the image. Now, the next step is to tell Docker what command to run when the image is executed inside a container by using the CMD command.

4.3 Build an image and run as a container

To build an image it is used the docker build command. The docker build command builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in this context. It is possible to run that image and see if the application is running correctly.

A container is a normal operating system process except that this process is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

To run an image inside of a container, it is used the docker run command. The docker run command requires one parameter which is the name of the image.

4.4 Docker Compose

The Compose file is a YAML file defining services, networks, and volumes for a Docker application.

In the compose file, Figure 16, three services backed by Docker images have been defined, db_payments, adminer_payment and api_payments. One persistent volume, attached to the back-end and a network were also defined.

```yaml
# Use root/example as user/password credentials
version: '3.1'

services:

  db_payments:
    image: mysql
    ports:
      - "3306:3306"
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - db_payment_data:/var/lib/mysql
    cap_add:
      - SYS_NICE  # CAP_SYS_NICE
    networks:
        - internal

  adminer_payment:
    image: adminer
    restart: always
    ports:
        - 5679:8080

  api_payments:
      container_name: payments_api
      build:
        context: .
        dockerfile: dockerfile
      environment:
           SECRET_KEY : 'thisissecret'
      ports:
        - 7002:5000
      networks:
        - internal
      volumes:
        - /home/lucia/Documents/EGS/Secrets_EGS/Payment/secret.txt:/var/run/Payment/secret.txt:ro

volumes:
  db_payment_data:

networks:
  internal:
    external:
      name: "internal"
```

Figure 16: Docker-Compose.

4.5 MySQL

Instead of downloading MySQL, installing, configuring, and then running the MySQL database as a service, it is possible to use the Official Docker Image for MySQL and run it in a container.

Before MySQL is run in a container, a pair of volumes are created that Docker can manage to store the persistent data and configuration. To create these volumes the command docker volume create mysql is executed.

4.6 Kubernetes

The API payment deployment, Figure 17, contains, for the Deployment object:

- the apiVersion, which indicates the version of the Kubernetes API used to create this object, is apps/v1;
- the kind, which indicates what kind of object is going to be created, is a Deployment object;
- metadata, the data that helps uniquely identify the object, including a name string, UID, and optional namespace, in this case, the name is payment, the namespace is egs10 and app label is payment;
- spec, which indicates the state for the object, that is detailed in Figure 17.

For the Service object:

- the apiVersion v1;
- the kind that is a Service object;
- metadata, in this case, the name is payment-service and the namespace is egs10;
- spec that is detailed in Figure 17.

The deployment file, Figure 18, contains, for the deployment object:

- the apiVersion is apps/v1;
- the kind is a Deployment object;
- metadata, the name is payment-db, the namespace is egs10 and app label is payment-db;
- spec is detailed in Figure 18.

For the Service object:

- the apiVersion v1;
- the kind that is a Service object;
- metadata, the name is payment-db and the namespace is egs10;
- spec that is detailed in Figure 18.

In the spec.containers section, it is specified the MySQL image and assigned the value of the MYSQL_ROOT_PASSWORD environment variable to the password specified.

Then it is connected the PVC to the deployment. And finally, in the separate section of the file, it is defined the service name and port.

The storage configuration file, Figure 19, consists of two parts:

- the first part defines the Persistent Volume. Customize the amount of allocated storage in spec.capacity.storage. In spec.hostPath specify the volume mount point;
- the second part of the file defines the PVC, Persistent Volume Claim.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment
  namespace: egs10
  labels:
    app: payment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: payment
  template:
    metadata:
      labels:
        app: payment
    spec:
      containers:
        - name: payment
          image: registry.deti:5000/egs10/api-payments:2022062401
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6001
          env:
            - name: db_root_password
              valueFrom:
                secretKeyRef:
                  name: flaskapi-secrets
                  key: db_root_password
            - name: db_name
              value: flaskapi

---
apiVersion: v1
kind: Service
metadata:
  name: payment-service
  namespace: egs10
spec:
  ports:
  - port: 6001
    protocol: TCP
    targetPort: 5000
  selector:
    app: payment
```

Figure 17: Payment API deployment.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-db
  namespace: egs10
  labels:
    app:  payment-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app:  payment-db
  template:
    metadata:
      labels:
        app: payment-db
    spec:
      containers:
      - name: payment-db
        image: mysql
        imagePullPolicy: IfNotPresent
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: flaskapi-secrets
              key: db_root_password
        ports:
        - containerPort: 3306
          name: dbpayment
        volumeMounts:
          - name: mysql-persistent-storage
            mountPath: /var/lib/othersql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv5-claim


---
apiVersion: v1
kind: Service
metadata:
  name: payment-db
  namespace: egs10
  labels:
    app: payment-db
spec:
  ports:
  - port: 3306
    protocol: TCP
    name: mysql
  selector:
    app: payment-db
```

Figure 18: Payment Database deployment.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv5-volume
  namespace: egs10
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv5-claim
  namespace: egs10
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Figure 19: Payment Persistent Volume.

5 File Manager Service

The file management service was implemented to allow the user to upload a video, download a video, or have access to see what videos have already been uploaded to the application. For this service it was used Spring Boot and MySQL.

5.1 File Manager API

For the file management service API three endpoints were created.

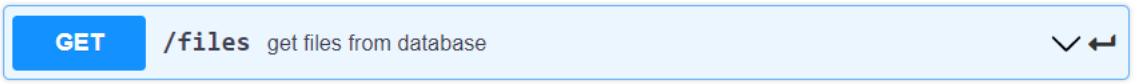

Figure 20: Endpoint to get files from database.

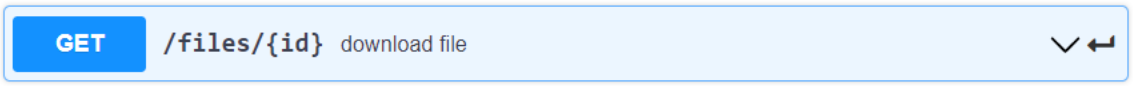

Figure 21: Endpoint to download a file.

The first endpoint, Figure 20, is a get method to return all the files that have been uploaded and are in the database.

The second endpoint, Figure 21, is a get method to return a specific file for download.

Figure 22: Endpoint to upload file.

The third endpoint, Figure 22, is a post method for loading a file into the database.

5.2 Create a Dockerfile for Java

In the Dockerfile for the file management service, Figure 23 ,it was added the first line that tells Docker which base image will be used for the application. It will use the official Openjdk 11 image.

The LABEL command is only indicating the person responsible for maintaining the image.

Then, with the ADD command, Docker copies the jar files into the image. The entrypoint will be the executable to start when the container is booting.

```
1  FROM openjdk:17
2  ADD target/fileManager-0.0.1-SNAPSHOT.jar fileManager.jar
3  ENTRYPOINT ["java","-jar","fileManager.jar"]
```

Figure 23: Dockerfile.

5.3 Kubernetes

An Application can be run by creating a Kubernetes Deployment object, and it is possible to describe a Deployment in a YAML file.

This YAML, Figure 24 creates a Service that is available to internal network requests. The port value is specified so that the service is allocated to that port in the cluster.

The API file management deployment, Figure 24, contains, for the Deployment object:

- the apiVersion, which indicates the version of the Kubernetes API used to create this object, is apps/v1;
- the kind, which indicates what kind of object is going to be created, is a Deployment object;
- metadata, the data that helps uniquely identify the object, including a name string, UID, and optional namespace, in this case, the name is fileManagerapi-deployment, the namespace is egs10 and app label is filemanagerapi;
- spec, which indicates the state for the object, that is detailed in Figure 24.

For the Service object:

- the apiVersion v1;
- the kind that is a Service object;
- metadata, in this case, the name is fileManagerapi-service and the namespace is egs10;
- spec that is detailed in Figure 24.

In Kubernetes a Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: filemanagerapi-deployment
  namespace: egs10
  labels:
    app: filemanagerapi
spec:
  replicas: 1
  selector:
    matchLabels:
      app: filemanagerapi
  template:
    metadata:
      labels:
        app: filemanagerapi
    spec:
      containers:
        - name: filemanagerapi
          image: registry.deti:5000/egs10/api-filemanager:2022062401
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6000
          env:
            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: filemanagerapi-secrets
                  key: DB_PASSWORD
            - name: DB_NAME
              value: testdb
---
apiVersion: v1
kind: Service
metadata:
  name: filemanagerapi-service
  namespace: egs10
spec:
  ports:
  - port: 6000
    protocol: TCP
    targetPort: 5000
  selector:
    app: filemanagerapi
```

Figure 24: File Management API deployment.

in a container image. Using a Secret means that it is not necessary to include confidential data in the application code. The YAML file, Figure 25, defines a Secret with one key in the data field, a password for the database.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: filemanagerapi-secrets
  namespace: egs10
type: Opaque
data:
  DB_PASSWORD: MTIzNDU2Nzg=
```

Figure 25: File Management API Secrets.

The storage configuration file, Figure 26, consists of two parts:

- the first part defines the Persistent Volume. Customize the amount of allocated storage in spec.capacity.storage. In spec.hostPath specify the volume mount point;
- the second part of the file defines the PVC, Persistent Volume Claim.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv3-volume
  namespace: egs10
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv3-volume
  namespace: egs10
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Figure 26: File Management Persistent Volume.

The deployment file, Figure 27, defines the resources the MySQL deployment will use.

In the spec.containers section, it is specified the MySQL image and assigned the value of the MYSQL_ROOT_PASSWORD environment variable to the password you specified in the Secret from Figure 25.

Then it is connected the PVC from Figure 26 to the deployment. And finally, in the separate section of the file, it is defined the service name and port.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: filemanager-db
  namespace: egs10
  labels:
    app: testdb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: testdb
  template:
    metadata:
      labels:
        app: testdb
    spec:
      containers:
      - name: filemanager-db
        image: mysql
        imagePullPolicy: IfNotPresent
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: filemanagerapi-secrets
              key: DB_PASSWORD
        ports:
        - containerPort: 3306
          name: testdb-contain
        volumeMounts:
          - name: mysql-persistent-storage
            mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv3-claim
---
apiVersion: v1
kind: Service
metadata:
  name: filemanager-db-service
  namespace: egs10
  labels:
    app: testdb
spec:
  ports:
  - port: 3306
    protocol: TCP
    name: filemanager-db-service
  selector:
    app: testdb
```

Figure 27: File Management database deployment.

6 Backend and Frontend

For this stage of the project, the backend and frontend were implemented using django and HTML, CSS and JavaScript, for the design and development of page components it was used bootstrap.

6.1 Kubernetes

The Application deployment, contains, for the Ingress object, Figure 28:

- the apiVersion is apps/v1;
- the kind is a Ingress object that manages external access to the services in a cluster;
- metadata, the data that helps uniquely identify the object, the name is aggregator, the namespace is egs10 and app label is aggregator;
- spec is detailed in Figure 28.

For the Deployment object, Figure 29:

- the apiVersion is apps/v1;
- the kind is a Deployment object;
- metadata, the data that helps uniquely identify the object, the name is aggregator, the namespace is egs10 and app label is aggregator;
- spec is detailed in Figure 29.

For the Service object, Figure 30:

- the apiVersion v1;
- the kind that is a Service object;
- metadata, the name is django-service and the namespace is egs10;
- spec is detailed in Figure 30.

The database deployment file, Figure **??**, contains, for the deployment object:

- the apiVersion is apps/v1;
- the kind is a Deployment object;
- metadata, the name is postgres, the namespace is egs10 and app label is postgres-container;
- spec is detailed in Figure **??**.

For the Service object:

- the apiVersion v1;
- the kind that is a Service object;
- metadata, the name is postgres-service and the namespace is egs10;
- spec that is detailed in Figure **??**.

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: django-ingress
  namespace: egs10
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/frontend-entry-points: http,https
    traefik.ingress.kubernetes.io/redirect-entry-point: https
    traefik.ingress.kubernetes.io/redirect-permanent: "true"
spec:
  rules:
  - host: agregator.k3s
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: django-service
            port:
              number: 8000

```

Figure 28: Application Deployment, Ingress.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: agregator
  namespace: egs10
  labels:
    app: agregator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: agregator
  template:
    metadata:
      labels:
        app: agregator
    spec:
      containers:
        - name: agregator
          image: registry.deti:5000/egs10/agregator:2022062402
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8000
          env:
            - name: POSTGRES_USER
              valueFrom:
                secretKeyRef:
                  name: postgres-credentials
                  key: user

            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: postgres-credentials
                  key: password

            - name: POSTGRES_HOST
              value: postgres-service
---
```

Figure 29: Application Deployment, Deployment.

The Kubernetes job functionality is to create one pod which do a specific job then dies without restarting. Using a Kubernetes Job inside of the cluster to run the migration, the application will already have access to the database and the credentials are already passed to the application,

```
64  ---
65
66  kind: Service
67  apiVersion: v1
68  metadata:
69    name: django-service
70    namespace: egs10
71  spec:
72    selector:
73      app: agregator
74    ports:
75    - protocol: TCP
76      port: 8000
77      targetPort: 8000
```

Figure 30: Application Deployment, Service.

using Secrets. The Job will run the new version of the application container image containing any new migrations that need to be run.

The storage configuration file, Figure 31, consists of two parts:

- the first part defines the Persistent Volume. Customize the amount of allocated storage in spec.capacity.storage. In spec.hostPath specify the volume mount point;
- the second part of the file defines the PVC, Persistent Volume Claim.

```
1   ---
2   apiVersion: v1
3   kind: PersistentVolume
4   metadata:
5     name: postgres-pv2-volume
6     namespace: egs10
7     labels:
8       type: local
9   spec:
10    storageClassName: manual
11    capacity:
12      storage: 1Gi
13    accessModes:
14      - ReadWriteOnce
15    persistentVolumeReclaimPolicy: Retain
16    hostPath:
17      path: "/mnt/data"
18  ---
19  apiVersion: v1
20  kind: PersistentVolumeClaim
21  metadata:
22    name: postgres-pv2-claim
23    namespace: egs10
24  spec:
25    storageClassName: manual
26    accessModes:
27      - ReadWriteOnce
28    resources:
29      requests:
30        storage: 1Gi
```

Figure 31: Persistent Volume Deployment.

The secrets deployment file, Figure 32, defines a Secret with one user in the data field and a password.

```
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: postgres-credentials
5    namespace: egs10
6  type: Opaque
7  data:
8    user: ZGphbmdv
9    password: MWEyNmQxZzI2ZDFnZXNiP2U3ZGVzYj9lN2Q=
```

Figure 32: Secrets Deployment.