Coursework Project Advanced Cloud Computing
“Piazza App”

Name: Francisco Macaya
Date: 01-12-2025
Github: https://github.com/Foco22/Piazza-backend-project

Disclaimer: The testing case videos are only available on GitHub because the ZIP file exceeded Moodle’s upload size limit when submitting the coursework.

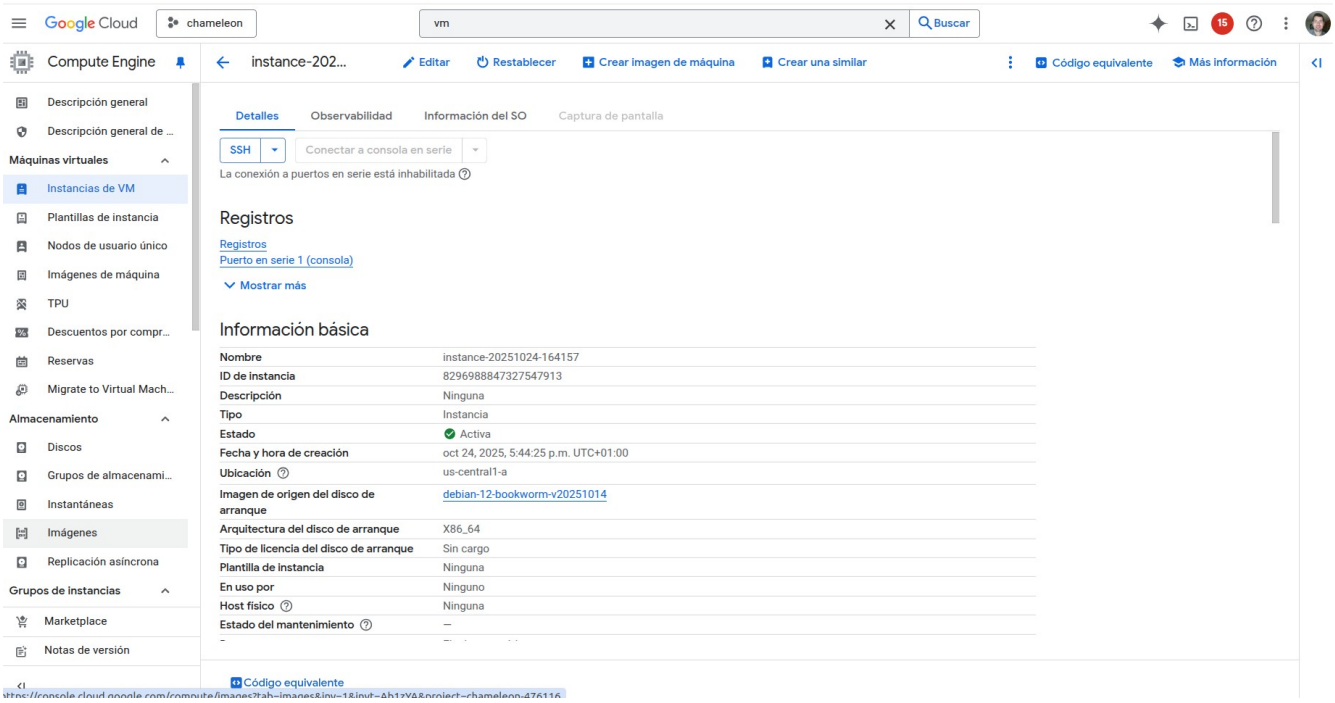
Introduction

This report will present the complete process of building the Piazza App, using Node.js for the backend along with a small JavaScript frontend to demonstrate some functionality of the application. The frontend will not be documented in detail but will be used to showcase certain backend features. The report follows the same structure as the Assignment Marking Criteria provided in the course, allowing for a clearer and more efficient review by the instructor.

1- Phase A: Install and deploy software in virtualised environments

a-) Install all the necessary packages in your virtual machine.

The first step was to create and install the necessary packages to run the backend on the Virtual Machine (VM). The image below shows the VM that was created in Google Cloud.



The VM was created using the default setup, so it was necessary to install several packages and libraries required for the application. The following libraries and packages were installed:

1. *Update the packages in the systems:* As good practice, it was neccesary to run the command sudo apt upgrade -y, so all the packages are installed with the newest librarias and versions.

2. *Docker:* It was necessary to install Docker on the virtual machine, as the GitHub Action connects to Docker Hub to pull the Docker image and deploy the application on the VM. The best way to install Docker was by connecting to the VM via SSH from Google Cloud and running the following commands:

   sudo apt install docker.io -y

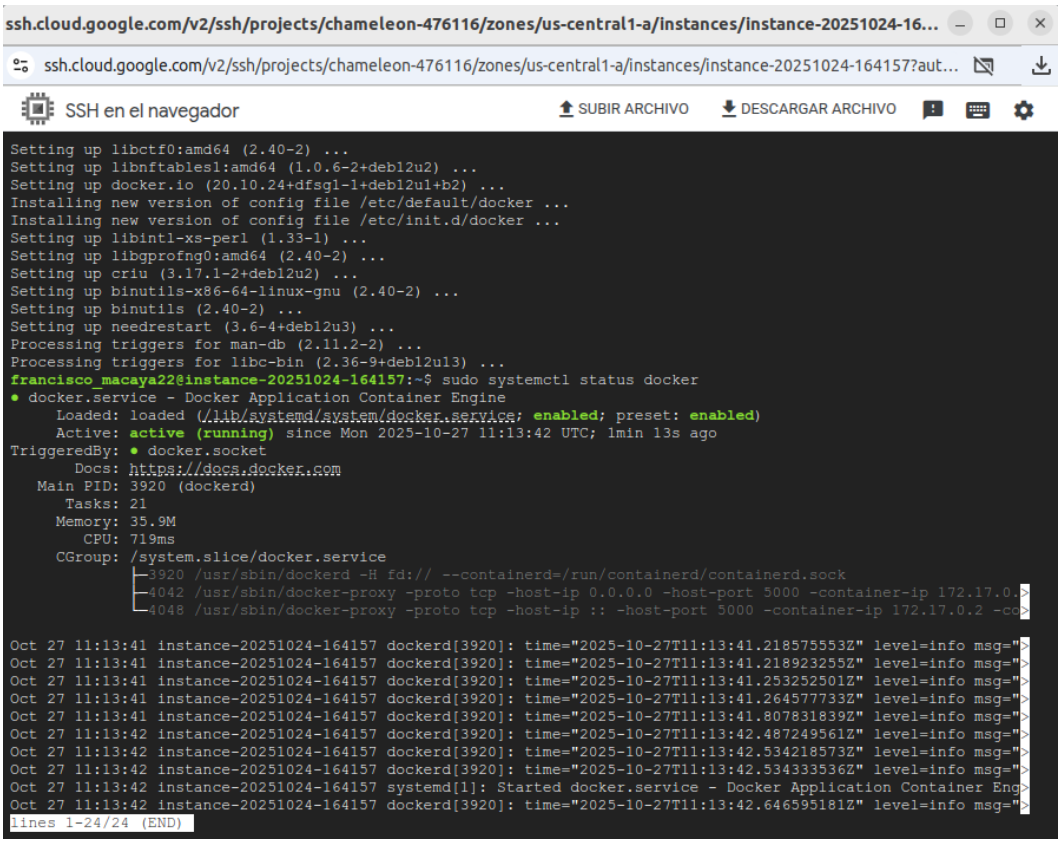
   Then, the following ones:

   sudo systemctl enable docker
   sudo systemctl start docker

   To check if Docker is running:

   sudo systemctl status docker

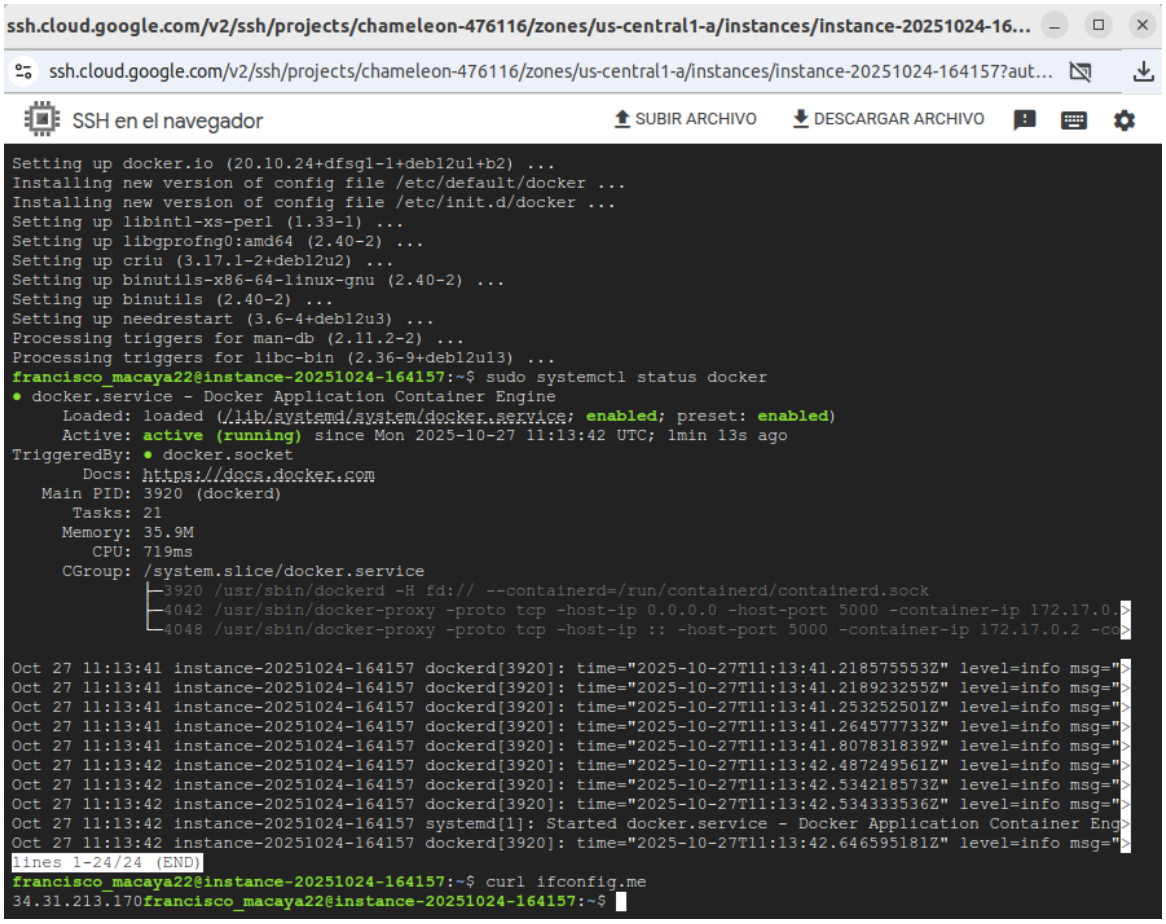
   As it can be seen in the following images:



3. *Public key:* Other important point about the VM is to connect to it, based on Github Actions. There are several way, but the better way to connect to the VM is geting the IP External, User, Key and port. In this case, the port is 22, the user name is francisco_macaya (the same of my user account name) and the external IP is 34.61.223.161. The external IP can be obtained by the following command:

curl ifconfig.me

and display like this:

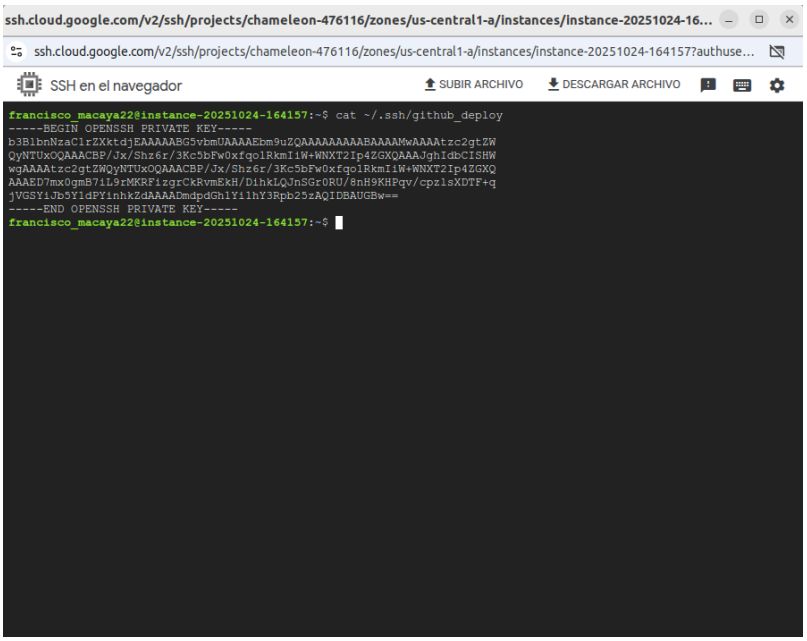


The remaining step was to create an SSH key, which enables secure remote connections to the VM through GitHub Actions. To generate the key, the following command was executed in the terminal:

ssh-keygen -t rsa -b 4096 -f ~/.ssh/github_deploy -C "francisco.macaya22@gmail.com"

This command generates both the private and public keys, which can then be displayed using the following command in the terminal:

cat ~/.ssh/github_deploy



With this key, I was able to access all the environment settings and configurations of the VM. Additionally, a public key must be configured on the VM to allow GitHub Actions to establish a secure connection with it. To perform this setup, the following command must be executed in the VM:

echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIE/8nH9KHPqv/cpzlsXDTF+qjVGSYiJb5Y1dPYinhkZd github-actions" >> ~/.ssh/authorized_keys

Then, the appropriate permissions must be set using the following commands:

chmod 600 ~/.ssh/authorized_keys
chmod 700 ~/.ssh

Finally, you need to get the username of the VM. You can obtain it with the following command:

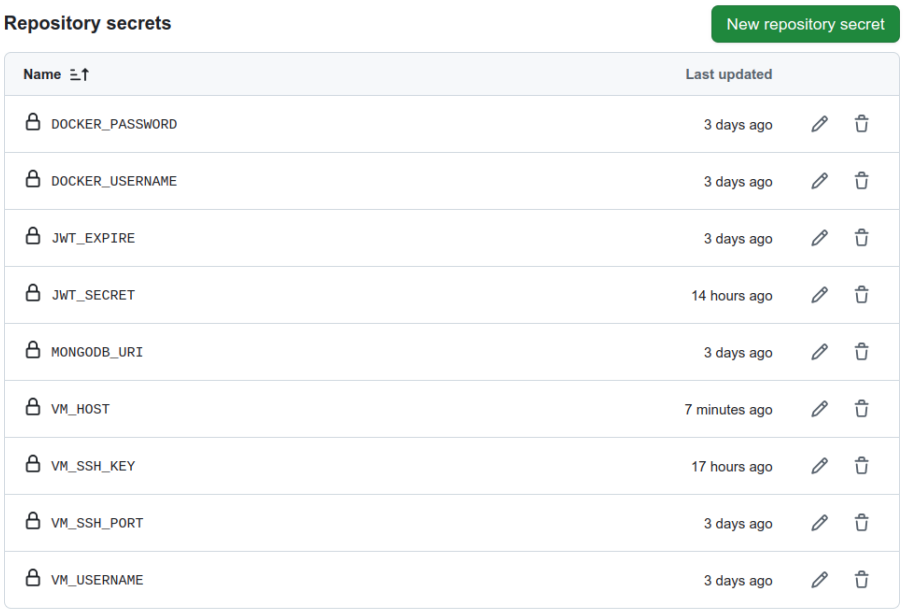
`whoami`

In my case, the username is **francisco_macaya22**.

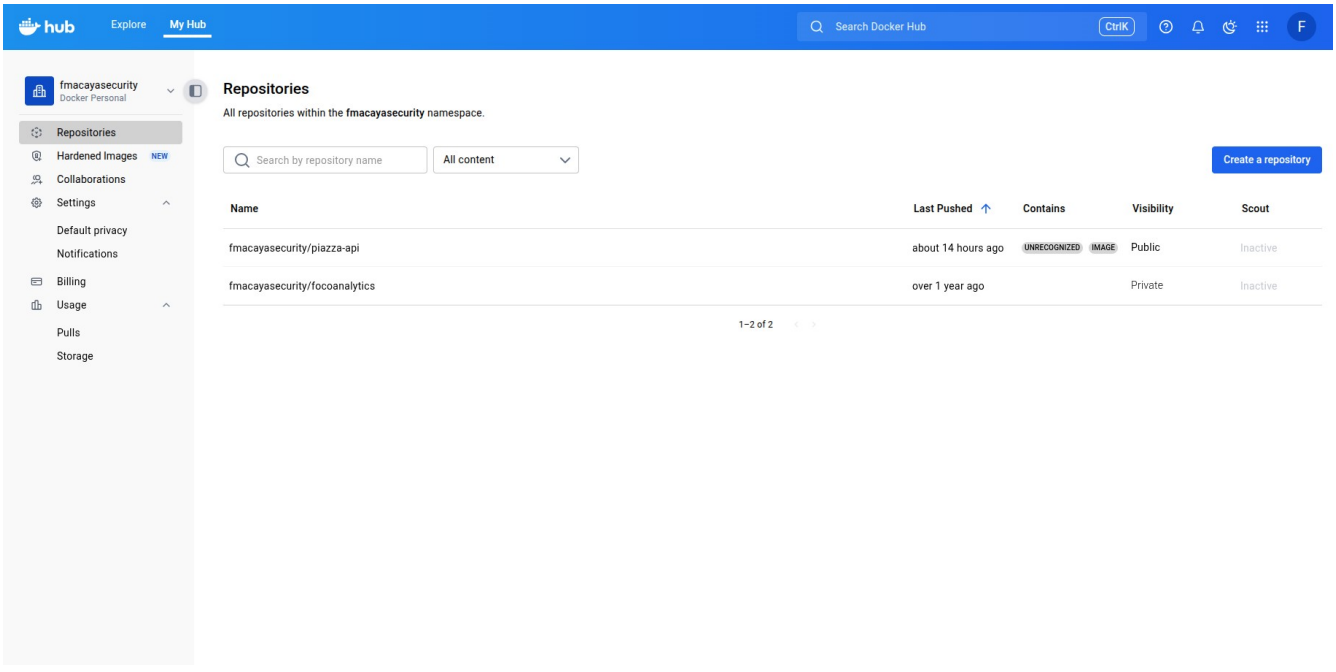
b-) Deploy your code in the virtual machine using your GitHub repository.

This task was completed using GitHub Actions, Docker Hub, and the Virtual Machine (VM). As is well known, the code can be automatically deployed through GitHub by following these steps:

1-Set up repository secrets in GitHub to securely store all application keys and environment variables outside the source code, preventing unauthorized access by others. The secrets are the following:



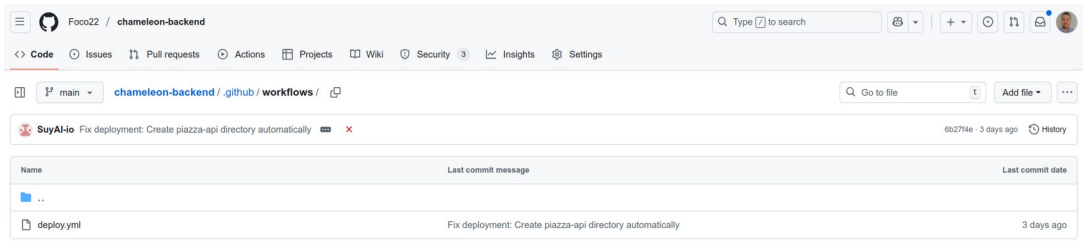
2- DockerHub: it was necessary to create an account on **Docker Hub**, as the Docker image is hosted there before being deployed to the VM. As shown in the image below, the Docker image was successfully uploaded to Docker Hub:



In Docker Hub, it is necessary to obtain your **username** and **password**, as these credentials are required for **GitHub Actions**.

3- Deploy.yml: This file is located in the project folder and is used to automatically deploy the code to the cloud. GitHub recognizes this file as a workflow configuration that connects the repository with the deployment environment.

The file is in the following images:



The main instructions contained in the file are as follows:

- Control which branches are affected by the workflow
- Connect to **Docker Hub**.
- Build and push the Docker image to **Docker Hub**.
- Connect to the **VM** via **SSH**.
- Pull the latest image from **Docker Hub** and deploy it on the VM using Docker.
- Test whether the deployment was successful.

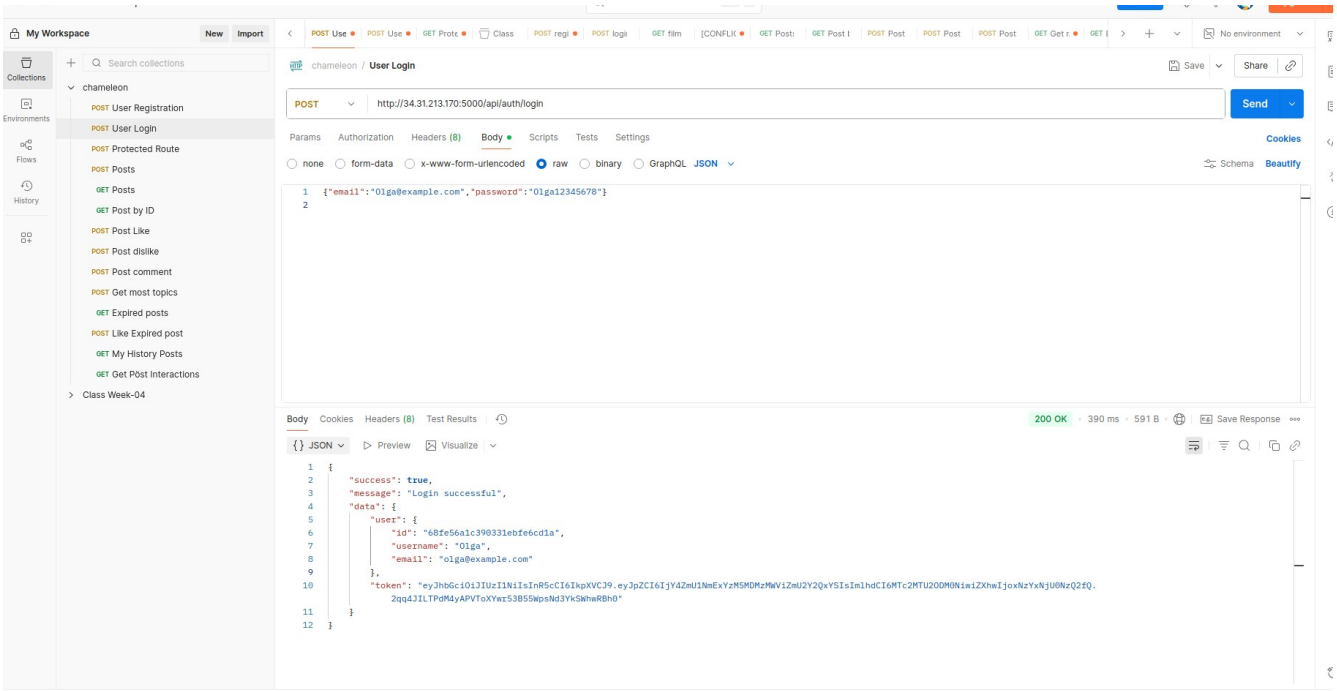
c-) REST API endpoints should be available under your virtual machine IP address based on examples and guidelines seen in class. Provide screenshots.

This section provides an example of how to execute a REST API request from the VM. The following chapter will include detailed API documentation; therefore, this section only presents a simple example of how a user can log in to the application.

After deploying the code to the VM, the application became accessible through the following endpoint:

URL: http://34.31.213.170:5000 *(the endpoint may not be working cause service was deleted)*

Using this URL, a user can log in to the application by sending the following request body and parameters:



As shown in the image, the user *Olga* can access the application using her username and password. The API responds with a success status and provides a **JWT token**, which allows the user to make authorized requests to other APIs within the application, confirming that she has the necessary permissions to execute them.

d) Provide a short description of your setup in the report.

The system setup is composed of the following components:

1. **MongoDB:** A database named *chameleon* was created to store the Piazza API data. It contains three collections:
    - **users** – stores user information,
    - **posts** – stores post data, and
    - **interactions** – stores the interactions performed on posts, such as likes, dislikes, and comments.
2. **Docker Hub:** The Docker image was built and uploaded to Docker Hub, allowing automatic connection and deployment to the VM.
3. **Virtual Machine (VM):** A VM was deployed on Google Cloud to run the backend of the application.

4. **Kubernetes:** A Kubernetes service was deployed on Google Cloud using three VMs, configured with a load balancer to distribute requests evenly across the nodes.
5. **GitHub:** The source code is hosted in the GitHub repository **Piazza-backend-project.** https://github.com/Foco22/Piazza-backend-project.

e) Discussion of installation and the structure of your folders.

The folders are the followings:

```
cam-backend/
├── src/
│   ├── config/
│   │   └── database.js          # MongoDB connection
│   ├── controllers/
│   │   ├── authController.js    # Authentication logic
│   │   └── postController.js    # Post CRUD and interactions
│   ├── middleware/
│   │   ├── auth.js              # JWT verification middleware
│   │   └── validation.js        # Input validation rules
│   ├── models/
│   │   ├── User.js              # User schema and model
│   │   ├── Post.js              # Post schema with expiration
│   │   └── Interaction.js       # Like/dislike/comment interactions
│   ├── routes/
│   │   ├── authRoutes.js        # Authentication endpoints
│   │   └── postRoutes.js        # Post and interaction endpoints
│   ├── utils/
│   │   └── generateToken.js     # JWT token generator
│   └── server.js                # Express app configuration
├── kubernetes/
│   ├── configmap.yaml           # Kubernetes ConfigMap
│   ├── deployment.yaml          # Kubernetes Deployment (3 replicas)
│   └── service.yaml             # LoadBalancer Service
├── .github/
│   └── workflows/
│       └── deploy.yml           # CI/CD pipeline for VM deployment
├── Dockerfile                   # Docker image configuration
├── .dockerignore
├── .env                         # Environment variables (local dev)
├── .gitignore
├── package.json
└── README.md
```

The explanation of the most important folders and files is as follows:

- **config:** Contains the configuration and connection settings for the MongoDB database.
- **controllers:** Includes all the logic and validation needed to execute the APIs and provide responses to client requests.
- **middleware:** Serves as a validation layer for incoming requests. For example, if a request does not include a valid token, the middleware prevents access and returns an appropriate error response.
- **models:** Defines the schema for each MongoDB collection.
- **routers:** Contains the different API endpoints and their associated routes.

- **utils:** Includes utility functions that can be reused across various parts of the application.
- **kubernetes:** Contains the configuration and scripts used to deploy the Kubernetes services in Google Cloud.
- **Dockerfile:** Defines all the commands required to build the Docker image and run the application in a consistent and isolated environment.

2- Phase B: Enforcing authentication/verification functionalities.

a) User management and JWT functionality using NodeJS.

At the first stage, a schema was created in MongoDB with the following structure:

Schema Definition:

- username: String (3–30 characters, unique, alphanumeric with underscores)
- email: String (unique, lowercase, validated format)
- password: String (hashed, minimum 6 characters, not returned by default)
- createdAt: Date (automatically generated)

Main Functionalities Implemented

1. Automatic Password Hashing: All passwords are securely hashed before being stored, ensuring that no one can access users' plain-text passwords.
2. Password Comparison Method: A built-in method allows comparing an entered password with its stored hashed version during the login process.
3. Automatic Data Sanitization: Input data is sanitized to prevent malicious or malformed input from being stored in the database.

Authentication Controller

Additionally, an authentication controller was developed to manage all endpoints related to user management, including registration, login, and authentication. The main controllers are:

1. User Registration:
   Validates whether the user already exists in the MongoDB collection. If the user exists, the API returns a **400 (Bad Request)** status. Otherwise, the new user is created.
2. User Login:
   Validates whether the user exists and whether the provided password matches the stored hash. Based on these checks, the system either grants access or returns an error response.
3. Get Current User:
   Retrieves the details of the currently authenticated user based on the provided token.

b) Authenticate users each time you perform any action point.

The middleware protect protege all the private routes (endpoints) based on a token. The routes are protected are:

Authentication routes (src/routes/authRoutes.js):
router.get('/me', protect, getMe);

Post routes (src/routes/postRoutes.js) - Todas usan protect:
router.post('/', protect, createPostValidation, validate, createPost);
router.post('/:id/like', protect, likePost);
router.post('/:id/dislike', protect, dislikePost);
router.post('/:id/comment', protect, commentValidation, validate, addComment);

History of users
router.get('/interactions/my-history', protect, getMyInteractions);

Based on this functionality, it is necessary to send requests using the user's authentication token. This mechanism prevents unauthorized access and protects the platform’s APIs from attacks by bots or other external services.

c) Complete Verification Process for Validation Purposes

For the validation process, several validation mechanisms were implemented, as described below:

1. Record Validations (User Registration):
   The user registration API includes multiple validation rules to ensure data integrity:
   - Username: Required field. The username must be between 3 and 40 characters, and can only contain letters, numbers, and underscores.
   - Email: Required field. The email must be unique and follow a valid email format.
   - Password: Required field. The password must be at least 6 characters long, and include uppercase letters, lowercase letters, and numbers.
2. Login Validations:
   During login, both the email and password fields are mandatory and cannot be empty.
3. Post Validations:
   When creating a post, the following rules are applied:
   - Title: Cannot be empty and must contain at least 3 characters.
   - Topic: Must be one of the predefined categories — Politics, Health, Sport, or Tech.

- Message: Cannot be empty and must contain between 10 and 5000 characters.
- Expiration Time: The minimum expiration time allowed is 1 minute.

3- Phase C: Development of Piazza RESTful APIs.

a) Implement basic functionalities provided in the Action points of Section 2.

The primary functionalities implemented in the system are described as follows:

- Name: user registration:
  Service Description: This endpoint allows the registration of a new user in the application.
  URL: http://localhost:5000/api/auth/register
  Body: {"username": "testuser",
  "email": "test@example.com",
  "password": "Test123"}
  Type: POST

- Name: Login application:
  Service Description: This endpoint allows to login into the application.
  URL: http://localhost:5000/api/auth/login
  Body: {"email": "test@example.com",
  "password": "Test123"}
  Type: POST

- Name: create post:
  Service Description: This endpoint allows to create a new post in the application.
  URL: http://localhost:5000/api/posts
  Authorization: Bearer YOUR_TOKEN
  Body: {"title": "My First Post",
  "topics": ["Tech"],
  "message": "This is my first post on Piazza!",
  "expirationMinutes": 60}
  Type: POST

- Name: collect post by ID:
  Service Description: This endpoint allows to get a post by its ID.
  URL: http://localhost:5000/api/posts/::id
  Authorization: Bearer YOUR_TOKEN
  Type: GET

- Name: Browse Post:
  Service Description: This endpoint allows to search the posts by topic in the application.
  URL: http://localhost:5000/api/posts?topic=Tech
  Type: GET

- Name: Like post:
  Service Description: This endpoint allows to like a post in the application.
  URL: http://localhost:5000/api/posts/POST_ID/like
  Authorization: Bearer YOUR_TOKEN
  Type: POST

- Name: DisLike post:
  Service Description: This endpoint allows to dislike a post in the application.
  URL: http://localhost:5000/api/posts/POST_ID/dislike
  Authorization: Bearer YOUR_TOKEN
  Type: POST

- Name: Add Comment:
  Service Description: This endpoint allows to add a comment to the post.
  URL: http://localhost:5000/api/posts
  Authorization: Bearer YOUR_TOKEN
  Body: {"title": "My First Post",
  "topics": ["Tech"]}
  Type: POST

- Name: most active posts by topic :
  Service Description: This endpoint allows to get the most active posts by topic
  URL: http://localhost:5000/api/posts/most-active/topic
  Authorization: Bearer YOUR_TOKEN
  Type: Get

- Name: History the expired post by topic :
  Service Description: This endpoint allows to get the expired posts by topic.
  URL: http://localhost:5000/api/posts/expired/Tech
  Authorization: Bearer YOUR_TOKEN
  Type: Get

*4- Phase D: T*esting your *application.*

Disclaimer: *This section was completed using a front end in the application because it was easier for me to demonstrate all the test cases. This front end was connected to the backend locally. In the video, I showed how posts appear in the application when they are created. As I mentioned before, the front end directly consumes the APIs developed for the coursework (such as "get all posts"), and those APIs are connected to MongoDB.*

There are some cases where I showed the posts directly in the front end without explicitly displaying them in the database; however, since the front end reads the endpoints from the Piazza services, it indirectly demonstrates that the posts were successfully created in MongoDB.

I hope this is an easier and better way to demonstrate the application, rather than taking several screenshots for each test case.

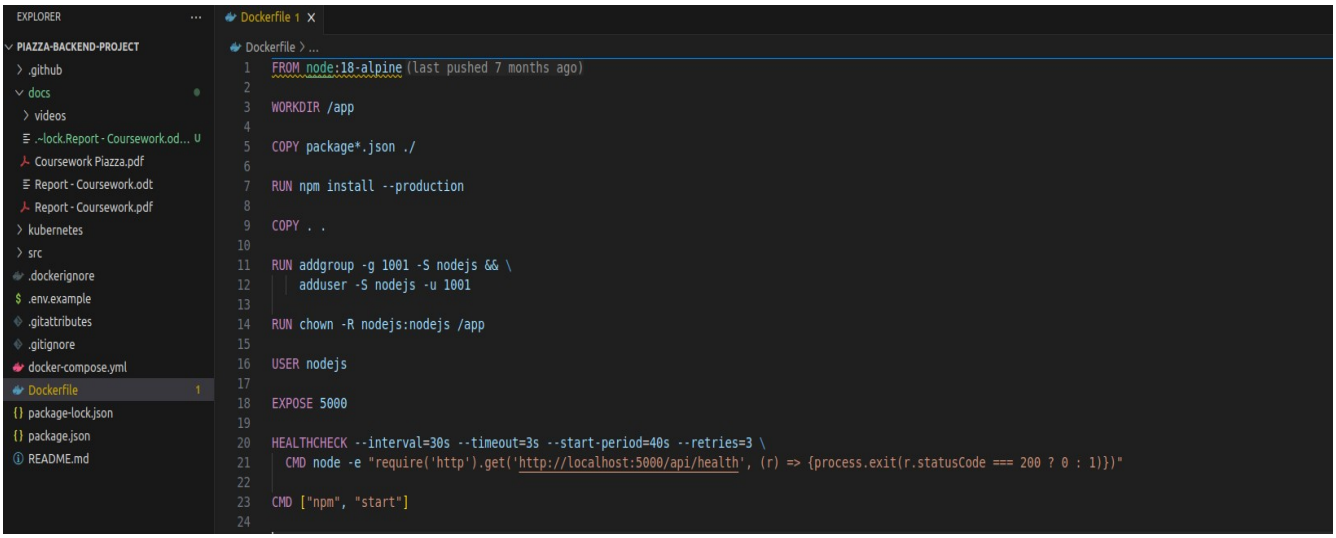
This section will be demonstrated in some videos. It was decided to present it in this format because it is easier to showcase the functionality directly through the front-end rather than capturing individual screenshots for each test case. The videos are the following:

Note: The *docs* folder is the main root of the code.

1. TC1: docs/videos/TC_1.mp4
2. TC2: docs/videos/TC_2.mp4
3. TC3: docs/videos/TC_3.mp4
4. TC4: docs/videos/TC_4.mp4
5. TC5: docs/videos/TC_5.mp4
6. TC6: docs/videos/TC_6.mp4
7. TC7: docs/videos/TC_7.mp4
8. TC8: docs/videos/TC_8.mp4
9. TC9: docs/videos/TC_9.mp4
10. TC10: docs/videos/TC_10.mp4
11. TC11: docs/videos/TC_11.mp4
12. TC12: docs/videos/TC_12.mp4
13. TC13: docs/videos/TC_13.mp4
14. TC14: docs/videos/TC_14.mp4
15. TC15: docs/videos/TC_15.mp4
16. TC16: docs/videos/TC_16.mp4
17. TC17: docs/videos/TC_17.mp4
18. TC18: docs/videos/TC_18.mp4
19. TC19: docs/videos/TC_19.mp4
20. TC20: docs/videos/TC_20.mp4

5- Phase E: Deploy your Piazza project into a VM using Docker

The backend was encapsulated in a Docker container based on the following Dockerfile:



As can be seen, the Dockerfile installs node:18-alpine, along with the necessary packages for the application. It then exposes the port and runs the Node.js server using the npm start command.

6- Phase F: Deploy your application in Kubernetes

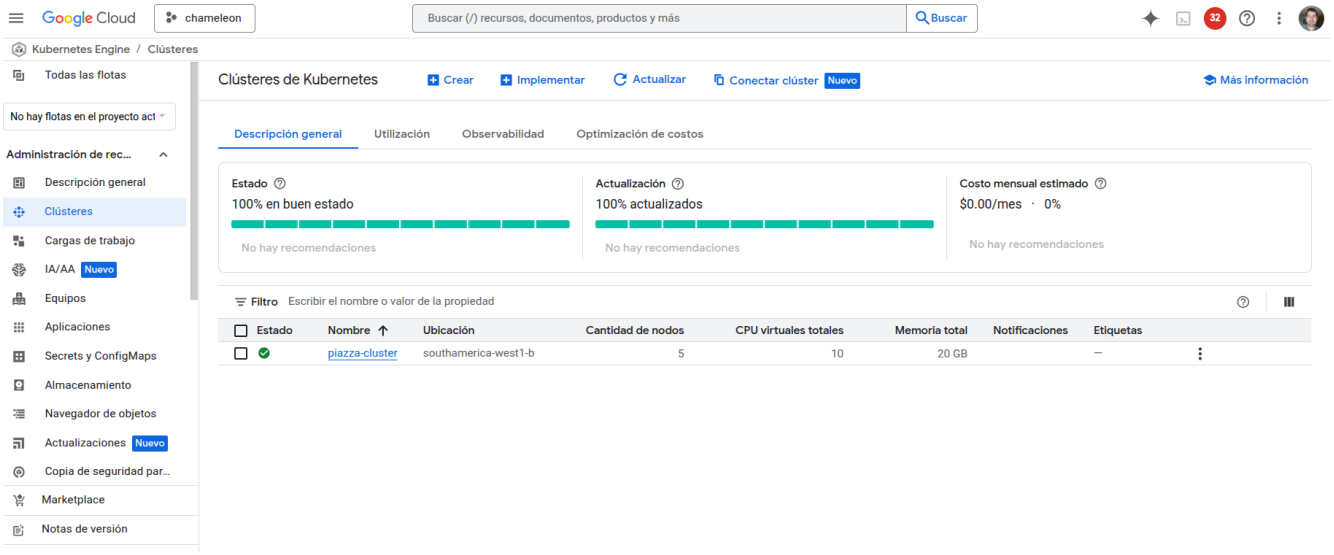
The application was deployed on a Kubernetes cluster with three replicas, using a LoadBalancer service to provide higher stability and resilience. This configuration helps distribute incoming requests evenly across the replicas, allowing the system to handle a larger flow of user requests efficiently.

The workflow functions as follows:

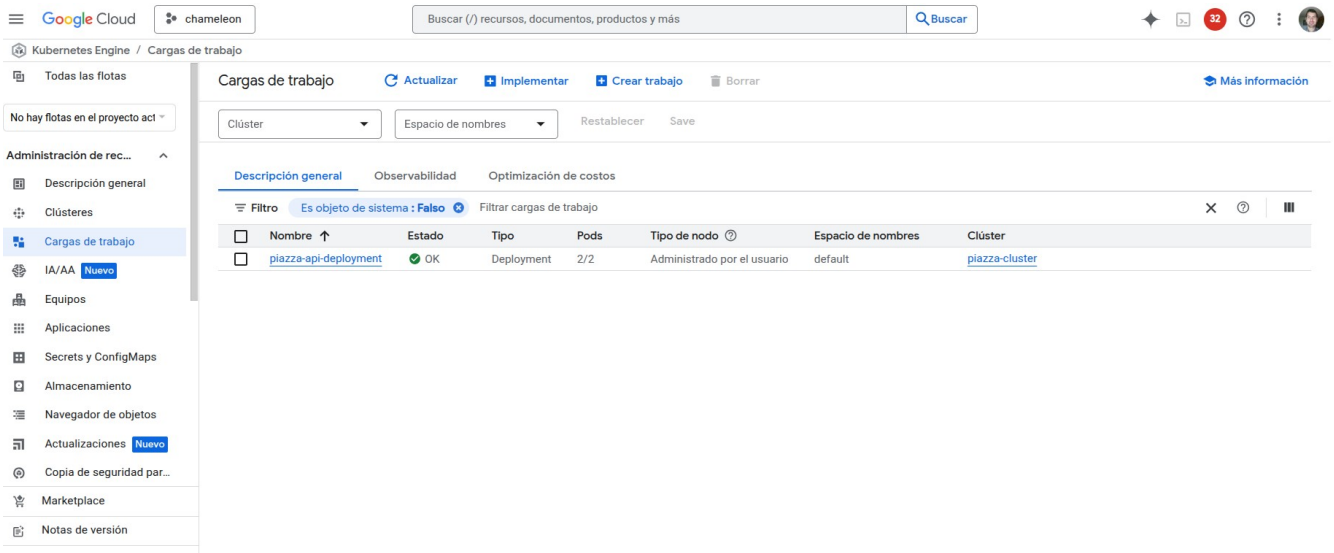
1. The user sends a request to the external IP of the Load Balancer. In this case, the IP is 34.136.8.35.
2. The Google Cloud Load Balancer receives the request. If it is the user's first request, the load balancer selects a pod using a round-robin algorithm. If the user has already made a previous request, they are redirected to the same pod to maintain session consistency.
3. The Load Balancer then forwards the request to the selected pod. For example, if the user is new, they are routed to the first available pod.
4. Inside the pod, traffic first reaches port 80, which is then redirected by kube-proxy to port 5000, where the application is running.
5. The Express.js server listens on localhost:5000 and receives the request (e.g., /api/auth/login). The request passes through the middleware layer and is handled by the corresponding controller

(e.g., authController.login), which verifies the credentials stored in MongoDB, generates a JWT token, and returns the response to the user.

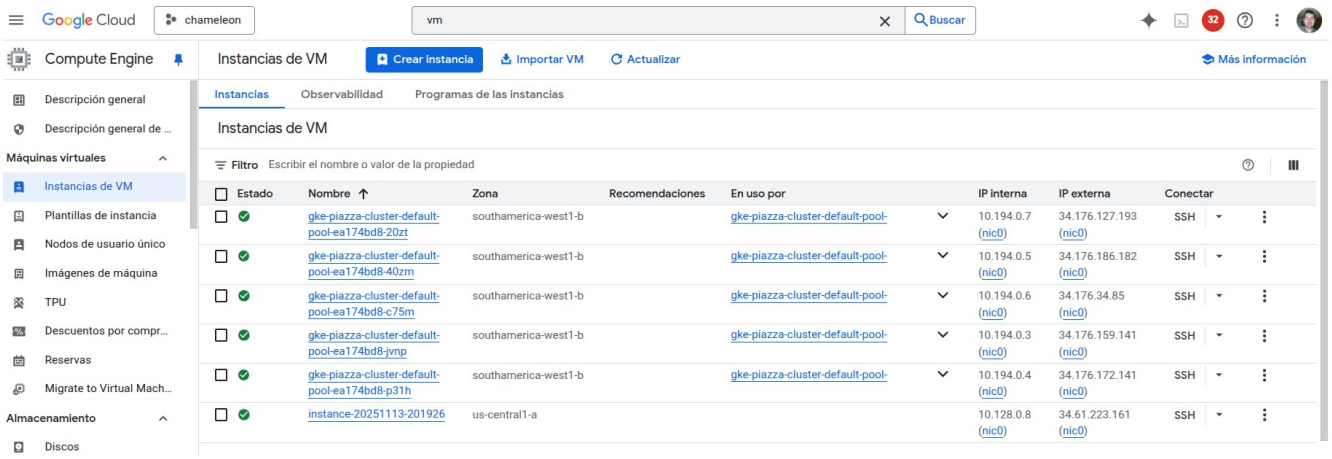
As proof of deployment, the following figure shows the cluster created in Google Cloud, which hosts the application and its associated services.



In addition, the Load Balancer configuration is shown in the following figure:



Finally, the virtual machines (VMs) created for the cluster are shown below:



It should be noted that, to ensure the proper operation of Kubernetes together with the load balancer, the following commands were used:

1. Inside Google Cloud, it was accessed the Cloud Shell and used the following command

   kubectl get services

2. Due to the database secrets, a set of keys was created for Kubernetes using the following command:

```
kubectl create secret generic piazza-secrets \
  --from-literal=MONGODB_URI='mongodb+srv://youruser:yourpass@cluster.mongodb.net/piazza?retryWrites=true&w=majority' \
  --from-literal=JWT_SECRET='your-super-secret-jwt-key-here'
```

Note: The bold words contains the secret of the mongodb.

3. The project repository was cloned using:

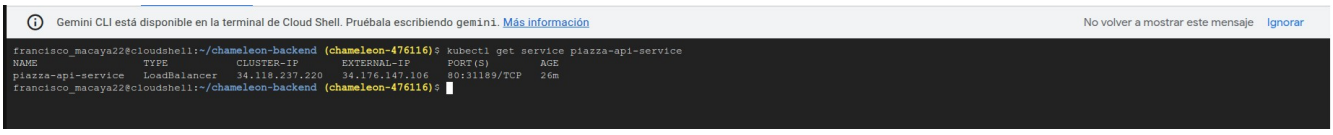
```
git clone https://github.com/Foco22/Piazza-backend-project.git
cd  Piazza-backend-project
```

4. The Kubernetes manifests (YAML files) were applied using the following command:

```
kubectl apply -f kubernetes/configmap.yaml
kubectl apply -f kubernetes/deployment.yaml
kubectl apply -f kubernetes/service.yaml
```

5. The status and the external IP were checked using:

```
kubectl get pods
kubectl get service piazza-api-service
```



6- Phase G: Present your solution in a technical report

As mentioned in the previous section, this part of the technical report includes all the information that was not presented earlier. It provides additional details about the system's configuration and development environment.

This section covers the following topics:

1. **Technical Stack**

The main technologies and frameworks used in the coursework are as follows:

Layer	Technology	Version	Purpose
Backend	Node.js	v18.20.4	Runtime environment
Framework	Express.js	5.1.0	Web framework
Database	MongoDB Atlas	8.19.2	Web framework
Authentication	JWT	9.0.2	Token-based auth
Containerization	Docker	26.1.1	Application packaging
Orchestration	Kubernetes (GKE)	N/A	Container management
CI/CD	GitHub Actions	N/A	Automated deployment
Cloud Provider	Google Cloud Platform	N/A	Infrastructure
Docker-Compose	Docker-Compose	N/A	Local deployment

2. **Deployment Models**

As required by the project, the application was deployed using two different deployment models: one based on a Virtual Machine (VM) and the other on Kubernetes.

Model 1: VM with Docker (Development)

- Single Google Compute Engine VM
- Docker containerized environment
- CI/CD pipeline implemented with GitHub Actions
- Public IP: http://34.31.213.170:5000 (the endpoints may not be works due to cost savings)

Model 2: Kubernetes (Production)

- Three replicas for high availability
- Automatic load balancing to distribute incoming requests

- Self-healing capabilities for fault tolerance
- Public IP: **http://34.176.147.106** (the endpoints may not be works due to cost savings)

Model 3: Local deployment with Docker-compose.

As Docker Compose is not taught in the course, but I have used it in my work experience, I decided to use it to deploy the application in an easier way. The magic of Docker Compose is that it allows you to run different containers on different ports on your computer, which is very useful if you want to install a database locally and run it together with your application.

In this case, MongoDB can also be installed locally on your computer, so I am able to run my backend with a local MongoDB instance while also testing that it works with the connection to MongoDB Cloud.

For this purpose, a `docker-compose.yml` file was created, allowing the Piazza API to run on port 5000, along with MongoDB on port 8001. To run this setup, you must use the following command:

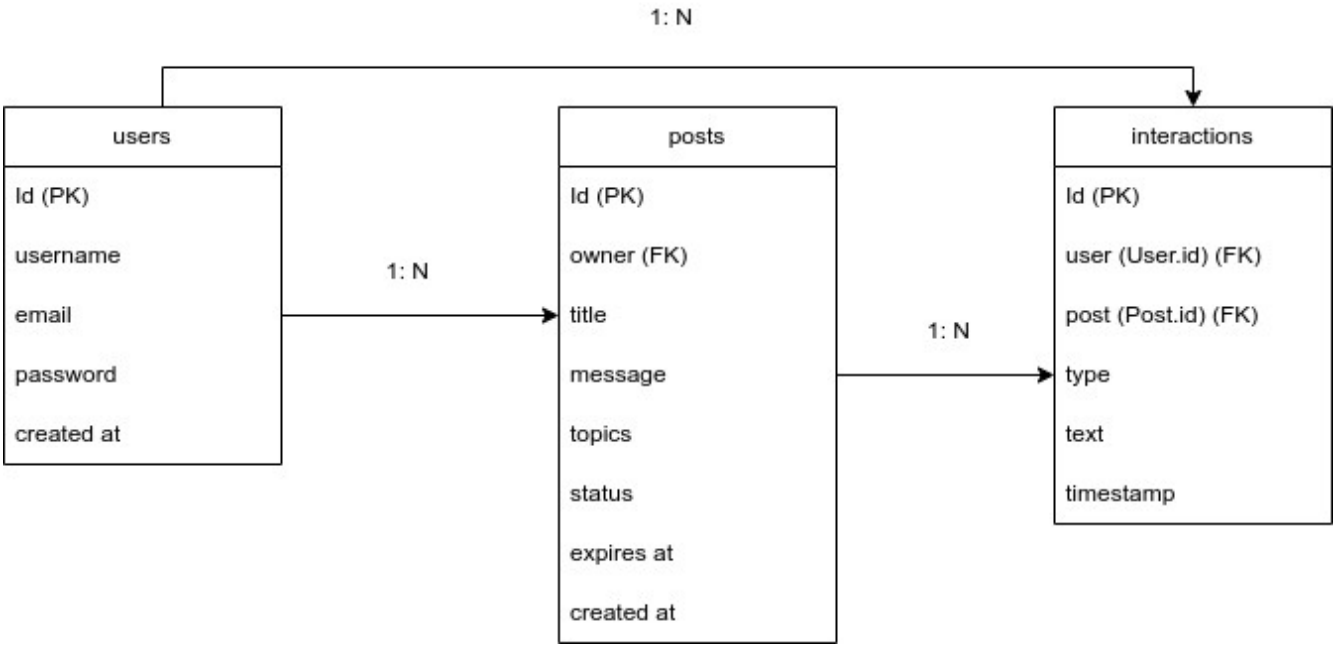
- Create a `.env` file with the required configuration variables.
- Run `docker-compose up -d`.
- Check the running services using `docker-compose ps`.
- If you want to stop the services, run the following command: `docker-compose down`.

PD: If you don’t have Docker Compose installed on your computer, you must install it depending on your operating system.

Windows: https://docs.docker.com/compose/install/

3. **Database Design**

The database design is the following:



The database contains a users collection that stores user information such as username, email, and password (secured with a JWT token).

The posts collection stores all the posts created within the application. Each post is linked to a user through a foreign key reference (owner), and includes fields such as title, message, topic, status, and expiration date.

Finally, the interactions collection records the different types of interactions that users can have with posts, such as likes, dislikes, and comments. If the interaction is a comment, the collection also stores the text of the comment.

4. **References**

1. Express.js Official Docs: https://expressjs.com/
2. Mongoose Documentation: https://mongoosejs.com/docs/
3. JSON Web Tokens: https://jwt.io/introduction
4. Kubernetes Concepts: https://kubernetes.io/docs/concepts/
6. MongoDB Atlas: https://www.mongodb.com/cloud/atlas
7. Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine
8. Docker Hub: https://hub.docker.com/