

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 screenshot shows the Google Cloud Platform Compute Engine interface. The left sidebar shows navigation options like Compute Engine, Descripción general, Descripción general de..., Máquinas virtuales, Instancias de VM, Plantillas de instancia, Nodos de usuario único, Imágenes de máquina, TPU, Descuentos por compr..., Reservas, Migrate to Virtual Mach..., Almacenamiento, Discos, Grupos de almacenamiento, Instantáneas, Imágenes, Replicación asíncrona, Grupos de instancias, Marketplace, and Notas de versión. The main panel displays the details of a VM instance named 'instance-20251024-164157'. The 'Detalles' tab is selected, showing the following information:

Nombre	instance-20251024-164157
ID de instancia	8296988847327547913
Descripción	Ninguna
Tipo	Instancia
Estado	Activa
Fecha y hora de creación	oct 24, 2025, 5:44:25 p.m. UTC+01:00
Ubicación	us-central1-a
Imagen de origen del disco de arranque	debian-12-bookworm-v20251014
Arquitectura del disco de arranque	X86_64
Tipo de licencia del disco de arranque	Sin cargo
Plantilla de instancia	Ninguna
En uso por	Ninguno
Host físico	Ninguna
Estado del mantenimiento	—

Below the details, there are sections for 'Registros' (Registers) and 'Información básica' (Basic Information). At the bottom, there are links for 'Código equivalente' (Equivalent code) and a copy link: 'https://console.cloud.google.com/compute/instances?tab=images&ini=1&input=%2Bh1%2B&project=chameleond47a116'.

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 necessary to run the command `sudo apt upgrade -y`, so all the packages are installed with the newest libraries 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
```

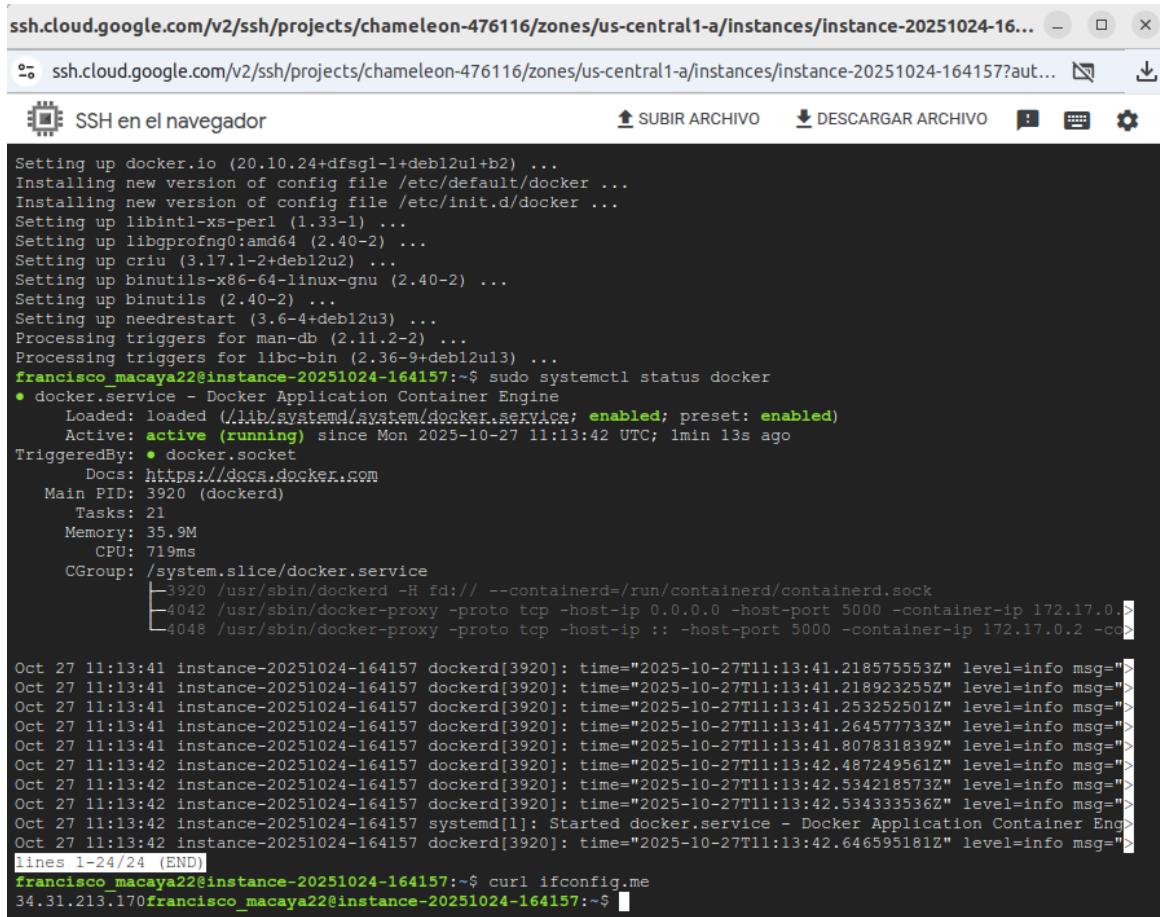
As it can be seen in the following images:

```
ssh.cloud.google.com/v2/ssh/projects/chameleon-476116/zones/us-central1-a/instances/instance-20251024-16...
ssh.cloud.google.com/v2/ssh/projects/chameleon-476116/zones/us-central1-a/instances/instance-20251024-164157?aut...
SSH en el navegador SUBIR ARCHIVO DESCARGAR ARCHIVO
Setting up libctf0:amd64 (2.40-2) ...
Setting up libnftables:amd64 (1.0.6-2+deb12u2) ...
Setting up docker.io (20.10.24+dfsg1-1+deb12u1+b2) ...
Installing new version of config file /etc/default/docker ...
Installing new version of config file /etc/init.d/docker ...
Setting up libintl-xs-perl (1.33-1) ...
Setting up libgprofng0:amd64 (2.40-2) ...
Setting up criu (3.17.1-2+deb12u2) ...
Setting up binutils-x86_64-linux-gnu (2.40-2) ...
Setting up binutils (2.40-2) ...
Setting up needrestart (3.6-4+deb12u3) ...
Processing triggers for man-db (2.11.2-2) ...
Processing triggers for libc-bin (2.36-9+deb12u13) ...
francisco_macaya22@instance-20251024-164157:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; preset: enabled)
  Active: active (running) since Mon 2025-10-27 11:13:42 UTC; 1min 13s ago
TriggeredBy: ● docker.socket
    Docs: https://docs.docker.com
  Main PID: 3920 (dockerd)
    Tasks: 21
   Memory: 35.9M
      CPU: 719ms
     CGroup: /system.slice/docker.service
             └─3920 /usr/sbin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
               ├─4042 /usr/sbin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 5000 -container-ip 172.17.0.0>
               ├─4048 /usr/sbin/docker-proxy -proto tcp -host-ip :: -host-port 5000 -container-ip 172.17.0.2 -co>
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.218575553Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.218923252Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.253252501Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.264577733Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.807831839Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.487249561Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.534218573Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.534333536Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 systemd[1]: Started docker.service - Docker Application Container Eng>
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.646595181Z" level=info msg=">
lines 1-24/24 (END)
```

3. *Public key*: Other important point about the VM is to connect to it, based on Github Actions. There are several ways, but the better way to connect to the VM is getting 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 screenshot shows an SSH session on a Linux system. The terminal window has a title bar with the URL "ssh.cloud.google.com/v2/ssh/projects/chameleon-476116/zones/us-central1-a/instances/instance-20251024-16..." and a status bar with "ssh.cloud.google.com/v2/ssh/projects/chameleon-476116/zones/us-central1-a/instances/instance-20251024-164157?aut...". The main area of the terminal displays the output of the curl command, which includes the IP address 34.31.213.170. The terminal interface includes standard Linux-style buttons for file operations (SUBIR ARCHIVO, DESCARGAR ARCHIVO) and a gear icon for settings.

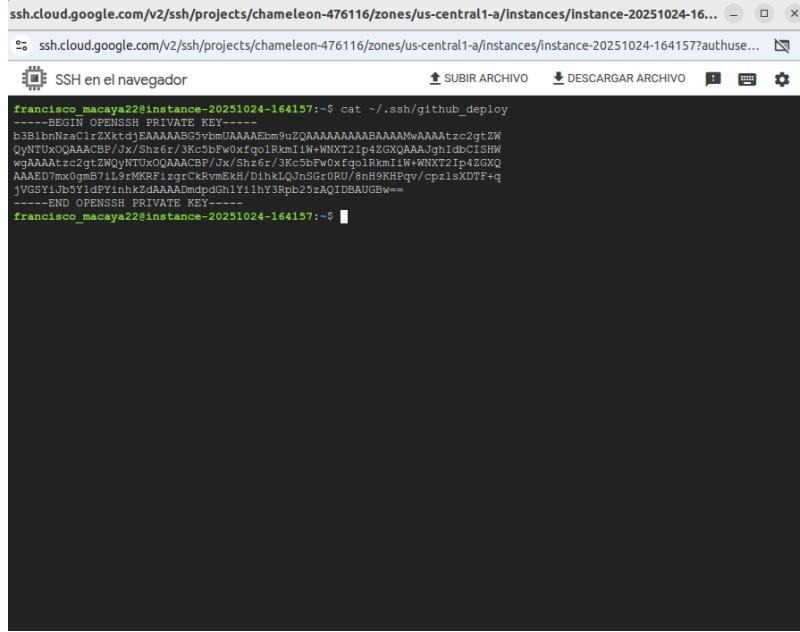
```
Setting up docker.io (20.10.24+dfsg1-1+deb12u1+b2) ...
Installing new version of config file /etc/default/docker ...
Installing new version of config file /etc/init.d/docker ...
Setting up libintl-xs-perl (1.33-1) ...
Setting up libgprofng0:amd64 (2.40-2) ...
Setting up criu (3.17.1-2+deb12u2) ...
Setting up binutils-x86-64-linux-gnu (2.40-2) ...
Setting up binutils (2.40-2) ...
Setting up needrestart (3.6-4+deb12u3) ...
Processing triggers for man-db (2.11.2-2) ...
Processing triggers for libc-bin (2.36-9+deb12u13) ...
francisco_macaya22@instance-20251024-164157:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; preset: enabled)
  Active: active (running) since Mon 2025-10-27 11:13:42 UTC; 1min 13s ago
TriggeredBy: ● docker.socket
  Docs: https://docs.docker.com
 Main PID: 3920 (dockerd)
    Tasks: 21
   Memory: 35.9M
     CPU: 719ms
      CGroup: /system.slice/docker.service
              └─3920 /usr/sbin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
                  ├─4042 /usr/sbin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 5000 -container-ip 172.17.0.0
                  ├─4048 /usr/sbin/docker-proxy -proto tcp -host-ip :: -host-port 5000 -container-ip 172.17.0.2 -cd>
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.21857553Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.218923255Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.253252501Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.264577733Z" level=info msg=">
Oct 27 11:13:41 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:41.807831839Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.487249561Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.534218573Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.534333536Z" level=info msg=">
Oct 27 11:13:42 instance-20251024-164157 systemd[1]: Started docker.service - Docker Application Container Eng
Oct 27 11:13:42 instance-20251024-164157 dockerd[3920]: time="2025-10-27T11:13:42.646595181Z" level=info msg=">
lines 1-24/24 (END)
francisco_macaya22@instance-20251024-164157:~$ curl ifconfig.me
34.31.213.170francisco_macaya22@instance-20251024-164157:~$
```

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:

Repository secrets		New repository secret
Name	Last updated	
DOCKER_PASSWORD	3 days ago	
DOCKER_USERNAME	3 days ago	
JWT_EXPIRE	3 days ago	
JWT_SECRET	14 hours ago	
MONGODB_URI	3 days ago	
VM_HOST	7 minutes ago	
VM_SSH_KEY	17 hours ago	
VM_SSH_PORT	3 days ago	
VM_USERNAME	3 days ago	

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:

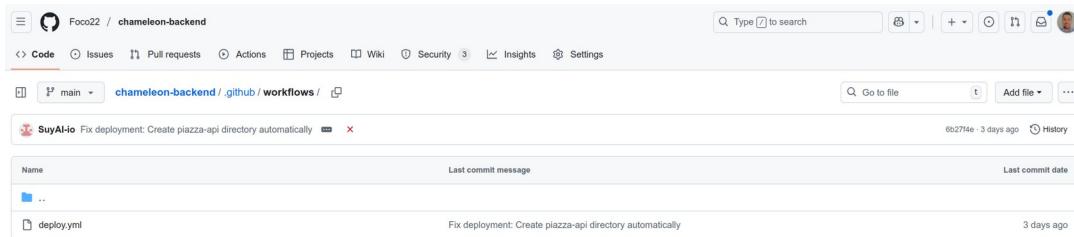
The screenshot shows the Docker Hub interface. On the left, there's a sidebar with options like 'Hardened Images', 'Collaborations', 'Settings', 'Billing', 'Usage', 'Pulls', and 'Storage'. The main area is titled 'Repositories' and shows two entries:

Name	Last Pushed	Contains	Visibility	Scout
fmacayasecurity/piazza-api	about 14 hours ago	UNRECOGNIZED IMAGE	Public	Inactive
fmacayasecurity/focoanalytics	over 1 year ago		Private	Inactive

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:



A screenshot of a GitHub repository interface. The repository is named 'Foco22 / chameleon-backend'. The 'Code' tab is selected, showing the 'main' branch. Under 'Code', there is a link to '.github/workflows/'. Inside this folder, there is a single file named 'deploy.yml'. The commit history for this file shows one entry from 'SuyAI-io' with the message 'Fix deployment: Create piazza-api directory automatically', made 3 days ago.

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:

```

POST http://34.31.213.170:5000/api/auth/login
{
  "email": "Olga@example.com",
  "password": "Olga12345678"
}
200 OK
{
  "success": true,
  "message": "Login successful",
  "data": {
    "user": {
      "id": "68fb5e0a1c390331ef6fe6cd1a",
      "username": "Olga",
      "email": "olga@example.com"
    },
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4ZmU1NmExYzM5MDMzMmV1ZmU2Y2QxYSIsImIhdCI6MTc2MTU2MDM0Niw1ZKhwIjoxNzYnJU0NzQ2fQ.2qq4JILTPdMyAPVtXYwz5B855WpsNd3YkSwRaRBh0"
  }
}
  
```

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: Testing 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

The evidence for this test case is as follows:

Olga:

The screenshot shows the Postman interface with a collection named "Piazza Backend". A POST request is made to `http://136.112.162.81:5000/api/auth/register`. The request body contains the JSON object `{"username": "olga", "email": "olga@gmail.com", "password": "Olga12345678"}`. The response status is `201 Created` with a response time of 645 ms and a response size of 645 B. The response body is a JSON object with fields `success`, `message`, and `data`. The `data` field contains a user object with properties `id`, `username`, `email`, `createdAt`, and a token string.

```

1 {
2     "success": true,
3     "message": "User registered successfully",
4     "data": {
5         "user": {
6             "id": "693560fbfdcc06ee4de0d99",
7             "username": "olga",
8             "email": "olga@gmail.com",
9             "createdAt": "2025-12-07T11:32:15.778Z"
10            },
11            "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IjY5MzU2NjBjY2MnMjNGR1M0QmOsIiAiLCI6MTc2NTExKzEzImizXheIjoXKzY1MTkzNTH2I0RLkEDgX5eL4XUNBqeyvT15SAubxMta-E6g9OfrYBI"
12        }
13    }
  
```

Nick:

The screenshot shows the Postman interface with a collection named "Piazza Backend". A POST request is made to `http://136.112.162.81:5000/api/auth/register`. The request body contains the JSON object `{"username": "nick", "email": "nick@gmail.com", "password": "Nick12345678"}`. The response status is `201 Created` with a response time of 633 ms and a response size of 645 B. The response body is a JSON object with fields `success`, `message`, and `data`. The `data` field contains a user object with properties `id`, `username`, `email`, `createdAt`, and a token string.

```

1 {
2     "success": true,
3     "message": "User registered successfully",
4     "data": {
5         "user": {
6             "id": "693560fbfdcc06ee4de0d99",
7             "username": "nick",
8             "email": "nick@gmail.com",
9             "createdAt": "2025-12-07T11:33:30.971Z"
10            },
11            "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IjY5MzU2NjBjY2MnMjNGR1M0QmOsIiAiLCI6MTc2NTExKzEzImizXheIjoXKzY1MTkzNTH2I0HbILlevlyjSBR6eMMUVfbzyuGP8ihbpJFV3kLU668g"
12        }
13    }
  
```

Mary:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to "User Registration" with the URL `http://136.112.162.81:5000/api/auth/register`. The request body is raw JSON:

```
{"username": "mary", "email": "mary@gmail.com", "password": "Mazy12345678"}
```

. The response status is 201 Created, with a response time of 622 ms and a response size of 645 B. The response body is:

```
{ "success": true, "message": "User registered successfully", "data": { "user": { "id": "e935663b4cc0c6ee4de0d0f", "username": "mary", "email": "mary@gmail.com", "createdAt": "2025-12-07T11:34:08.238Z" }, "token": "eyJhbGciOiJIUzI1NiIsInR5cCIkIkpXVCj9.eyJpZC1oIjY5MzU2NjMyYjY2MmNvIjNGR1MGQwZisImh0Mtc2NTEnZI00CwiZXhwIjoxNzY1MTkzMjQ4fQ.C_ipIcgBa4yrlE0sZXCnDNLyJ999RqiiJM511_RW3ZU" }}
```

Nestor:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to "User Registration" with the URL `http://136.112.162.81:5000/api/auth/register`. The request body is raw JSON:

```
{"username": "nestor", "email": "nestor@gmail.com", "password": "Nestor12345678"}
```

. The response status is 201 Created, with a response time of 658 ms and a response size of 649 B. The response body is:

```
{ "success": true, "message": "User registered successfully", "data": { "user": { "id": "e9356650b4cc0c6ee4de0d12", "username": "nestor", "email": "nestor@gmail.com", "createdAt": "2025-12-07T11:34:40.993Z" }, "token": "eyJhbGciOiJIUzI1NiIsInR5cCIkIkpXVCj9.eyJpZC1oIjY5MzU2NjUyYjY2MmNvIjNGR1MGQwZisImh0Mtc2NTEnZI4MSwiZXhwIjoxNzY1MTkzMjgxfQ.mFVfdRInv3W2vmVpGeiy4WnuI5SSacdyLnoRzdVXI" }}
```

2. TC2: docs/videos/TC_2.mp4

Nestor:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to "User Login" with the URL `http://136.112.162.81:5000/api/auth/login`. The request body is JSON containing the email and password for "nestor@gmail.com". The response status is 200 OK, and the response body is a JSON object indicating a successful login with user information and a token.

```

1 {
2   "success": true,
3   "message": "Login successful",
4   "data": {
5     "user": {
6       "id": "e935665984ccc0ee4de0d12",
7       "username": "nestor",
8       "email": "nestor@gmail.com"
9     },
10    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6Ijy5MzU2NjUxYjRjY2MwNnVlNGRlMGQwMjIsImh0Ci0MTC2NTEmNzQxNzIwIjoxNzY1MTkzODE0fQ.eyJfdHl0QjAuZx2zRwIDltLtb2YJNwnnDdytBKUC0KE"
11  }
12 }

```

Mary:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to "User Login" with the URL `http://136.112.162.81:5000/api/auth/login`. The request body is JSON containing the email and password for "mary@gmail.com". The response status is 200 OK, and the response body is a JSON object indicating a successful login with user information and a token.

```

1 {
2   "success": true,
3   "message": "Login successful",
4   "data": {
5     "user": {
6       "id": "e935665984ccc0ee4de0d8f",
7       "username": "mary",
8       "email": "mary@gmail.com"
9     },
10    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6Ijy5MzU2NjUxYjRjY2MwNnVlNGRlMGQwMjIsImh0Ci0MTC2NTEmNzQxNzIwIjoxNzY1MTkzOTE0fQ.3dFNkIArtI4Xqxa8TmrcoequlB1Pp6Xc1tyxxt3K7s"
11  }
12 }

```

Olga:

The screenshot shows the Postman interface with the following details:

- Collection:** My Workspace
- Request:** POST User Login (under Piazza Backend)
- URL:** http://136.112.162.81:5000/api/auth/login
- Body:** raw JSON ({"email": "olga@gmail.com", "password": "Olga12345678"})
- Response Status:** 200 OK
- Response Body:** (JSON) Success: true, message: "Login successful", data: { user: { id: '693565bfbd4ccc06ee4de0d09', username: 'olga', email: 'olga@gmail.com' }, token: 'eyJhbGciOiJIUzI1NiIsInR5cCIkVXVCJ9.eyJpZC16jYSMzU2NjnjYjRjY2MwNwVlNGRlMQwOsImlhCI6MTc2NTEmNzU3Os1ZXhwIjoxNzY1MTkzOTc5fQ. L8SwPhAU3yIfSgQ1HMFVfc5Kwv5Q010f_KibK_BuZ_uU' }

Nick:

The screenshot shows the Postman interface with the following details:

- Collection:** My Workspace
- Request:** POST User Login (under Piazza Backend)
- URL:** http://136.112.162.81:5000/api/auth/login
- Body:** raw JSON ({"email": "nick@gmail.com", "password": "Nick12345678"})
- Response Status:** 200 OK
- Response Body:** (JSON) Success: true, message: "Login successful", data: { user: { id: '693566abab4ccc06ee4de0d0c', username: 'nick', email: 'nick@gmail.com' }, token: 'eyJhbGciOiJIUzI1NiIsInR5cCIkVXVCJ9.eyJpZC16jYSMzU2NjBhYjRjY2MwNwVlNGRlMQwYisImlhCI6MTc2NTEmNzY0MSw1ZXhwIjoxNzY1MTk0MDQxfQ. EKMNaISPTkxwFxre-Ix3jZLx2mso5CPL_ThP3grk' }

3. TC3: docs/videos/TC_3.mp4

Olga without token:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A GET request is made to `http://136.112.162.81:5000/api/posts?`. In the Headers tab, the "Authorization" field is set to "none". The response status is 401 Unauthorized, with the message "Not authorized, token failed".

```

1 {
2   "success": false,
3   "message": "Not authorized, token failed"
4 }
    
```

Olga with token:

The screenshot shows the Postman interface with the same collection and request setup as the previous screenshot. However, the "Authorization" field now contains "Bearer Token" and has a dropdown menu showing "Token". The response status is 200 OK, indicating success.

```

1 {
2   "success": true,
3   "count": 23,
4   "data": [
5     "posts": [
6       {
7         "_id": "64028bd617e885cd2a81df51",
8         "title": "Hugging Face is the best company",
9         "topics": [
10           "Tech"
11         ],
12         "message": "Hugging Face release the most AI model in the World.",
13         "owner": null,
14         "expirationTime": "2025-10-29T22:49:10.107Z",
15         "status": "Live",
16         "likes": [],
17         "dislikes": [],
18         "comments": [
19           {
20             "user": null,
21           }
22         ]
23       }
24     ]
25   }
26 }
    
```

4. TC4: docs/videos/TC_4.mp4

post created by Olga:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to `http://136.112.162.81:5000/api/posts`. The request body contains the following JSON:

```
1 {
2   "title": "JavaScript Frameworks Discussion",
3   "message": "What do you think about the latest trends in JavaScript frameworks? I've been exploring React Server Components and they seem quite promising for reducing client-side JavaScript.",
4   "topics": ["Tech"],
5   "expirationMinutes": 5
6 }
```

The response status is `201 Created` with a response time of 738 ms and 1 KB. The response body is:

```
3   "message": "Post created successfully",
4   "data": {
5     "post": {
6       "title": "JavaScript Frameworks Discussion",
7       "topics": [
8         "Tech"
9       ],
10      "message": "What do you think about the latest trends in JavaScript frameworks? I've been exploring React Server Components and they seem quite promising for reducing client-side JavaScript.",
11      "owner": {
12        "_id": "693565bb4ccc06ee4de0d09",
13        "username": "olga",
14        "email": "olga@gmail.com"
15      },
16      "expirationTime": "2025-12-07T11:51:32.798Z",
17      "status": "Live",
18      "likes": [],
19      "dislikes": [],
20      "_id": "6935602b4ccc06ee4de0d35",
21      "comments": []
22    }
23 }
```

5. TC5: docs/videos/TC_5.mp4

Post created by nick:

The screenshot shows the Postman interface with a collection named "Plaza Backend". A POST request is made to `http://136.112.162.81:5000/api/posts`. The request body contains the following JSON:

```
1 {
2   "title": "Mobile Development Trends",
3   "message": "Flutter vs React Native - which one are you using for cross-platform development? I'm trying to decide for my next project and would appreciate real-world experiences.",
4   "topics": ["Tech"],
5   "expirationMinutes": 5
6 }
```

The response status is `201 Created` with a response time of 679 ms and 1007 B. The response body is:

```
3   "message": "Post created successfully",
4   "data": {
5     "post": {
6       "title": "Mobile Development Trends",
7       "topics": [
8         "Tech"
9       ],
10      "message": "Flutter vs React Native - which one are you using for cross-platform development? I'm trying to decide for my next project and would appreciate real-world experiences.",
11      "owner": {
12        "_id": "693560ab4ccc06ee4de0d0c",
13        "username": "nick",
14        "email": "nick@gmail.com"
15      },
16      "expirationTime": "2025-12-07T11:54:08.011Z",
17      "status": "Live",
18      "likes": [],
19      "dislikes": [],
20      "_id": "693560b4b4ccc06ee4de0d3f",
21      "comments": []
22    }
23 }
```

6. TC6: docs/videos/TC_6.mp4

Post created by Mary:

The screenshot shows the Postman interface with a collection named "Plazza Backend". A POST request is made to the endpoint `http://136.112.162.81:5000/api/posts`. The request body contains the following JSON:

```
1 {
2   "title": "API Design Principles",
3   "message": "Designing a RESTful API for our new microservice. Debating between REST and GraphQL. What are your thoughts on API versioning and backward compatibility strategies?",
4   "topics": ["Tech"],
5   "expirationMinutes": 10
6 }
```

The response status is `201 Created`, with a response time of 704 ms and a response size of 1000 B. The response body is:

```
3   "message": "Post created successfully",
4   "data": [
5     "post": {
6       "title": "API Design Principles",
7       "topics": [
8         "Tech"
9       ],
10      "message": "Designing a RESTful API for our new microservice. Debating between REST and GraphQL. What are your thoughts on API versioning and backward compatibility strategies?",
11      "owner": {
12        "_id": "69356630b4ccc00ee4de0d0f",
13        "username": "mary",
14        "email": "mary@gmail.com"
15      },
16      "expirationTime": "2025-12-07T12:02:51.198Z",
17      "status": "Live",
18      "likes": [],
19      "dislikes": [],
20      "_id": "69356a93b4ccc00ee4de0d44",
21      "comments": []
22    }
23  ]
```

7. TC7: docs/videos/TC_7.mp4

Nick searched for available posts.

The screenshot shows the Postman interface with the same "Plazza Backend" collection. A GET request is made to the endpoint `http://136.112.162.81:5000/api/posts?`. The request includes an "Authorization" header set to "Bearer Token" with a placeholder token value. The response status is `200 OK`, with a response time of 646 ms and a response size of 2.27 KB. The response body is:

```
1 {
2   "success": true,
3   "count": 3,
4   "data": [
5     "posts": [
6       {
7         "id": "693569ff1b4ccc00ee4de0d70",
8         "title": "DevOps Tools Comparison",
9         "topics": [
10           "Tech"
11         ],
12         "message": "Currently evaluating CI/CD tools for our team. We're considering GitHub Actions, Jenkins, and GitLab CI. What has been your experience with these platforms? Looking for pros and cons.",
13         "owner": {
14           "_id": "69356630b4ccc00ee4de0d0f",
15           "username": "mary",
16           "email": "mary@gmail.com"
17         },
18         "expirationTime": "2025-12-07T15:27:11.159Z",
19         "status": "Live",
20       }
21     ]
22   ]
23 }
```

Olga searched for available posts.

The screenshot shows the Postman interface with a collection named "Piazza Backend". A specific POST request for "/api/posts" is selected. The "Authorization" tab is active, showing "Bearer Token" as the type. The "Body" tab displays a JSON response with 3 posts, one of which is owned by "mary". The response body is as follows:

```
1 {
2   "success": true,
3   "count": 3,
4   "data": [
5     "posts": [
6       {
7         "_id": "693599ffb4ccc06ee4de0d70",
8         "title": "DevOps Tools Comparison",
9         "topics": [
10           "Tech"
11         ],
12         "message": "Currently evaluating CI/CD tools for our team. We're considering GitHub Actions, Jenkins, and GitLab CI. What has been your experience with these platforms? Looking for pros and cons.",
13         "owner": {
14           "_id": "69356630b4ccc06ee4de0d0f",
15           "username": "mary",
16           "email": "mary@gmail.com"
17         },
18         "expirationTime": "2025-12-07T15:27:11.159Z",
19         "status": "Live",
20       }
21     ]
22   }
23 }
```

8. TC8: docs/videos/TC_8.mp4

Mary Post: 693599ffb4ccc06ee4de0d70 (id)

Olga liked Mary's post.

The screenshot shows the Postman interface with the same collection and POST request. The "Authorization" tab is active, showing "Be..." as the type. The "Body" tab displays a JSON response with a message indicating the post was liked. The response body is as follows:

```
1 {
2   "success": true,
3   "message": "post liked",
4   "data": {
5     "post": {
6       "_id": "693599ffb4ccc06ee4de0d70",
7       "title": "DevOps Tools Comparison",
8       "topics": [
9         "Tech"
10       ],
11       "message": "Currently evaluating CI/CD tools for our team. We're considering GitHub Actions, Jenkins, and GitLab CI. What has been your experience with these platforms? Looking for pros and cons.",
12       "owner": {
13         "_id": "69356630b4ccc06ee4de0d0f",
14         "username": "mary",
15         "email": "mary@gmail.com"
16       },
17       "expirationTime": "2025-12-07T15:27:11.159Z",
18       "status": "Live",
19       "likes": [
20         {
21           "user": {
22             "_id": "693599ffb4ccc06ee4de0d70",
23             "username": "olga",
24             "email": "olga@gmail.com"
25           }
26         }
27       ]
28     }
29   }
30 }
```

Nick liked Mary's post.

The screenshot shows the Postman interface with the following details:

- Header Bar:** Home, Workspaces, API Network, Search Postman, Ctrl + K.
- Left Sidebar:** My Workspace, New, Import, Collections (Plaza Backend), Environments, History, Flows.
- Request URL:** http://136.112.162.81:5000/api/posts/693599fffb4ccc06ee4de0d70/like
- Method:** POST
- Authorization:** Bearer Token (highlighted with a yellow box)
- Body:** JSON response (200 OK) containing a liked post object with fields like _id, title, topics, message, owner, expirationTime, status, and likes.

9. TC9: docs/videos/TC_9.mp4

Nick's post ID: 693599a8b4ccc06ee4de0d66

The screenshot shows the Postman application interface. The left sidebar has sections for 'Collections' (with 'Plazza Backend' expanded), 'Environments' (with one environment named 'Production'), 'History' (empty), 'Flows' (empty), and 'Files' (empty). The main workspace title is 'My Workspace'. A top navigation bar includes 'Home', 'Workspaces', 'API Network', a search bar ('Search Postman'), and buttons for 'Invite', 'Upgrade', and environment selection ('No environment'). The central area shows a 'Plazza Backend / Post Like' collection. A specific POST request is selected, with the URL being `http://136.112.162.81:5000/api/posts/693599a8b4ccc06ee4de0d66/like`. The 'Authorization' tab is active, showing 'Auth Type' as 'Be...' and 'Token' as a masked value. Below the request details, the response section shows a 200 OK status with a response time of 1.23 s, a size of 1.04 KB, and a preview of the JSON response body:

```
1 {  
2   "success": true,  
3   "message": "Post liked",  
4   "data": {  
5     "post": {  
6       "_id": "693599a8b4ccc06ee4de0d66",  
7       "title": "API Design Principles",  
8       "topics": [  
9         "Tech"  
10      ],  
11      "message": "Designing a RESTful API for our new microservice. Debating between REST and GraphQL. What are your thoughts on API versioning and backward compatibility strategies?",  
12      "owner": {  
13        "_id": "6935660ab4ccc06ee4de0d0c",  
14        "username": "nick",  
15        "email": "nick@gmail.com"  
16      },  
17      "expirationTime": "2025-12-07T15:43:44.636Z",  
18      "status": "Live",  
19      "likes": [  
20    ]  
21  }  
22}  
23
```

Mary's post ID: 693599ffb4ccc06ee4de0d70

The screenshot shows the Postman application interface. On the left, the sidebar includes sections for 'My Workspace' (Collections, Environments, History, Flows), 'New Import', and a search bar. The main workspace displays a collection named 'Piazza Backend'. A specific POST request is selected, which is intended to dislike a post. The request URL is `http://136.112.162.81:5000/api/posts/693599ffbf4ccc06ee4de0d70/dislike`. The 'Authorization' tab is active, showing a 'Token' field with a placeholder '.....'. The 'Body' tab shows a JSON response with the following content:

```
1 {  
2     "success": true,  
3     "message": "Post disliked",  
4     "data": {  
5         "post": {  
6             "_id": "693599ffbf4ccc06ee4de0d70",  
7             "title": "DevOps Tools Comparison",  
8             "topics": [  
9                 "Tech"  
10            ],  
11            "message": "Currently evaluating CI/CD tools for our team. We're considering GitHub Actions, Jenkins, and GitLab CI. What has been your experience  
with these platforms? Looking for pros and cons.",  
12            "owner": {  
13                "_id": "69356630b4ccc06ee4de0d0f",  
14                "username": "mazy",  
15                "email": "mazy@gmail.com"  
16            },  
17            "expirationTime": "2025-12-07T15:50:11.159Z",  
18            "status": "Live",  
19            "likes": [  
20        ]  
21        }  
22    }  
23 }
```

10. TC10: docs/videos/TC_10.mp4

Nick looks at Mary's posts (2 likes and 1 dislike).

The screenshot shows the Postman application interface. The left sidebar has sections for Home, Workspaces, and API Network, with 'My Workspace' selected. Under 'Collections', there is a 'Plazza Backend' collection containing various API endpoints like User Registration, User Login, Protected Route, Posts, and Post Details. The 'Posts' section is expanded, and the 'GET Posts' endpoint is selected. The request URL is set to `http://136.112.162.81:5000/api/posts?`. The 'Authorization' tab shows 'Bearer Token' selected as the auth type, with a placeholder box containing a redacted token. The 'Body' tab shows a JSON response with one post object. The response body is as follows:

```
{
  "data": {
    "posts": [
      {
        "_id": "6493566fbabccc06ee4de0d70",
        "topic": "DevOps Tools Comparison",
        "tags": ["Tech"]
      }
    ],
    "message": "Currently evaluating CI/CD tools for our team. We're considering GitHub Actions, Jenkins, and GitLab CI. What has been your experience with these platforms? Looking for pros and cons.",
    "owner": {
      "id": "6493566fbabccc06ee4de0d71",
      "name": "Alice",
      "email": "alice@gmail.com"
    },
    "createdAt": "2025-12-07T15:58:11.159Z",
    "status": "Live",
    "likes": [
      {
        "_id": "6493566fbabccc06ee4de0d99",
        "username": "olga"
      },
      {
        "_id": "6493566fbabccc06ee4de0d9c",
        "username": "nick"
      }
    ],
    "dislikes": [
      {
        "_id": "6493566fbabccc06ee4de0d12",
        "username": "nestor"
      }
    ]
  }
}
```

The status bar at the bottom indicates a 200 OK response with 693 ms and 2.48 KB. There are also buttons for Save Response, Copy, Find, and Refresh.

Nick looks at his own post (one like).

The screenshot shows a Postman collection named "Piazza Backend" with a "Posts" endpoint. The "Authorization" tab is selected, showing "Bearer Token". The "Headers" tab includes "Content-Type: application/json" and "Accept: application/json". The "Body" tab shows a JSON payload:

```
[{"id": "693599fb4cc06ee4de0d070", "title": "API Design Principles", "body": "Designing a RESTful API for our new microservice. Debating between REST and GraphQL. What are your thoughts on API versioning and backward compatibility strategies?", "author": "Nick", "url": "https://www.nick.com", "createdAt": "2025-12-07T15:43:44.636Z", "updatedAt": "2025-12-07T15:43:44.636Z", "likes": 1, "comments": 0, "commentCount": 0, "totalLikesCount": 1}, {"id": "693599fb4cc06ee4de0d071", "title": "Tech Stack", "body": "Exploring the latest technologies for our new project. Considering Node.js, Python, and Java. Any recommendations?", "author": "Sarah", "url": "https://www.sarah.com", "createdAt": "2025-12-07T15:43:44.641Z", "updatedAt": "2025-12-07T15:43:44.641Z", "likes": 1, "comments": 0, "commentCount": 0, "totalLikesCount": 1}, {"id": "693599fb4cc06ee4de0d072", "title": "Data Science", "body": "Discussing the future of data science. What are the most promising research areas right now?", "author": "David", "url": "https://www.david.com", "createdAt": "2025-12-07T15:43:44.646Z", "updatedAt": "2025-12-07T15:43:44.646Z", "likes": 1, "comments": 0, "commentCount": 0, "totalLikesCount": 1}], "message": "Designing a RESTful API for our new microservice. Debating between REST and GraphQL. What are your thoughts on API versioning and backward compatibility strategies?"}
```

The response status is 200 OK with a 693 ms duration and 2.48 KB size.

11. TC11: docs/videos/TC_11.mp4

Mary likes her own post

The screenshot shows a Postman collection named "Piazza Backend" with a "Post Like" endpoint. The "Authorization" tab is selected, showing "Bearer Token". The "Headers" tab includes "Content-Type: application/json" and "Accept: application/json". The "Body" tab shows a JSON payload:

```
{"success": false, "message": "Cannot like your own post"}
```

The response status is 403 Forbidden with a 495 ms duration and 329 B size.

12. TC12: docs/videos/TC_12.mp4

Nick comments on Mary's post:

The screenshot shows the Postman interface with a collection named "Piazza Backend". A POST request is selected to "Post comment" at the URL <http://136.112.162.81:5000/api/posts/693599fb4ccc06ee4de0d70/comment>. The request body is JSON with the following content:

```
1 {
2   "text": "Nice. This was a really good topic."
3 }
```

The response status is 201 Created, and the response body contains detailed information about the comment, including its ID, author (Mary), creation date, and a message from Mary.

Olga comments on Mary's post

The screenshot shows the Postman interface with the same collection and request setup as the previous screenshot. The request body is identical. However, the response body now includes a nested "comments" array under the comment object, indicating that Mary's comment has received two replies from Nick and Olga.

13. TC13: docs/videos/TC_13.mp4

This can be seen in a better way based on the video.

14. TC14: docs/videos/TC_14.mp4

Nestor creates a post.

The screenshot shows the Postman interface with a collection named "Piazza Backend". A POST request is selected with the URL `http://136.112.162.81:5000/api/posts`. The request body contains the following JSON:

```
1 {
2   "title": "Cancer is the worst illness",
3   "message": "Cancer is the worst illness in the human being. It destroys a whole family and i never imaginiate how can be helpful",
4   "topics": ["Health"],
5   "expirationMinutes": 12
6 }
```

The response status is **201 Created** with a duration of 752 ms and a size of 962 B. The response body is:

```
1 {
2   "success": true,
3   "message": "Post created successfully",
4   "data": [
5     "post": {
6       "title": "Cancer is the worst illness",
7       "topics": [
8         "Health"
9       ],
10      "message": "Cancer is the worst illness in the human being. It destroys a whole family and i never imaginiate how can be helpful",
11      "owner": {
12        "_id": "69356650b4ccc06ee4de0d12",
13        "username": "nestor",
14        "email": "nestor@gmail.com"
15      },
16      "expirationTime": "2025-12-07T16:23:00.685Z",
17      "status": "Live",
18      "likes": [],
19      "dislikes": [],
20      "_id": "6935a714bdccc06ee4de0de5",
21      "comments": [],
22      "createdAt": "2025-12-07T16:11:06.690Z",
23      "updatedAt": "2025-12-07T16:11:06.690Z"
24    }
25  ]
26}
```

15. TC15: docs/videos/TC_15.mp4

Mary searches for health-related posts.

The screenshot shows the Postman interface with a collection named "Piazza Backend". A GET request is selected with the URL `http://136.112.162.81:5000/api/posts?topic=Health`. The request includes an **Authorization** header set to **Bearer Token**. The response status is **200 OK** with a duration of 646 ms and a size of 932 B. The response body is:

```
1 {
2   "success": true,
3   "count": 1,
4   "data": [
5     "posts": [
6       {
7         "_id": "6935a714bdccc06ee4de0de5",
8         "title": "Cancer is the worst illness",
9         "topics": [
10           "Health"
11         ],
12         "message": "Cancer is the worst illness in the human being. It destroys a whole family and i never imaginiate how can be helpful",
13         "owner": {
14           "_id": "69356650b4ccc06ee4de0d12",
15           "username": "nestor",
16           "email": "nestor@gmail.com"
17         },
18         "expirationTime": "2025-12-07T16:23:00.685Z",
19         "status": "Live",
20         "likes": [],
21         "dislikes": [],
22         "comments": [],
23         "createdAt": "2025-12-07T16:11:06.690Z",
24         "updatedAt": "2025-12-07T16:11:06.690Z"
25       }
26     ]
27   }
28 }
```

16. TC16: docs/videos/TC_16.mp4

Mary comments on Nestor's post.

The screenshot shows the Postman interface with the following details:

- Collection:** My Workspace / Piazza Backend
- Request Type:** POST
- URL:** http://136.112.162.81:5000/api/posts/6935a714b4ccc06ee4de0e5/comment
- Body:** JSON (raw)

```
1 {
2   "text": "Nice. This was a really good topic."
3 }
```
- Response Status:** 201 Created
- Response Body:**

```
1 {
2   "success": true,
3   "message": "Comment added successfully",
4   "data": {
5     "post": {
6       "_id": "6935a714b4ccc06ee4de0e5",
7       "title": "Cancer is the worst illness",
8       "topics": [
9         "Health"
10       ],
11       "message": "Cancer is the worst illness in the human being. It destroys a whole family and i never imaginete how can be helpful",
12       "owner": {
13         "_id": "6935a714b4ccc06ee4de0e5",
14         "username": "nestor",
15         "email": "nestor@gmail.com"
16       },
17       "expirationTime": "2025-12-07T16:23:00.685Z",
18       "status": "Live",
19       "likes": [],
20       "dislikes": [],
21       "comments": []
22     }
23   }
24 }
```

17. TC17: docs/videos/TC_17.mp4

Mary dislikes the nestor's message.

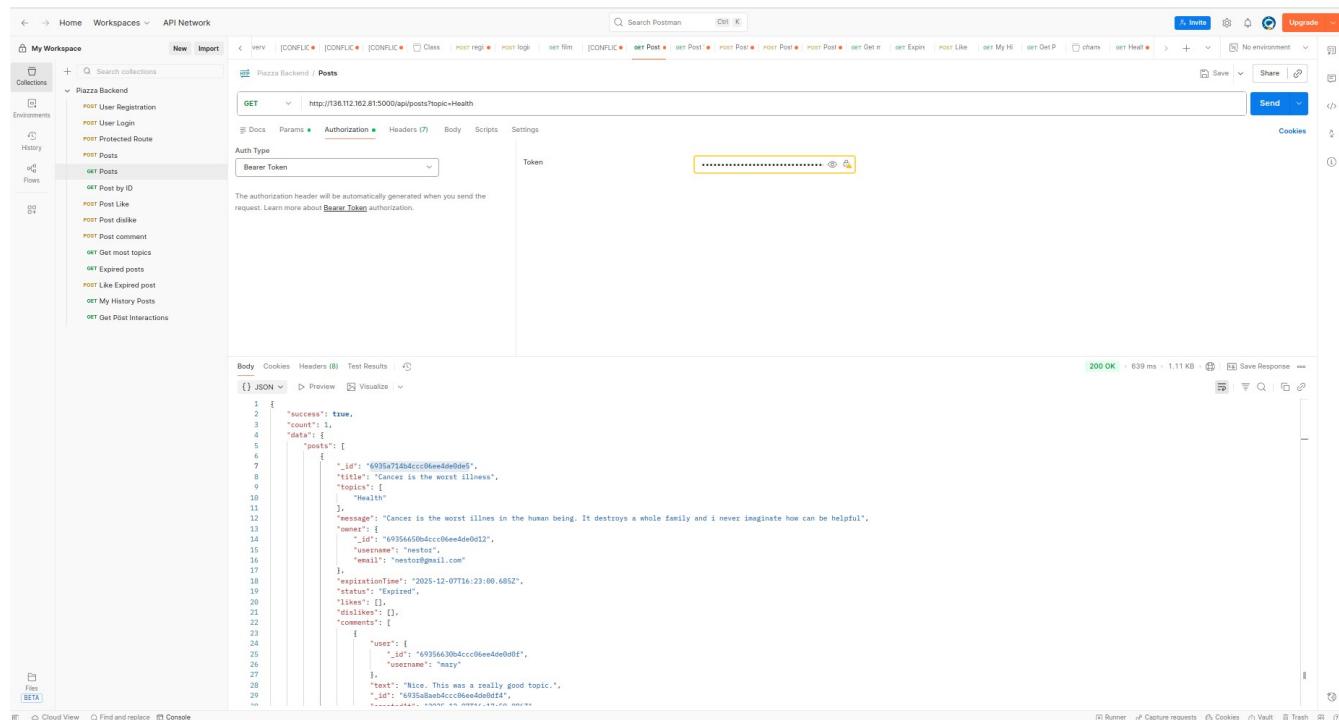
The screenshot shows the Postman interface with the following details:

- Collection:** My Workspace / Piazza Backend
- Request Type:** POST
- URL:** http://136.112.162.81:5000/api/posts/6935a714b4ccc06ee4de0e5/like
- Headers:** Authorization (Bearer Token)
- Auth Type:** Bearer
- Response Status:** 400 Bad Request
- Response Body:**

```
1 {
2   "success": false,
3   "message": "Cannot like an expired post"
4 }
```

18. TC18: docs/videos/TC_18.mp4

Nestor browses all the messages on the Health topic.



My Workspace

Plaza Backend / Posts

GET http://136.112.162.81:5000/api/posts?topic=Health

Auth Type: Bearer Token

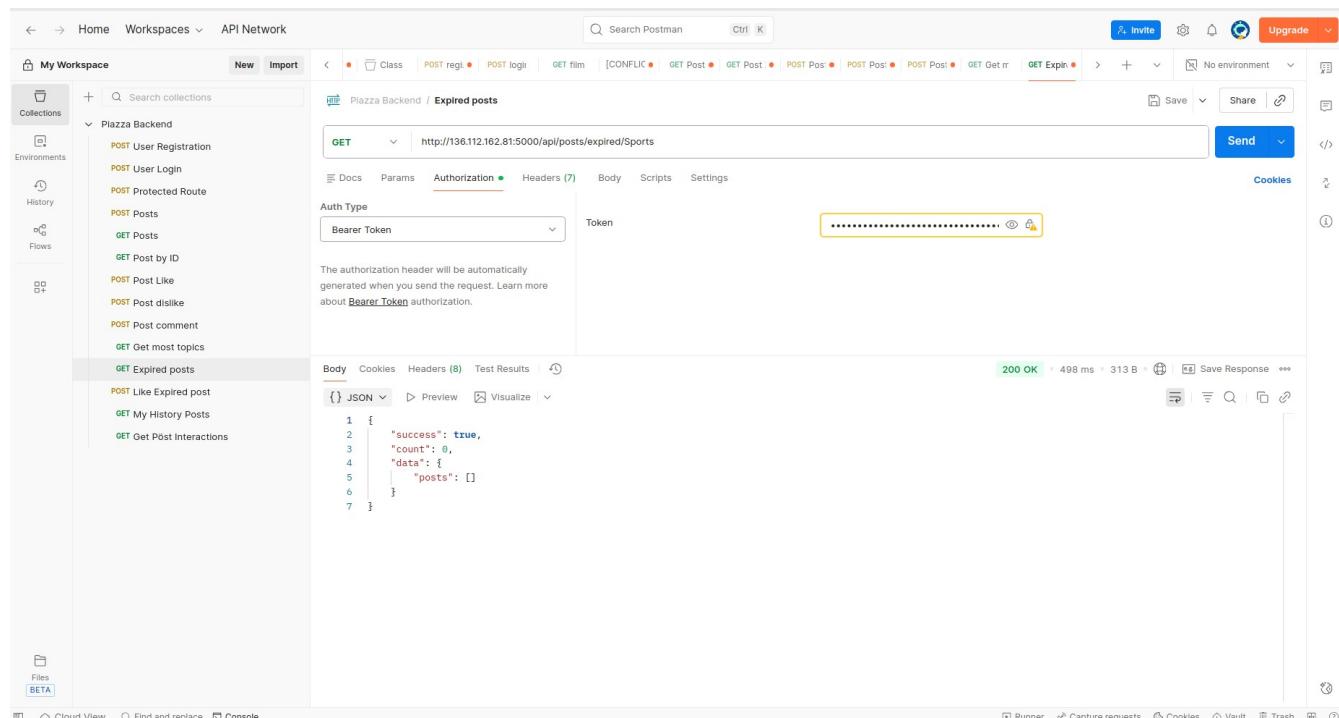
Body:

```
{ "success": true, "count": 1, "data": [ { "posts": [ { "id": "6935a71d4cc06eedde0d0d5", "title": "Cancer is the worst illness", "topics": [ "Health" ], "message": "Cancer is the worst illness in the human being. It destroys a whole family and i never imagine how can be helpful.", "owner": { "id": "6935a6500dc06eedde0d0d12", "username": "nestor", "email": "nestor@gmail.com" }, "expirationTime": "2025-12-07T16:23:00.685Z", "status": "expired", "likes": 0, "dislikes": 0, "comments": [ { "user": { "id": "6935a66304cc06eedde0d0f1", "username": "mary" }, "text": "Nirv. This was a really good topic.", "id": "6935a8ab4ccc06eedde0d0fd1" } ] } } ] }
```

200 OK

19. TC19: docs/videos/TC_19.mp4

Nick browses all the expired messages on the Sports topic.



My Workspace

Plaza Backend / Expired posts

GET http://136.112.162.81:5000/api/posts/expired/Sports

Auth Type: Bearer Token

Body:

```
{ "success": true, "count": 0, "data": { "posts": [] } }
```

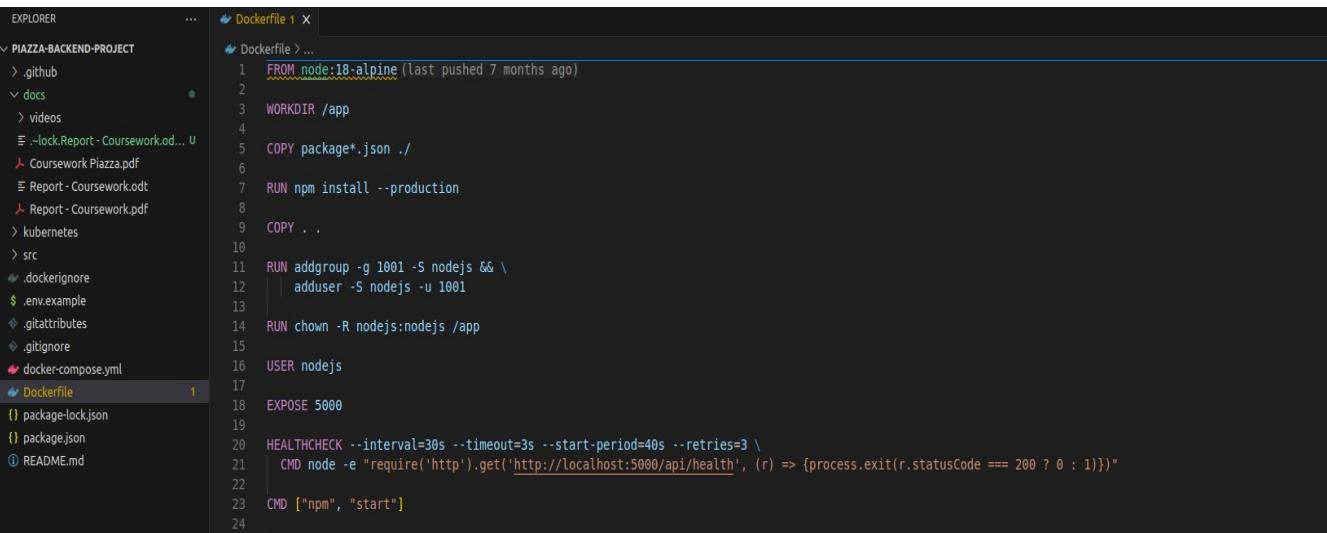
200 OK

20. TC20: docs/videos/TC_20.mp4

The best way to see this test case is by watching the video.

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:



```
FROM node:18-alpine
WORKDIR /app
COPY package*.json .
RUN npm install --production
COPY . .
RUN addgroup -g 1001 -S nodejs && \
    | adduser -S nodejs -u 1001
RUN chown -R nodejs:nodejs /app
USER nodejs
EXPOSE 5000
HEALTHCHECK --interval=30s --timeout=3s --start-period=40s --retries=3 \
  CMD node -e "require('http').get('http://localhost:5000/api/health', (r) => {process.exit(r.statusCode === 200 ? 0 : 1)})"
CMD ["npm", "start"]
```

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.

The screenshot shows the Google Cloud Kubernetes Engine Clusters page. The left sidebar has a 'Clústeres' section selected. The main area displays a single cluster named 'piazza-cluster' with the following details:

Estado	Actualización	Costo mensual estimado
100% en buen estado	100% actualizados	\$0.00/mes · 0%

Below this, there is a table showing the cluster configuration:

Estado	Nombre	Ubicación	Cantidad de nodos	CPU virtuales totales	Memoria total	Notificaciones	Etiquetas
OK	piazza-cluster	southamerica-west1-b	5	10	20 GB	-	:

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

The screenshot shows the Google Cloud Workload Balancer page. The left sidebar has a 'Cargas de trabajo' section selected. The main area displays a single load balancer named 'piazza-api-deployment' with the following details:

Nombre	Estado	Tipo	Pods	Tipo de nodo	Espacio de nombres	Clúster
piazza-api-deployment	OK	Deployment	2/2	Administrado por el usuario	default	piazza-cluster

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

The screenshot shows the Google Cloud Compute Engine interface for managing VM instances. The left sidebar has sections for Compute Engine, Descripción general, Descripción general de..., Máquinas virtuales (selected), Plantillas de instancia, Nodos de usuario único, Imágenes de máquina, TPU, Descuentos por compr..., Reservas, Migrate to Virtual Mach..., Almacenamiento, and Discos. The main area is titled 'Instancias de VM' with tabs for 'Instancias' (selected), 'Observabilidad', and 'Programas de las Instancias'. A search bar at the top right contains 'vm'. Below is a table titled 'Instancias de VM' with columns: Estado, Nombre ↑, Zona, Recomendaciones, En uso por, IP interna, IP externa, and Conectar. There are 8 rows of data, each with a green checkmark icon and a link to 'gke-piazza-cluster-default-pool-ea174bd8-...'. The last row is for 'instance-20251113-201926' in 'us-central1-a'.

Estado	Nombre ↑	Zona	Recomendaciones	En uso por	IP interna	IP externa	Conectar
<input type="checkbox"/>	gke-piazza-cluster-default-pool-ea174bd8-202t	southamerica-west1-b		gke-piazza-cluster-default-pool-ea174bd8-40zm	10.194.0.7 (nic0)	34.176.127.193 (nic0)	SSH
<input type="checkbox"/>	gke-piazza-cluster-default-pool-ea174bd8-40zm	southamerica-west1-b		gke-piazza-cluster-default-pool-ea174bd8-c75m	10.194.0.5 (nic0)	34.176.186.182 (nic0)	SSH
<input type="checkbox"/>	gke-piazza-cluster-default-pool-ea174bd8-c75m	southamerica-west1-b		gke-piazza-cluster-default-pool-ea174bd8-jvnp	10.194.0.6 (nic0)	34.176.34.85 (nic0)	SSH
<input type="checkbox"/>	gke-piazza-cluster-default-pool-ea174bd8-jvnp	southamerica-west1-b		gke-piazza-cluster-default-pool-ea174bd8-p31h	10.194.0.3 (nic0)	34.176.159.141 (nic0)	SSH
<input type="checkbox"/>	gke-piazza-cluster-default-pool-ea174bd8-p31h	southamerica-west1-b			10.194.0.4 (nic0)	34.176.172.141 (nic0)	SSH
<input type="checkbox"/>	instance-20251113-201926	us-central1-a			10.128.0.8 (nic0)	34.61.223.161 (nic0)	SSH

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
```

The screenshot shows a terminal window with the following content:

```

Gemini CLI está disponible en la terminal de Cloud Shell. Pruébala escribiendo gemini. Más información
No volver a mostrar este mensaje Ignorar

francisco_macaya22@cloudshell:~/chameleon-backend (chameleon-476116)$ kubectl get service piazza-api-service
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
piazza-api-service   LoadBalancer  34.118.237.220  34.176.147.106  80:31189/TCP  26m
francisco_macaya22@cloudshell:~/chameleon-backend (chameleon-476116)$

```

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 work 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 work 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:

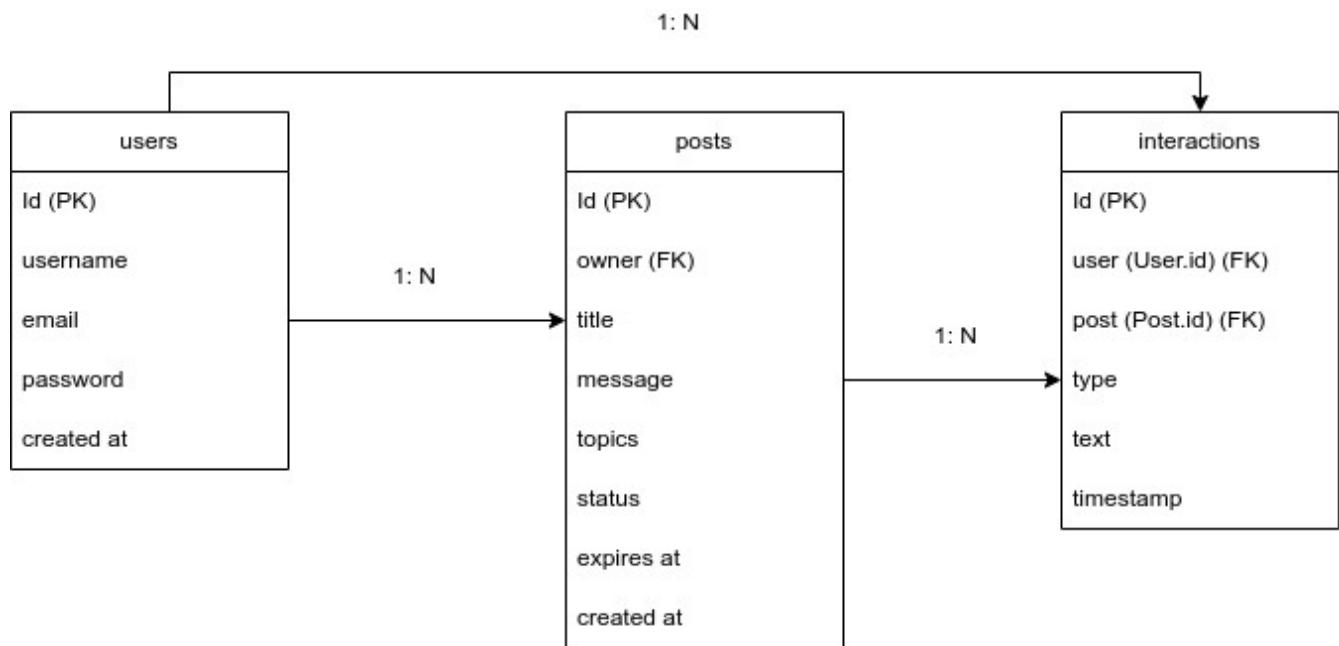
- Create a `.env` file with the required configuration variables.
- Run `docker-compose up -d`.
- Check the running services using `docker-compose ps`.
- If you want to stop the services, run the following command: `docker-compose down`.

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:



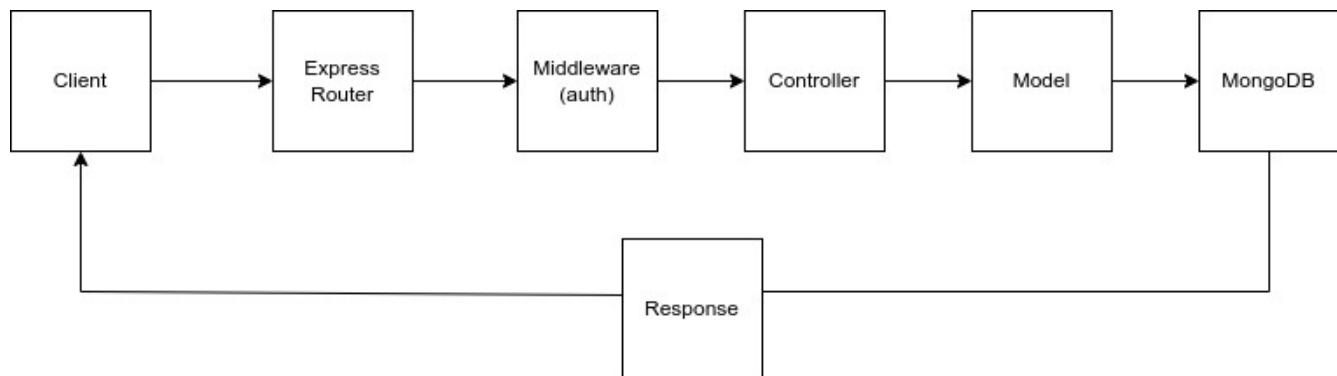
The database contains a **users** collection that stores user information such as username, email, and password (secured with a JWT token).

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. Arquitecture Diagram

The architecture diagram is shown below.



This diagram contains the following components:

- Client: Sends requests such as user login, registering, retrieving posts, etc.
- Express Router: A server-side layer that enables the separation of API routes into two main sections: AuthRouter and PostRouter.

AuthRouter: contains APIs related to user management (login, registration, etc.).

PostRouter : contains APIs related to posts.

- Middleware: A layer that validates incoming requests based on predefined rules, ensuring input correctness before processing.
- Controller: The layer responsible for executing the main business logic of the API and handling requests.
- MongoDB: Used to store and retrieve data related to users and posts.
- Response: The response returned to the client based on the processed request.

5. 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/>