Name: Viplavi Wade
Student Id: 13922741
Subject: Cloud Computing
Course: MSC Advanced Computing
Date: December 10, 2023

Cloud Computing Coursework

GitHub Repo Link: https://github.com/ViplaviWade/Piazza.git

1.0 Introduction:

Piazza is a dynamic social media platform designed to provide users with a personalized space for sharing thoughts, ideas, and engaging with a community. Leveraging modern web technologies, Piazza offers features such as user registration, authentication, and a robust set of APIs for creating, updating, and interacting with posts.

2.0 Architecture Overview: The architecture of Piazza follows a server-client model:

Server: Implemented using Node.js and Express, the server handles HTTP requests and interacts with the MongoDB database.

Database: MongoDB is employed as the primary database for its flexibility and scalability. It stores user information in the 'users' collection and post-related data in the 'posts' collection.

Authentication: OAuth v2 is integrated to secure user authentication, ensuring a seamless and secure login process.

3.0 Technology Stack:

Server-Side: Node.js with Express for building a fast and efficient server.

Database**:** MongoDB for its NoSQL capabilities, facilitating flexibility in data storage.

Authentication**:** OAuth v2 ensures secure user authentication with support for third-party login providers.

4.0 Database Schema

4.1 Users Collection

The 'users' collection contains:

1. **username**: User's unique identifier.
2. **email**: User's email for communication and identification.
3. **password**: Encrypted password for secure authentication.
4. **date**: Registration date for tracking user activity.

4.2 Posts Collection

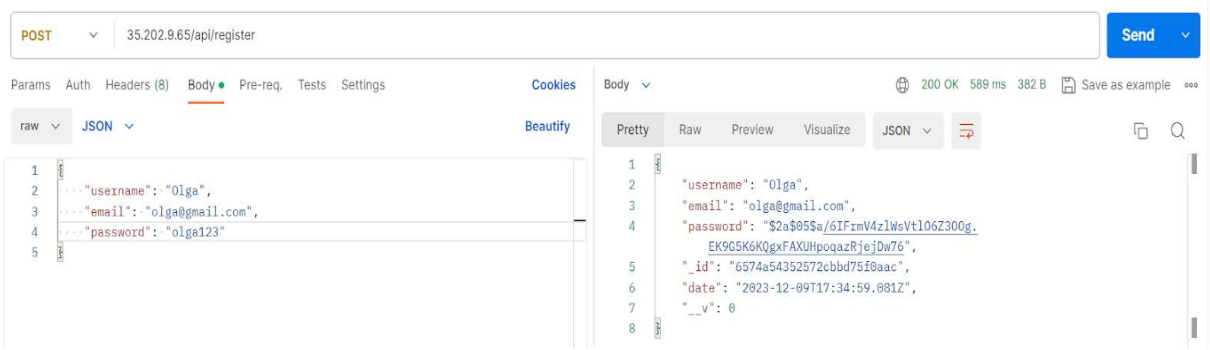
The 'posts' collection includes:

1. **post_title**: Title of the post.
2. **post_topic**: Topic or category to which the post belongs.
3. **message**: Content of the post.
4. **expiration_time**: Time after which the post is marked as 'EXPIRED'.
5. **status**: Current status of the post ('LIVE' or 'EXPIRED').
6. **like_users**, **dislike_users**: Arrays storing users who liked or disliked the post.
7. **likes_count**, **dislikes_count**: Counts of likes and dislikes.
8. **comments**: Array storing comments on the post.
9. **comments_count**: Count of comments on the post.
10. **post_id**: Unique identifier for the post.
11. **timestamp**: Timestamp indicating when the post was created.
12. **postOwner**: Username of the post creator.

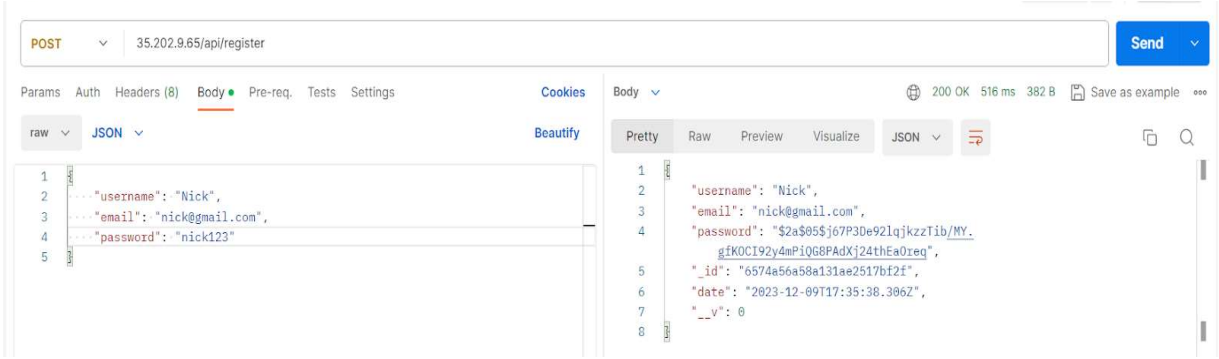
5.0 API Endpoints using test cases: (used Kubernetes deployment Endpoint IP address)

5.1 Test Case-1: Olga, Nick, Mary, and Nestor register and are ready to access the Piazza API.

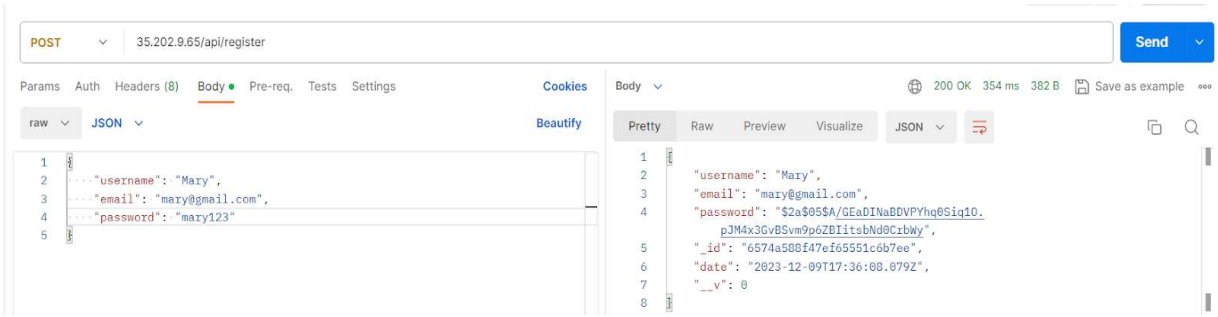
Olga registers to Piazza API



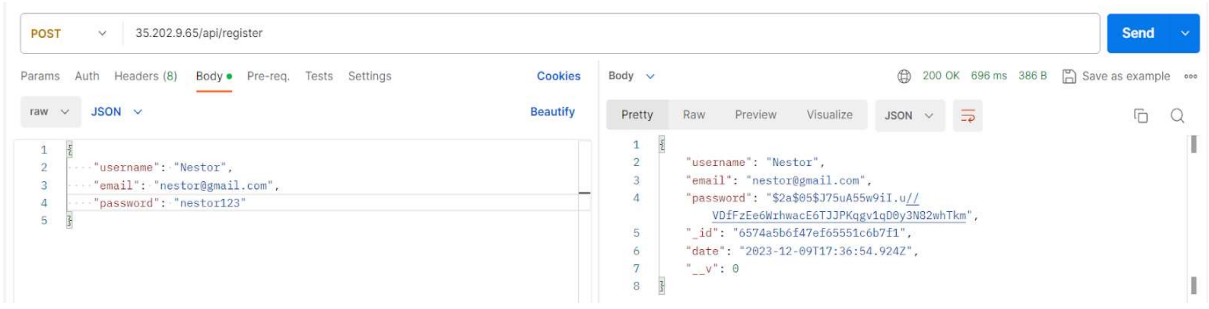
Nick registers to Piazza API



Mary registers to Piazza API



Nestor registers to Piazza API



All the users: Olga, Nick, Mary and Nestor are registered in the **Piazza API.**

Piazza.users

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 887B TOTAL DOCUMENTS: 6 INDEXES TOTAL SIZE: 36KB

Find Indexes Schema Anti-Patterns Aggregation Search Indexes

Filter Type a query: { field: 'value' }


```
_id: ObjectId('6574a54352572cbbd75f0aac')
username: "Olga"
email: "olga@gmail.com"
password: "$2a$05$a/6IFrmV4zlWsVtlO6Z3OOg.EK9G5K6KQgxFAXUHpoqazRjejDw76"
date: 2023-12-09T17:34:59.081+00:00
__v: 0
```

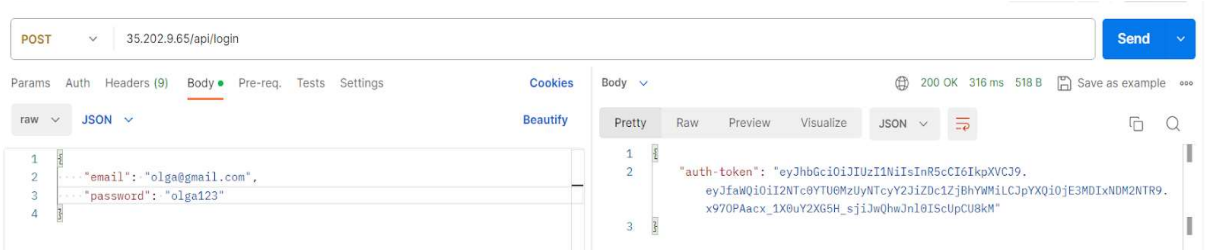
```
_id: ObjectId('6574a56a58a131ae2517bf2f')
username: "Nick"
email: "nick@gmail.com"
password: "$2a$05$j67P3De92lqjkzzTib/MY.gfKOCI92y4mPiQG8PAdXj24thEaOreq"
date: 2023-12-09T17:35:38.306+00:00
__v: 0
```

```
_id: ObjectId('6574a588f47ef65551c6b7ee')
username: "Mary"
email: "mary@gmail.com"
password: "$2a$05$A/GEaDINaBDVPYhq0Siq1O.pJM4x3GvBSvm9p6ZBIitsbNd0CrbWy"
date: 2023-12-09T17:36:08.079+00:00
__v: 0
```

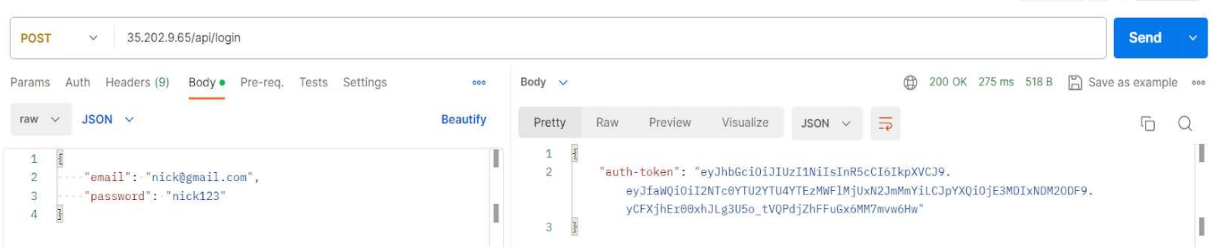
```
_id: ObjectId('6574a5b6f47ef65551c6b7f1')
username: "Nestor"
email: "nestor@gmail.com"
password: "$2a$05$J75uA55w9iI.u//VDfFzEe6WrhwacE6TJJPKqgv1qD0y3N82whTkm"
date: 2023-12-09T17:36:54.924+00:00
__v: 0
```

5.2 Test Case 2: Olga, Nick, Mary, and Nestor use the oAuth v2 authorisation service to register and get their tokens.

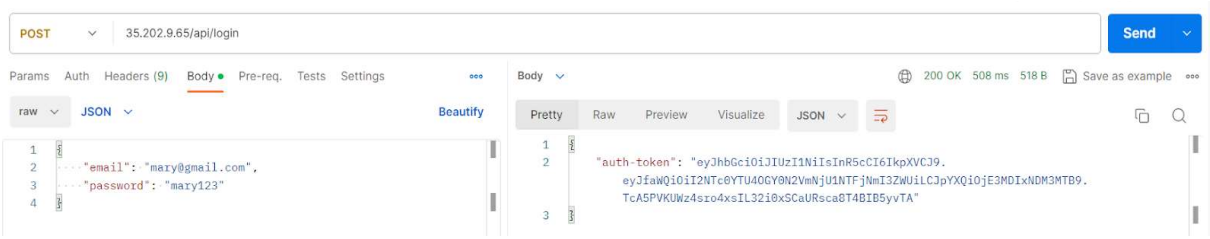
Olga receives a token for her login



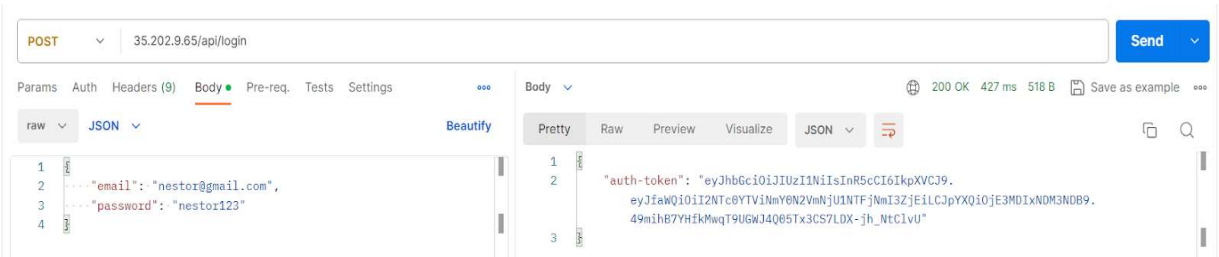
Nick receives a token for his login



Mary receives a token for her login

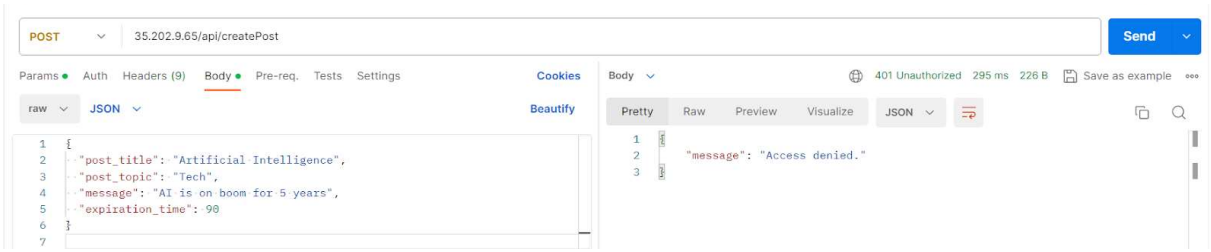


Nestor receives a token for her login



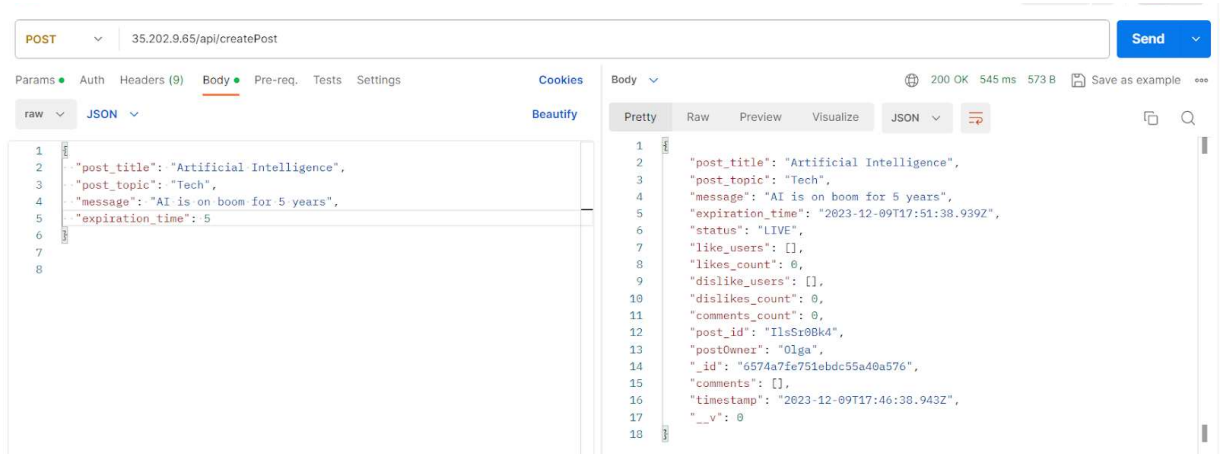
All the users used the **oAuth v2 authorization** service to register and receive their tokens.

5.3 Test Case 3: Olga makes a call to the API without using her token. This call should be unsuccessful as the user is unauthorized.

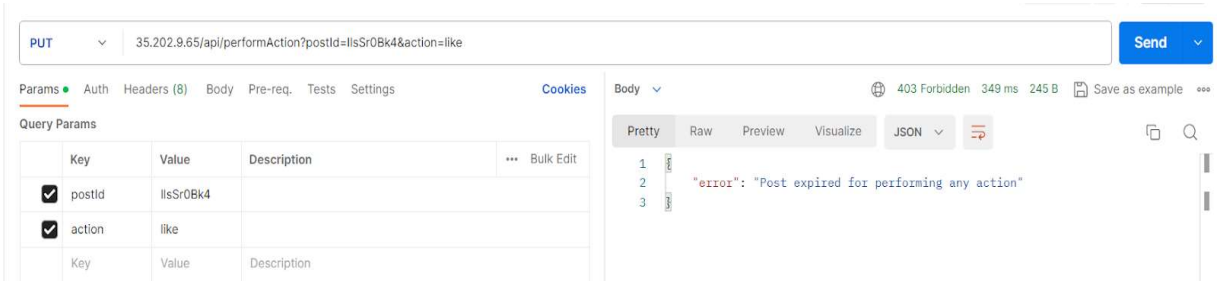


When Olga tried to make a call without her token, the API call failed as the user was not authorized. And the API returned the error message as "**Access Denied**".

5.4 Test Case 4: Olga posts a message in the Tech topic with an expiration time (e.g. 5 minutes) using her token. After the end of the expiration time, the message will not accept any further user interactions (likes, dislikes, or comments).

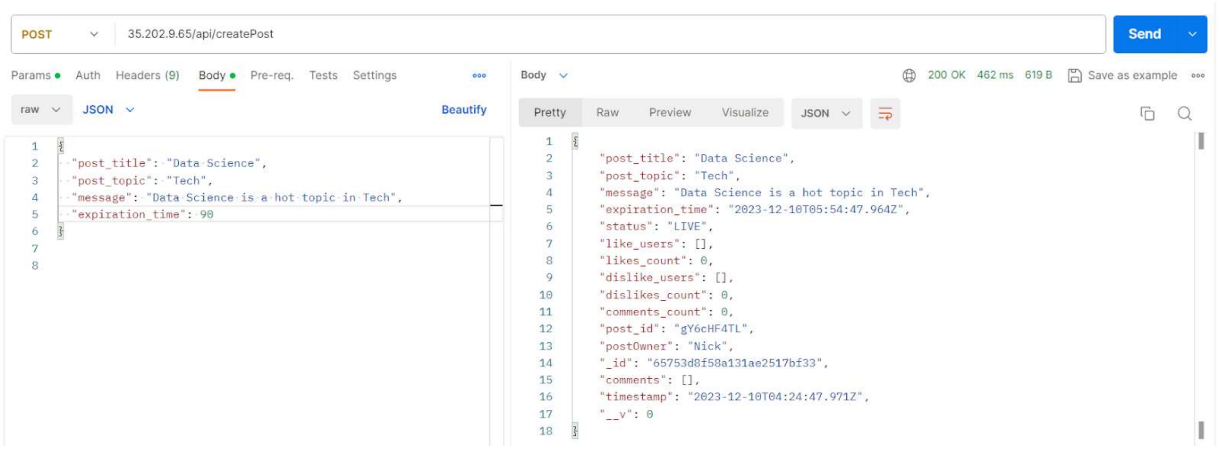


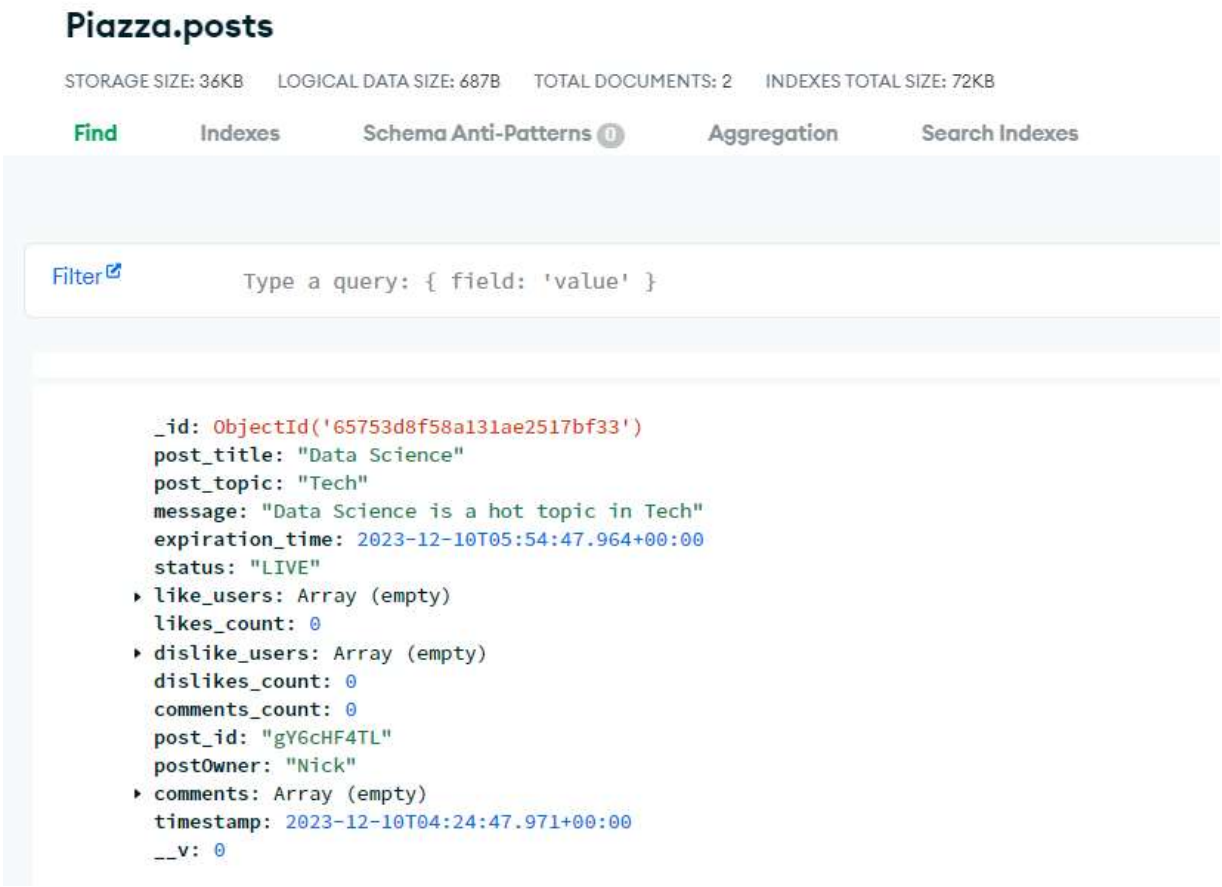
After 5 minutes (expiration time) the post will not accept any further interactions (like/dislike/comments)



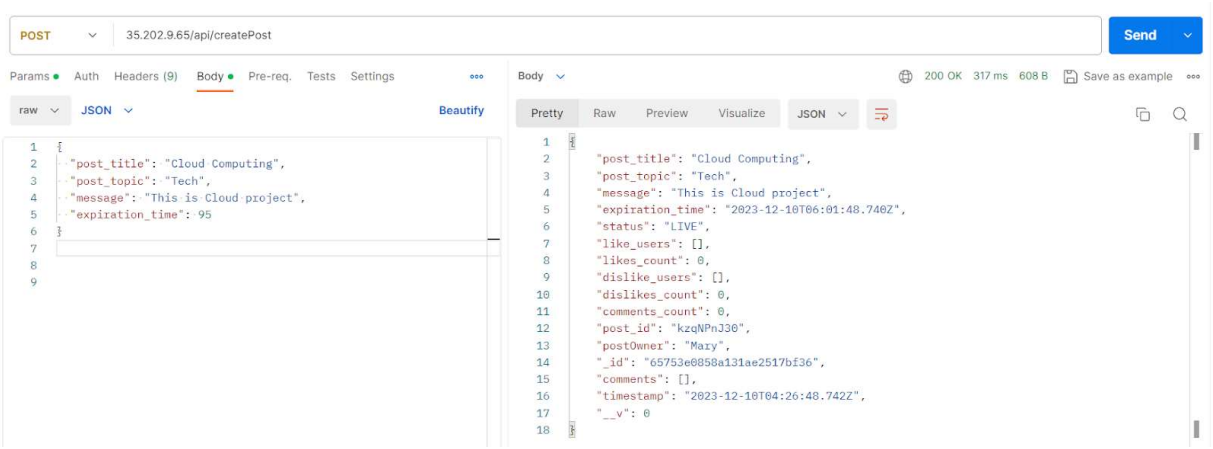
5.5 Test Case 5: Nick posts a message in the Tech topic with an expiration time using his token.

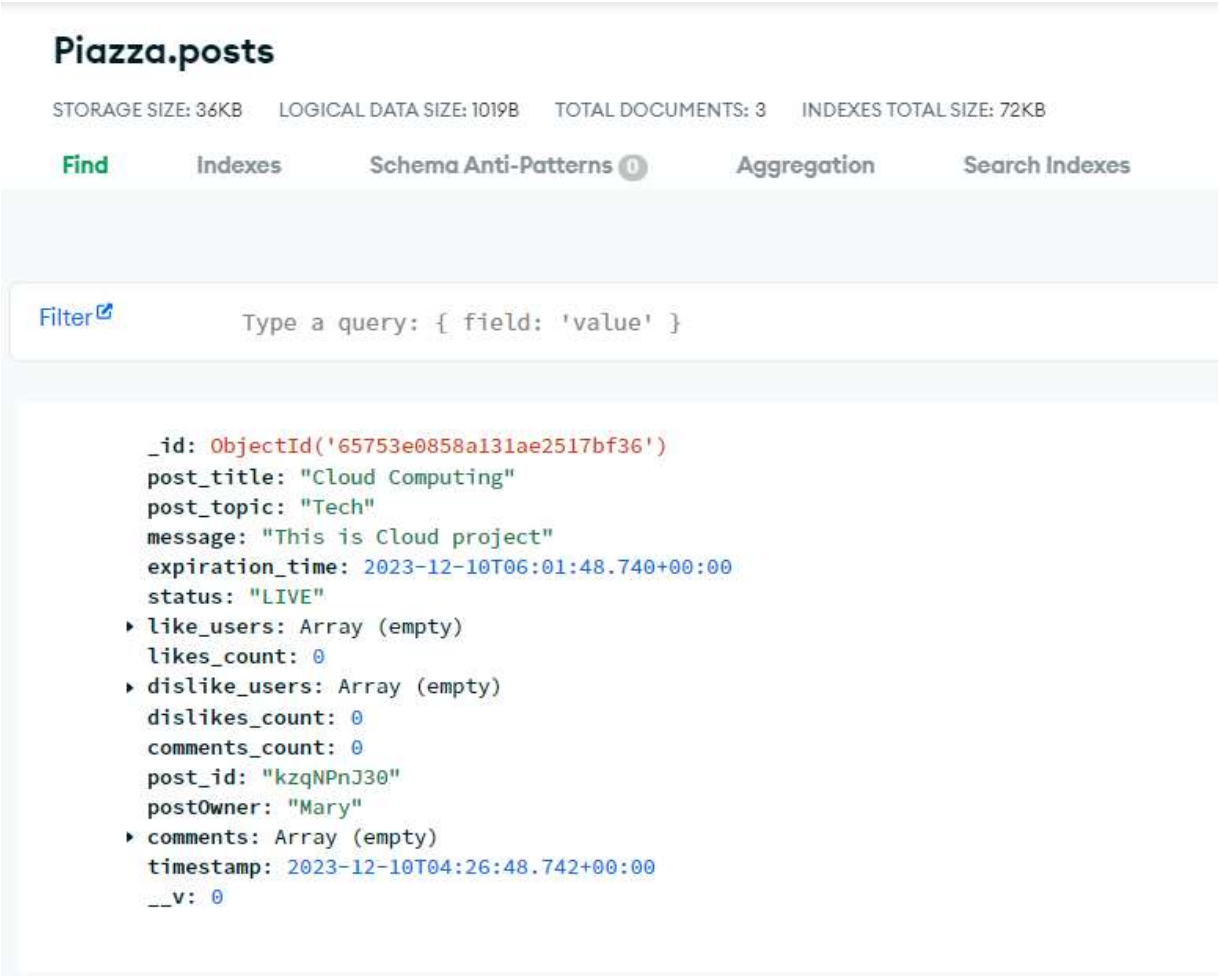
Nick posts a message in Tech topic with a n expiration time of 90 minutes





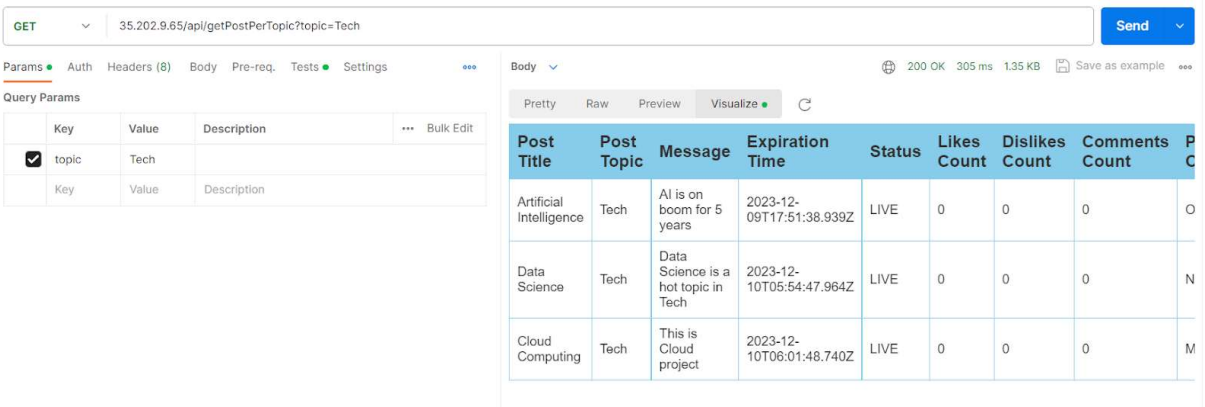
5.6 Test Case 6: Mary posts a message in the Tech topic with an expiration time using her token.





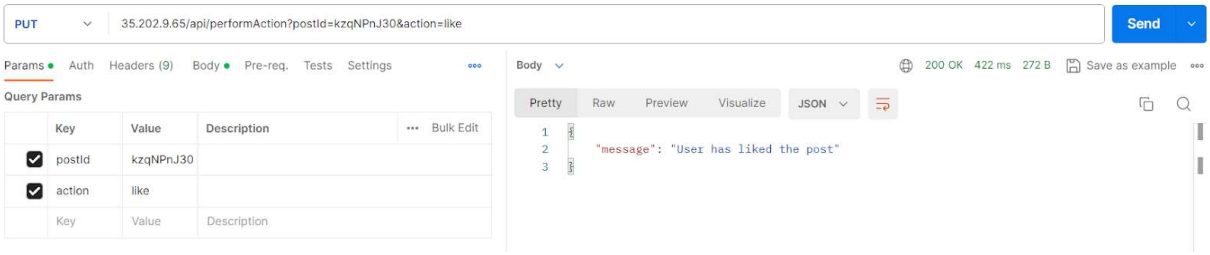
These two messages are added in the database as shown above

5.7 Test Case 7: Nick and Olga browse all the available posts in the Tech topic; three posts should be available with zero likes, zero dislikes and no comments.



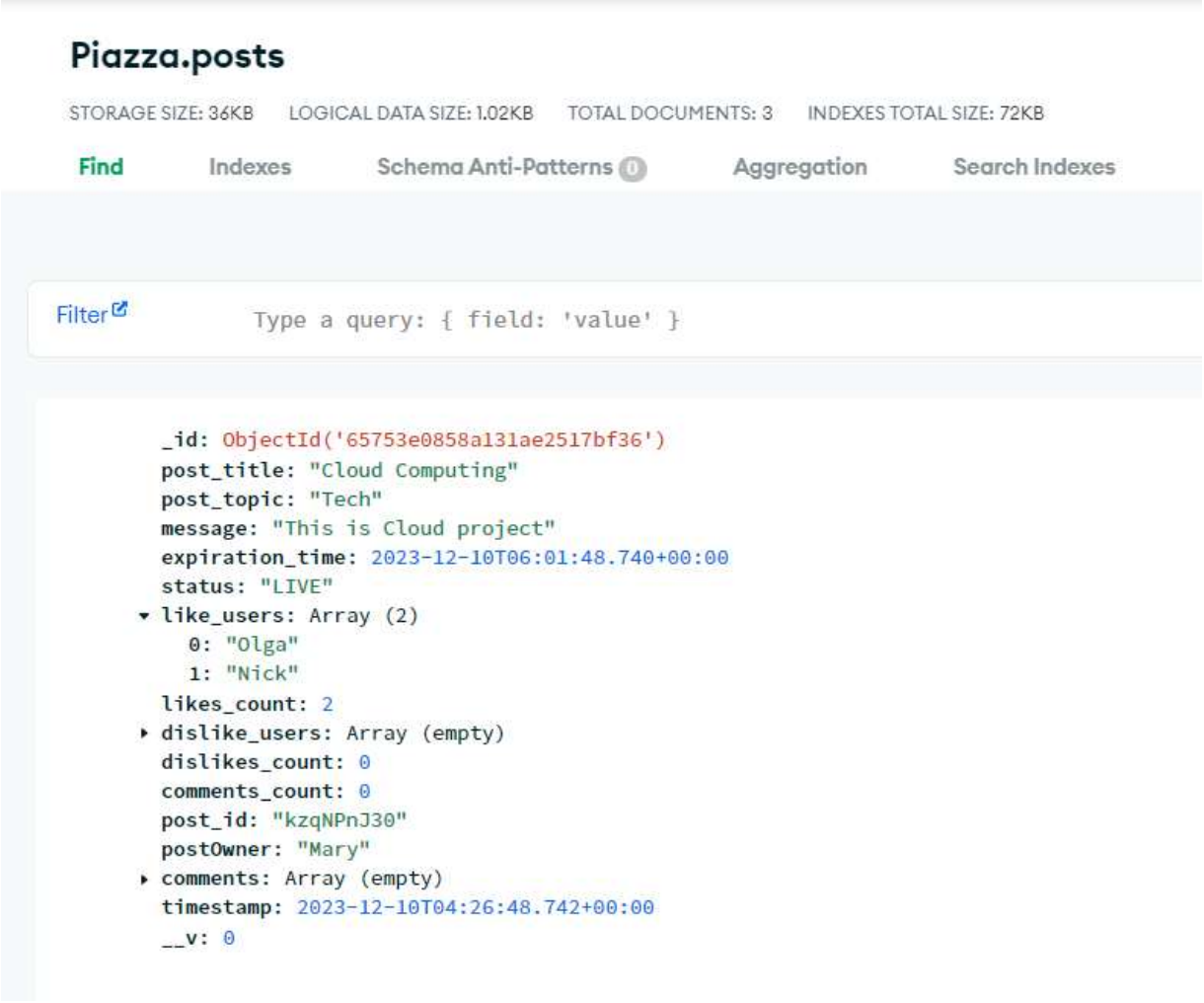
5.8 Test Case 8: Nick and Olga “like” Mary’s post on the Tech topic.

Olga likes Mary’s post on Tech topic



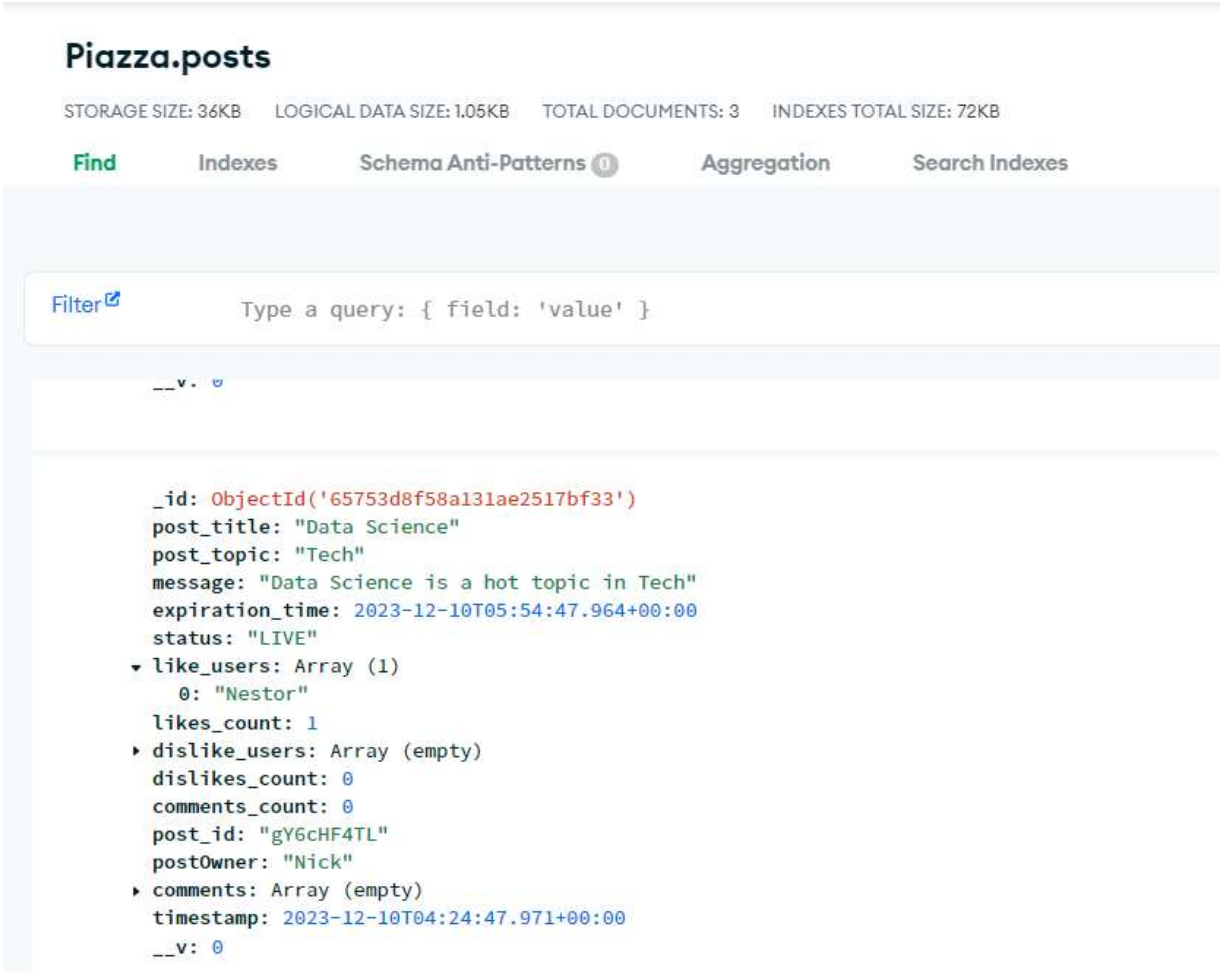
Nick likes Mary’s post on Tech topic

In the DB we can see likes_count as 2 and like_users are { Nick and Olga }

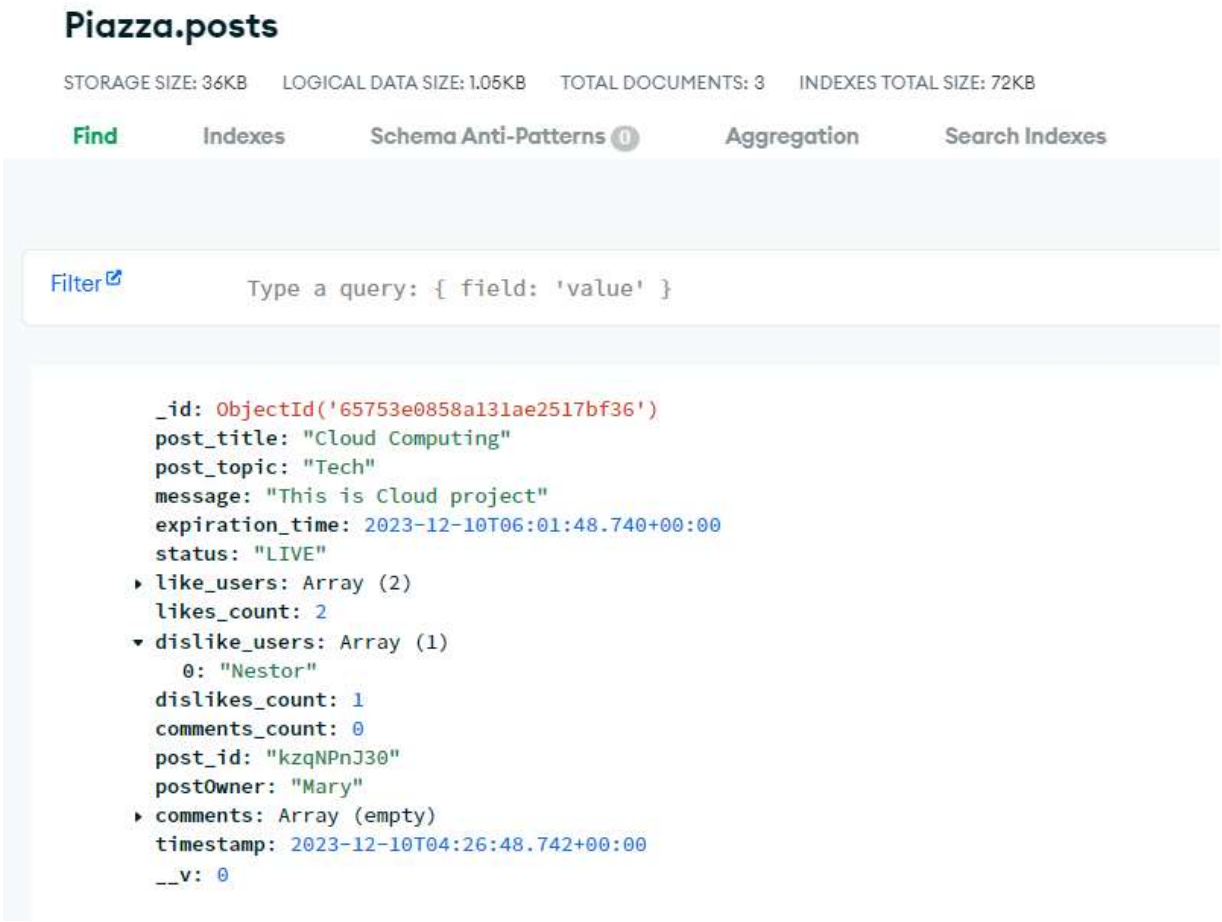


5.9 Test Case 9: Nestor “likes” Nick’s post and “dislikes” Mary’s on the Tech topic

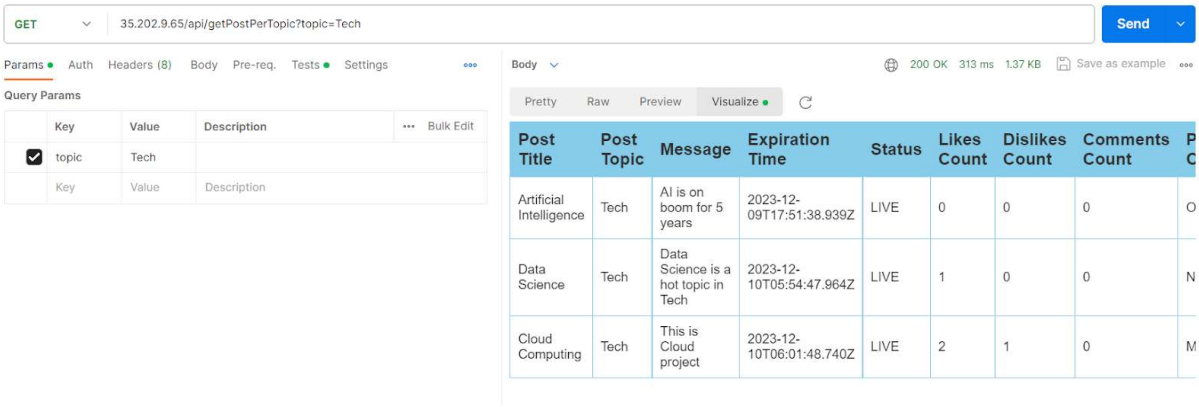
Nestor likes Nicks post on Tech



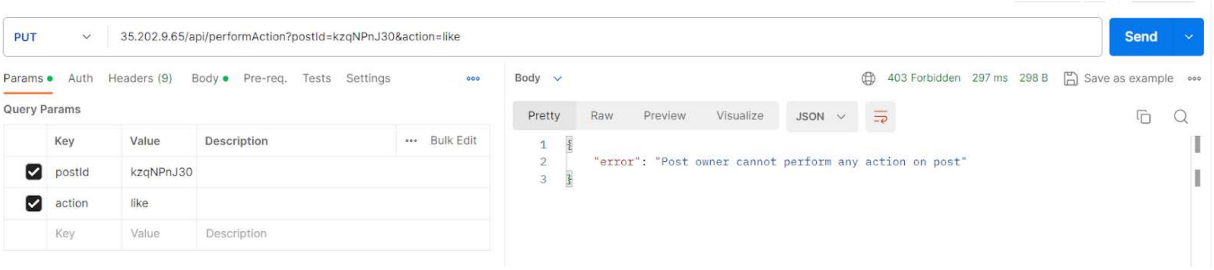
Nestor dislikes Mary's topic on tech



5.10 Test Case 10: Nick browses all the available posts on the Tech topic; at this stage, he can see the number of likes and dislikes for each post (Mary has two likes and one dislike, and Nick has one like). There are no comments made yet.

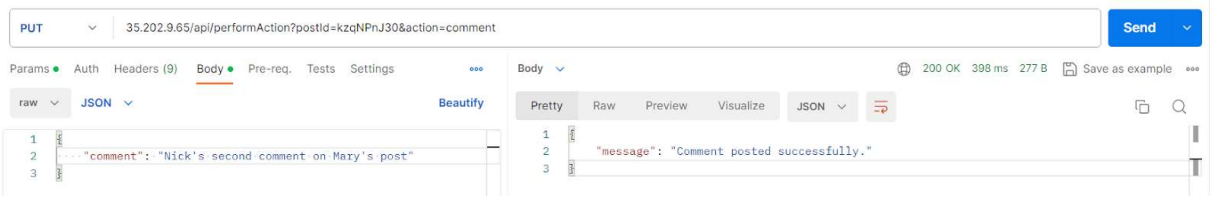


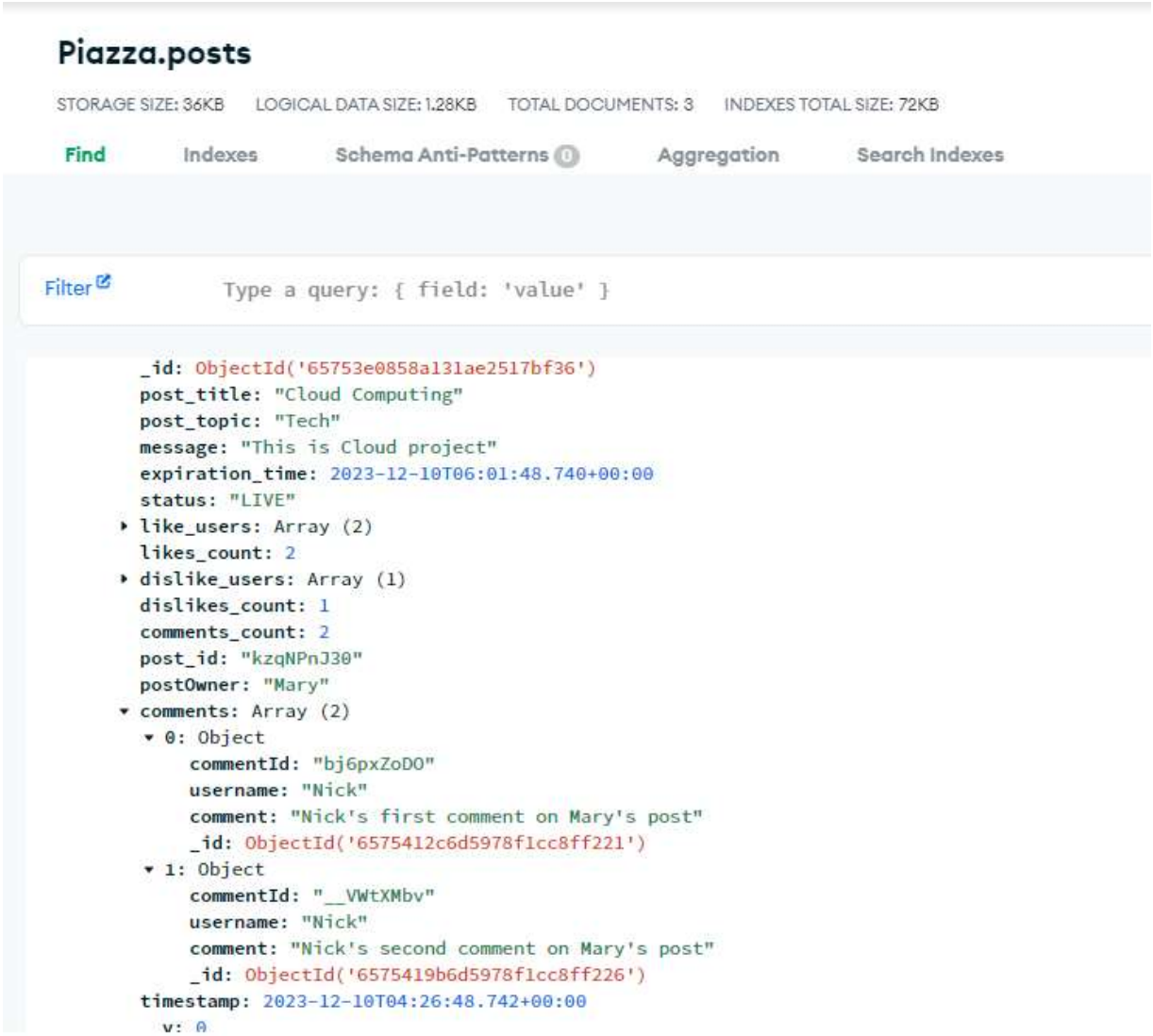
5.11 Test Case 11: Mary likes her post on the Tech topic. This call should be unsuccessful; in Piazza, a post owner cannot like their messages.



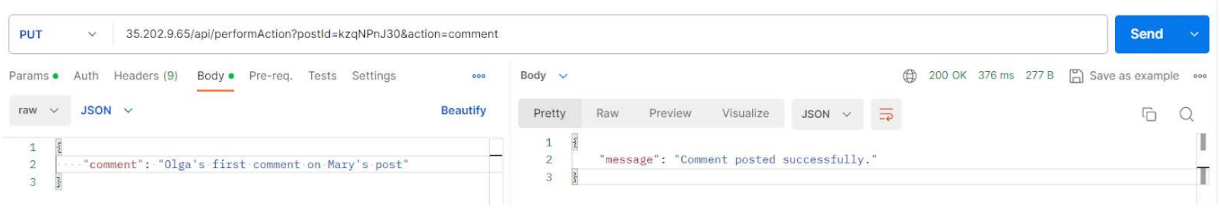
5.12 Test Case 12: Nick and Olga comment on Mary's post on the Tech topic in a round-robin fashion (one after the other, adding at least two comments each).

Nick comments on Mary's post

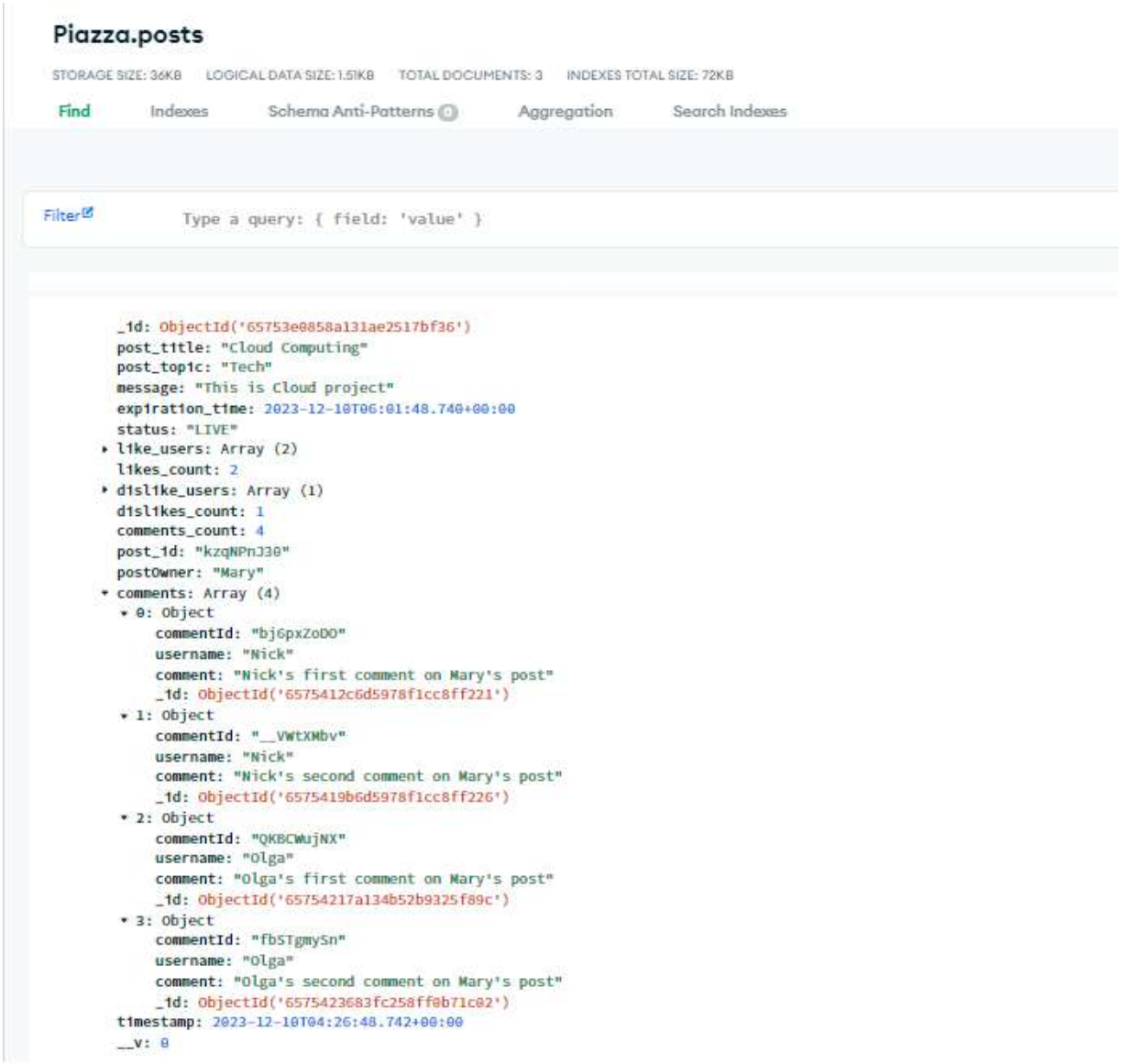




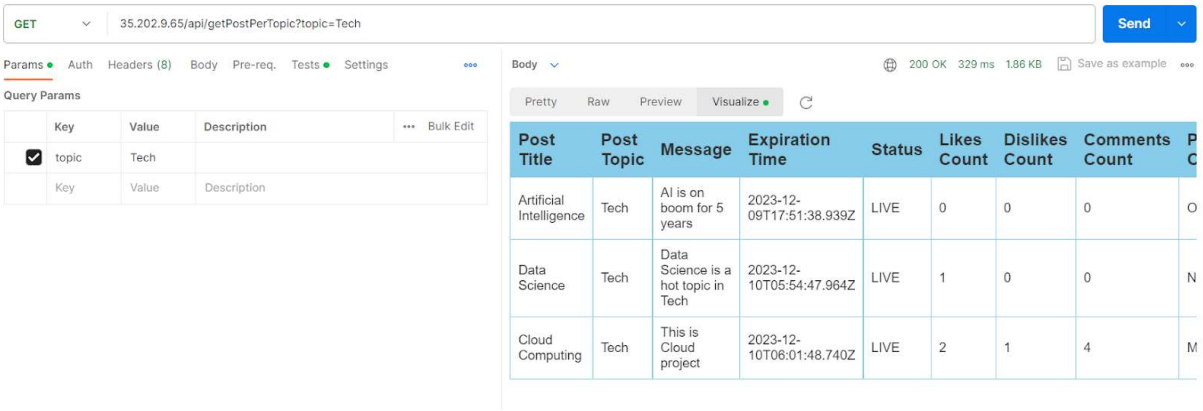
Olga comments on Mary’s post



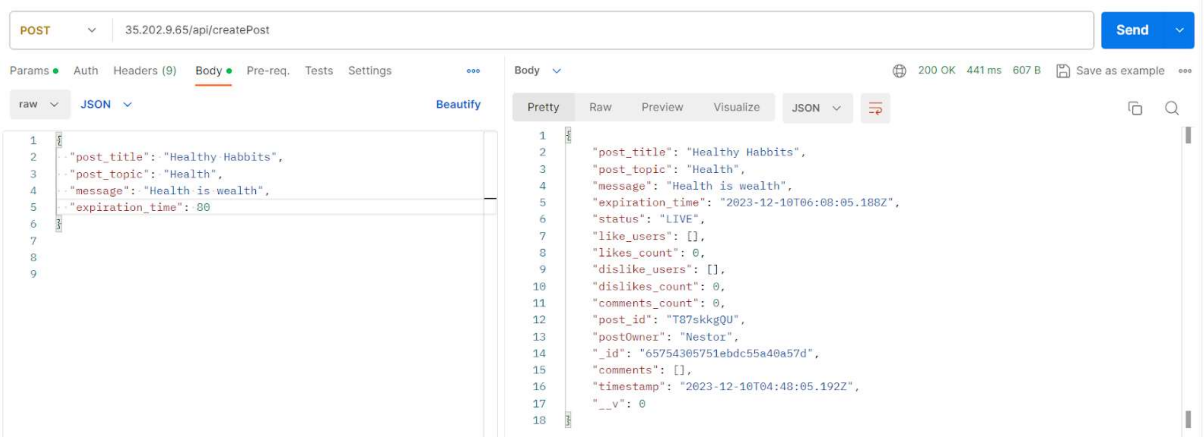

PUT
35.202.9.65/api/performAction?postId=kzqNPnJ30&action=comment
Send
Params
Auth
Headers (9)
Body
Pre-req.
Tests
Settings
raw
JSON
Beautify
{
"comment": "Olga's second comment on Mary's post"
}
Body
200 OK
377 ms
277 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"message": "Comment posted successfully."
}

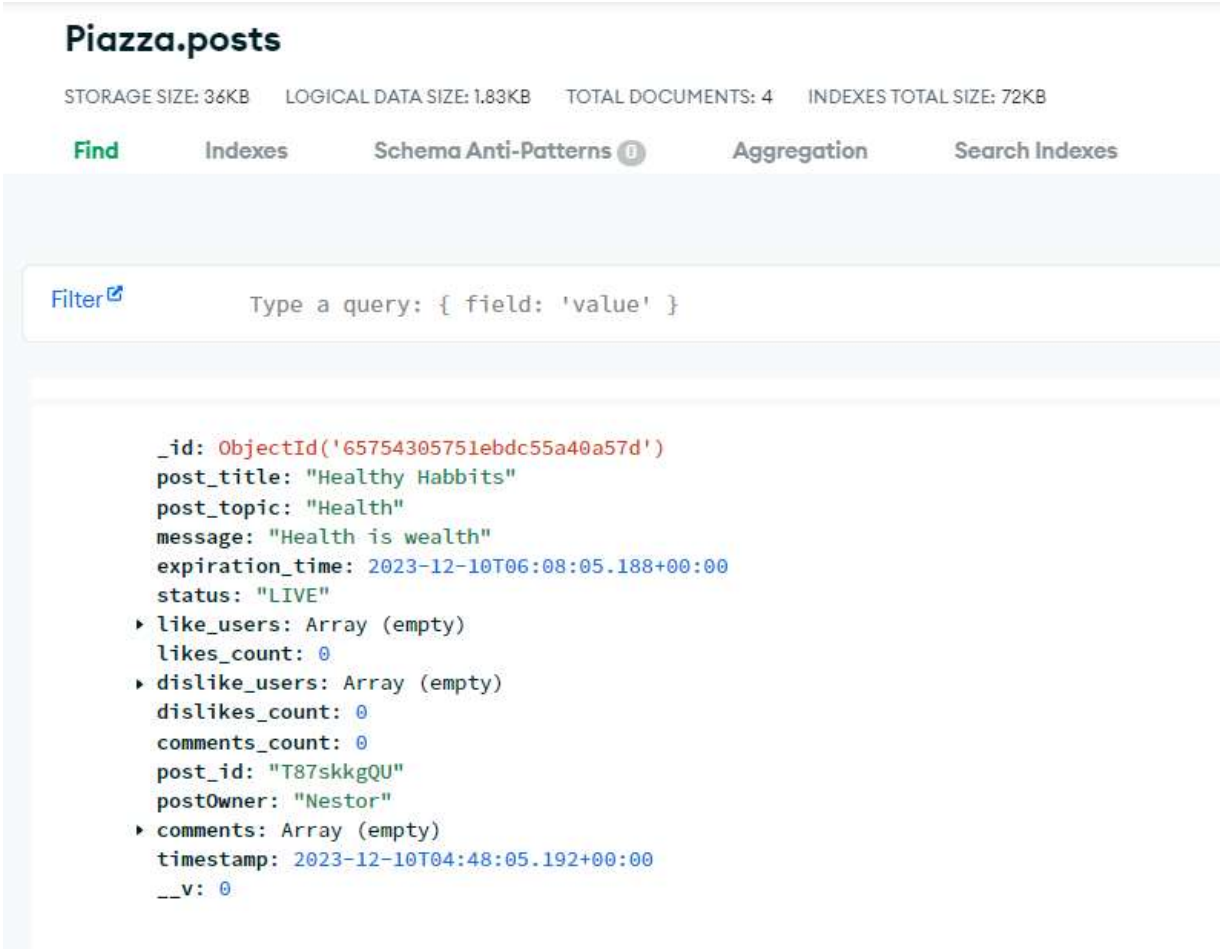

Piazza.posts
STORAGE SIZE: 36KB
LOGICAL DATA SIZE: 1.51KB
TOTAL DOCUMENTS: 3
INDEXES TOTAL SIZE: 72KB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
Filter
Type a query: { field: 'value' }
_id: ObjectId('65753e0858a131ae2517bf36')
post_title: "Cloud Computing"
post_topic: "Tech"
message: "This is Cloud project"
expiration_time: 2023-12-10T06:01:48.740+00:00
status: "LIVE"
like_users: Array (2)
likes_count: 2
dislike_users: Array (1)
dislikes_count: 1
comments_count: 4
post_id: "kzqNPnJ30"
postOwner: "Mary"
comments: Array (4)
0: Object
commentId: "bj6pxZoDO"
username: "Nick"
comment: "Nick's first comment on Mary's post"
_id: ObjectId('6575412c6d5978f1cc8ff221')
1: Object
commentId: "__VWtXMbv"
username: "Nick"
comment: "Nick's second comment on Mary's post"
_id: ObjectId('6575419b6d5978f1cc8ff226')
2: Object
commentId: "QKBCWujNX"
username: "Olga"
comment: "Olga's first comment on Mary's post"
_id: ObjectId('65754217a134b52b9325f89c')
3: Object
commentId: "fb5TgmySn"
username: "Olga"
comment: "Olga's second comment on Mary's post"
_id: ObjectId('6575423683fc258ff8b71c02')
timestamp: 2023-12-10T04:26:48.742+00:00
__v: 0

5.13 Test Case 13: Nick browses all the available posts in the Tech topic; at this stage, he can see the number of likes and dislikes of each post and the comments made.



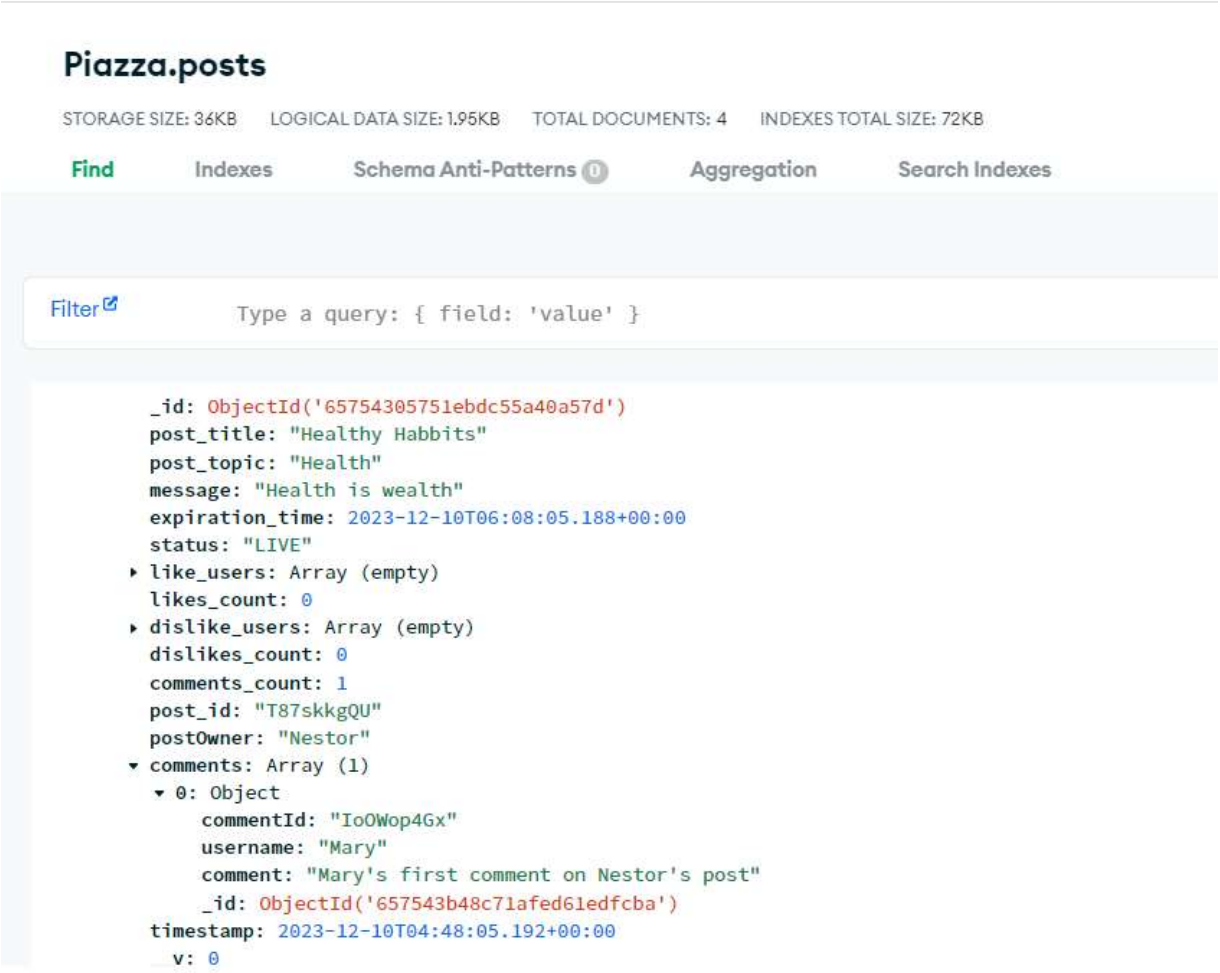
5.14 Test Case 14: Nestor posts a message in the Health topic with an expiration time using her token.



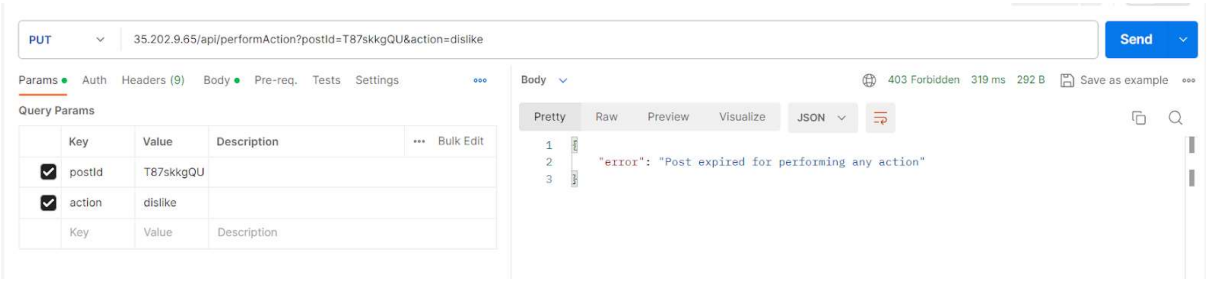


5.15 Test Case 15: Mary browses all the available posts on the Health topic; at this stage, she can see only Nestor's post.

5.16 Test Case 16: Mary posts a comment in Nestor's message on the Health topic.

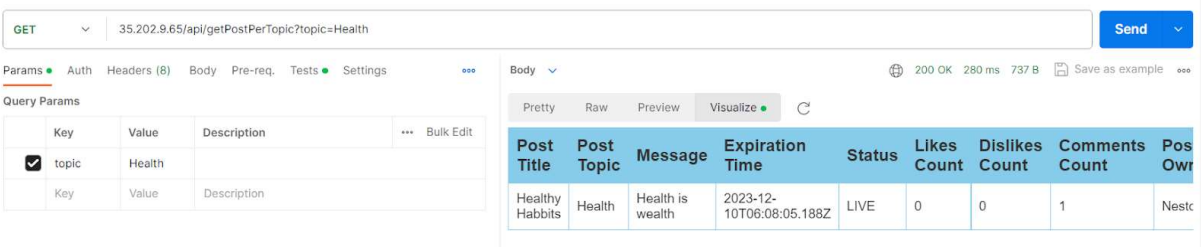


5.17 Test Case 17: Mary dislikes Nestor's message on the Health topic after the end of post-expiration time. This should fail

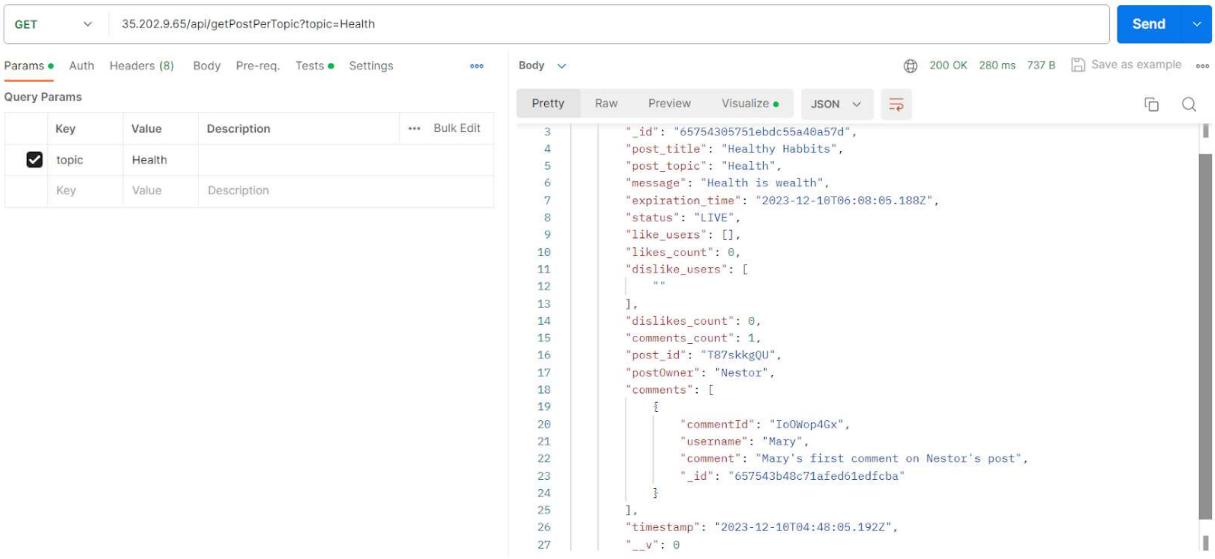


Mary dislikes Nestor's message on Health topic after the end of post-expiration, and this API call fails as the post is expired for performing any actions such as (likes/ dislikes/ comments)

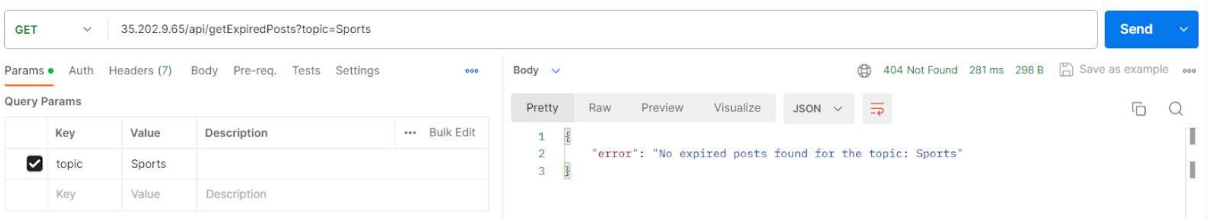
5.18 Test Case 18: Nestor browses all the messages on the Health topic. There should be only one post (his own) with one comment (Mary's).



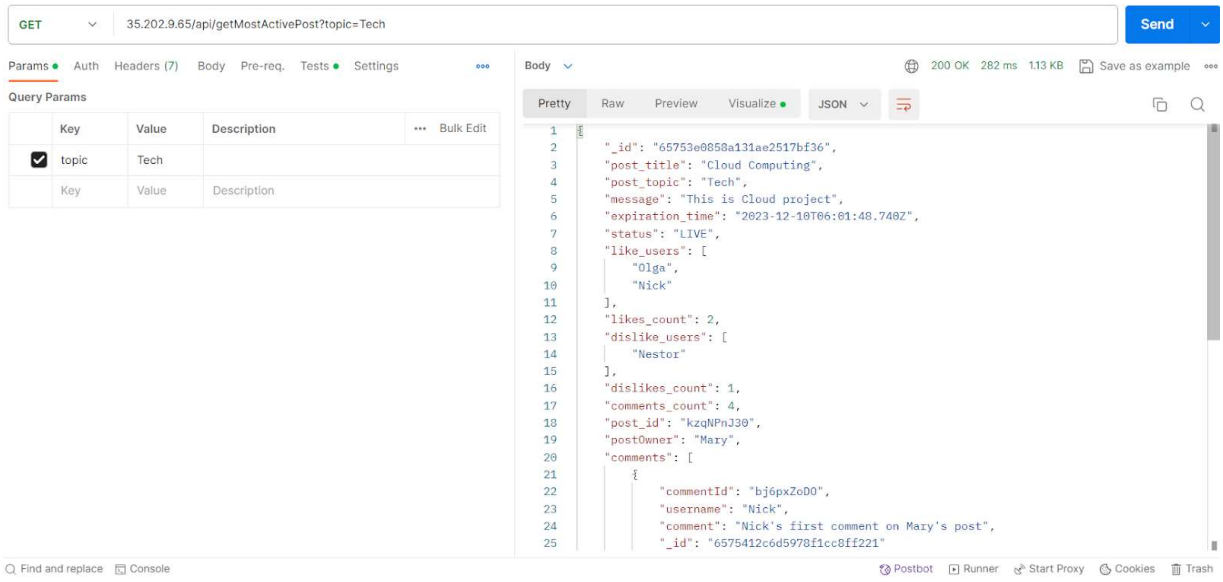
Json view of the response:



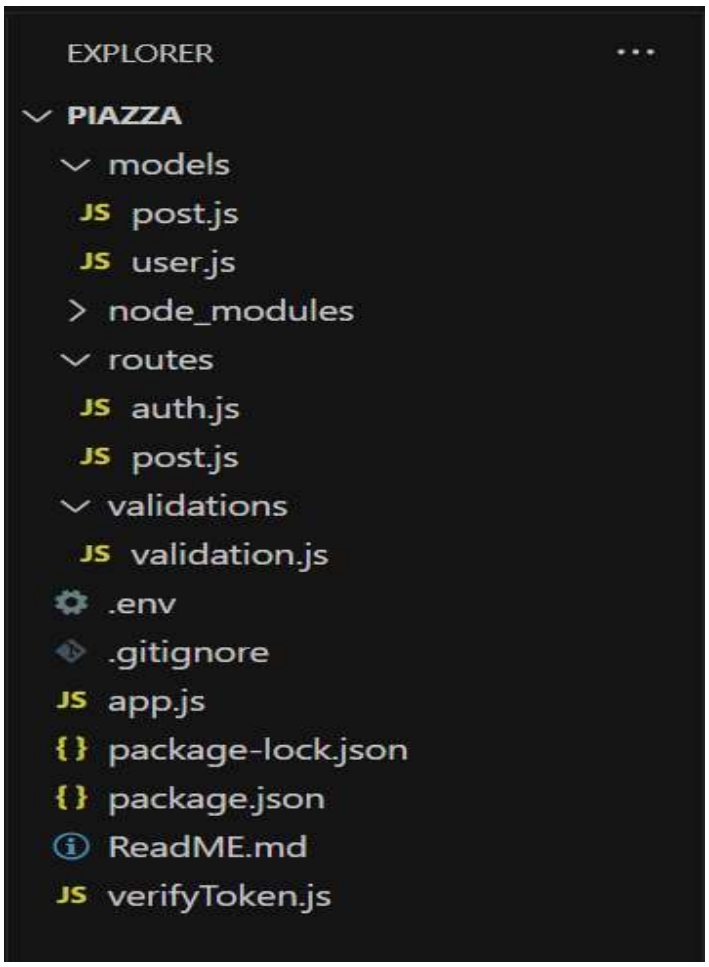
5.19 Test Case 19: Nick browses all the expired messages on the Sports topic. These should be empty.



5.20 Test Case 20: Nestor queries for an active post with the highest interest (maximum number of likes and dislikes) in the Tech topic. This should be Mary's post.



6.0 File Structure:



The Models folder: has the schema models for posts and users schema.
The Node Modules: has the imported node modules required for node project
The Routes folder: has two routers Auth and Post where Auth Router handles the registration of a new user to the Piazza API and login for the registered users.
The Validation folder: has a validation.js file which incorporates the validation of the user inputs using the JOI module, which is popular for data validation.
The .env file: has the Database connector URI (`DB_CONNECTOR`) for connection for the database to the Mongoose and the (`TOKEN_SECRET`) for the user.
The .gitignore file: To ignore the node_modules, npm-debug.logs and .DS_Store while committing the codebase to GitHub.
The app.js file: has the router path for both Auth and Post users, the database connection and the port localhost:3000 server configurations.
The package-lock.json and package.json: has the necessary packages required for implementation.
The ReadME.md file: has all the commands used for implementing the project
The verifyToken.js: handles the verification and authentication of users in Piazza API

7.0 Cloud Deployment

7.1 Docker

Commands to setup docker in GCP Virtual Machine

sudo apt-get install docker.io
sudo adduser docker-user
sudo usermod -aG sudo docker-user

Upload piazza application on github repository

` git init
git remote add origin https://YOUR_GIT_USERNAME:YOUR_GIT_TOKEN@github.com/ViplaviWade/Piazza.git
git add .
git commit -m “code push”
git push -f origin master

Switched to vm and used the following steps

su - docker-user
git clone https://YOUR_GIT_USERNAME:YOUR_GIT_TOKEN@github.com/heenal66/Piazza.git

cd Piazza

Created the below Dockerfile:

```
FROM alpine
RUN apk add --update nodejs npm
COPY . /src
WORKDIR /src
EXPOSE 3000
ENTRYPOINT ["node", "./app.js"]
```

Commands to build docker image

docker image build -t piazza-image .
docker container run -d –name piazza --publish 80:3000 piazza-image

```
docker-user@docker-vm:~/Piazza$ docker ps -a
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS
         NAMES
f6e7dc2f1f6b   piazza-image   "node ./app.js"   21 minutes ago   Up 21 minutes
00/tcp   piazza
docker-user@docker-vm:~/Piazza$
```

7.2Kubernetes

Commands to build and push piazza images to DokerHub:

```
docker image build -t heenalj6520/piazza-deployment
docker push wadeviplavi/piazza-app
```

Opened the GCP Cloud Terminal and connected to GCloud Cluster Link:

```
kubectl run piazza-app --image=wadeviplavi/piazza-app
```

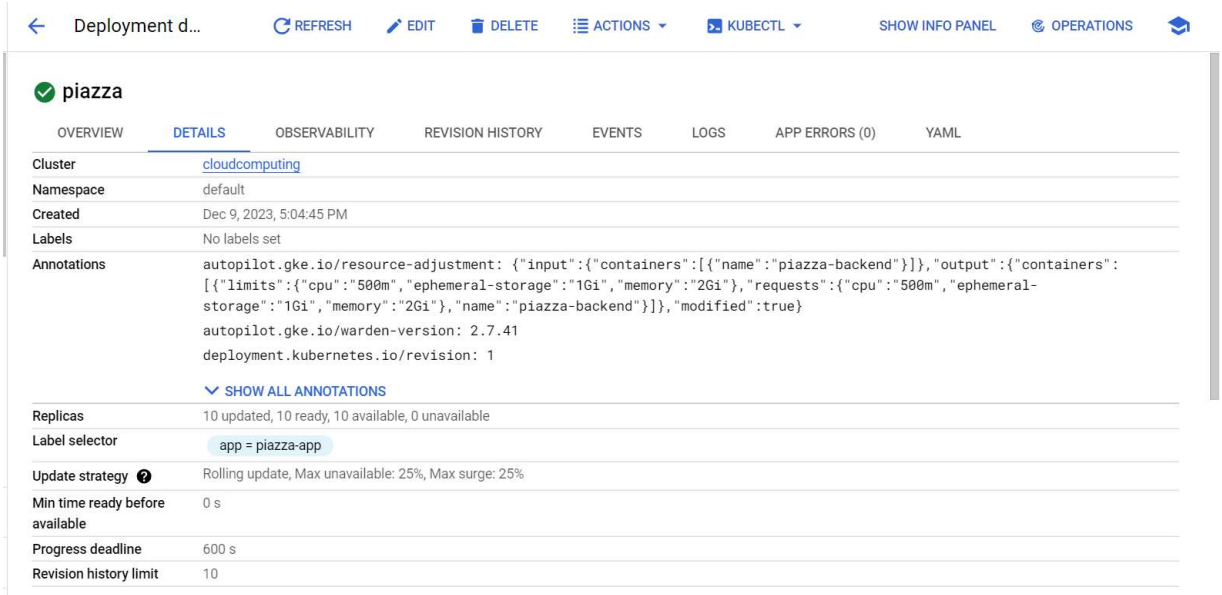
Create a deployment yaml file to create Pods:

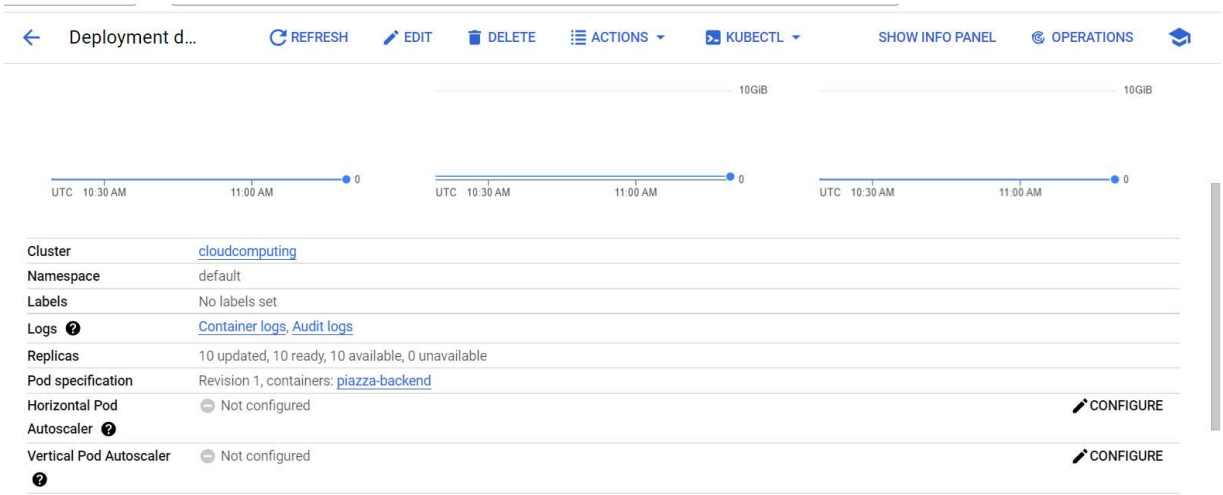
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Piazza
spec:
  replicas: 10
  selector:
    matchLabels:
      app: piazza-app
  template:
    metadata:
      labels:
        app: piazza-app
    spec:
      containers:
```

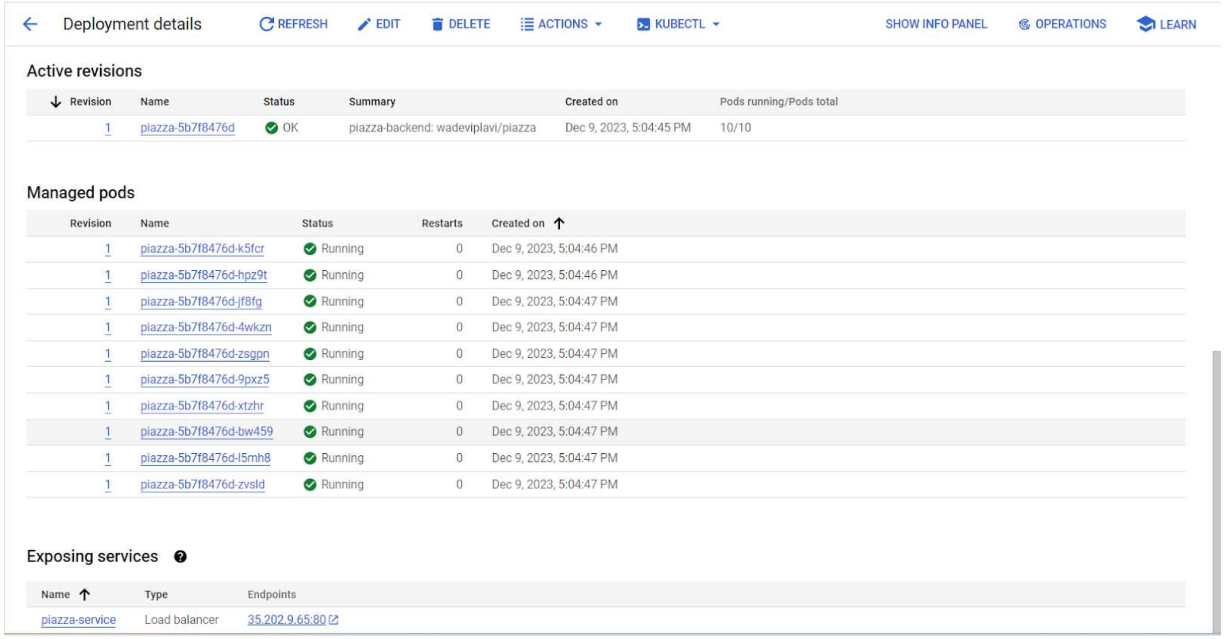
```
    - name: piazza-backend
      image: wadeviplavi/piazza
      ports:
      - containerPort: 3000

---

apiVersion: v1
kind: Service
metadata:
  name: piazza-service
spec:
  selector:
    app: piazza-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
```




Deployment d...
REFRESH
EDIT
DELETE
ACTIONS
KUBECTL
SHOW INFO PANEL
OPERATIONS
10GiB
10GiB
UTC 10:30 AM
11:00 AM
UTC 10:30 AM
11:00 AM
UTC 10:30 AM
11:00 AM
Cluster cloudcomputing
Namespace default
Labels No labels set
Logs Container logs, Audit logs
Replicas 10 updated, 10 ready, 10 available, 0 unavailable
Pod specification Revision 1, containers: piazza-backend
Horizontal Pod Autoscaler Not configured CONFIGURE
Vertical Pod Autoscaler Not configured CONFIGURE
Active revisions
Revision Name Status Summary Created on Pods running/Pods total
1 piazza-5b7f8476d OK piazza-backend: wadeviplavi/piazza Dec 9, 2023, 5:04:45 PM 10/10


Deployment details
REFRESH
EDIT
DELETE
ACTIONS
KUBECTL
SHOW INFO PANEL
OPERATIONS
LEARN
Active revisions
Revision Name Status Summary Created on Pods running/Pods total
1 piazza-5b7f8476d OK piazza-backend: wadeviplavi/piazza Dec 9, 2023, 5:04:45 PM 10/10
Managed pods
Revision Name Status Restarts Created on
1 piazza-5b7f8476d-k5fcr Running 0 Dec 9, 2023, 5:04:46 PM
1 piazza-5b7f8476d-hpz9t Running 0 Dec 9, 2023, 5:04:46 PM
1 piazza-5b7f8476d-jf8fg Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-4wkzn Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-zsgpn Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-9pxz5 Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-xtzhr Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-bw459 Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-l5mh8 Running 0 Dec 9, 2023, 5:04:47 PM
1 piazza-5b7f8476d-zvsld Running 0 Dec 9, 2023, 5:04:47 PM
Exposing services
Name Type Endpoints
piazza-service Load balancer 35.202.9.65:80

7.0 Validation and Authentication

Piazza implements rigorous data validation to ensure data integrity and user security. OAuth v2 is employed for secure user authentication, safeguarding user accounts from unauthorized access.

JWTs are a good way of securely transmitting information between parties because they can be signed, which means the users are identified with the unique tokens.

The async auth function covers the authentication part which authenticates the users with a unique token and maintains the user.id and user.username for the signed in user.

JS verifyToken.js > ...

```
const jwt = require('jsonwebtoken')

const User = require('./models/user')

async function auth(req, res, next) {

    const token = req.header('auth-token')

    if (!token) {
        return res.status(401).send({message: "Access denied."})
    }

    try {

        const verifiedToken = jwt.verify(token, process.env.TOKEN_SECRET)

        const user = await User.findById(verifiedToken._id)

        if(!user) {
            return res.status(401).json({ error: "Unauthorized. Invalid User"})
        }

        req.user = {
            _id: user._id,
            username: user.username
        }

        next()

    } catch {
        res.status(401).json({error: "Unauthorized. Invalid token"})
    }
}

module.exports = auth
```

8.0 API Endpoints:

1. CreatePost:

```
router.post('/createPost', verifyToken, async (req, res) => {

    const currentTime = Date.now();
    const expiration_time = req.body.expiration_time;
    const expirationTimeInMilliseconds = currentTime + expiration_time * 60 * 1000;

    const newPost = new Post ({
        post_id:shortid.generate(),
        post_title: req.body.post_title,
        post_topic: req.body.post_topic,
        message: req.body.message,
        status: 'LIVE',
        likes_count: 0,
        dislikes_count: 0,
        comments_count: 0,
        postOwner: req.user.username,
        expiration_time: new Date(expirationTimeInMilliseconds)
    })

    try {
        const savedRecord = await newPost.save()
        res.json(savedRecord)
    } catch (error) {
        res.status(500).json({error: error.message})
    }
})
```

2. **Update Post:**

```
router.put('/updatePost', verifyToken,async (req, res) => {
    const {postId} = req.query
    const {postTitle, postTopic, message} = req.body
    const loggedUser = req.user.username;

    try {
        const postExists = await Post.findOne({post_id: postId})

        if(!postExists) {
            return res.status(404).json({error: "Post not found"})
        }

        if(postExists.postOwner != loggedUser) {
            return res
            .status(403)
            .json({error: "Access Denied. You do not have the permission to modify this post"})
        }

        if(postExists.expiration_time < Date.now) {
            return res.status(403).json({error: "Post expired for update"})
        }

        postExists.title = postTitle || postExists.title
        postExists.post_topic = postTopic || postExists.post_topic
        postExists.message = message || postExists.message

        await postExists.save();
        res.status(200).json({message: "Post updated successfully."})
    }
    catch {
        console.error(error);
        res.status(500).json({ error: "Internal server error"})
    }
})
```

3. **Perform Actions (Like/ Dislike/ Comment):**

```
router.put('/performAction', verifyToken, async (req, res) => {

    const {postId, action} = req.query
    const loggedUser = req.user.username;
    const commentBody = req.body.comment
    const currentTime = Date.now();

    let userAlreadyPerformedAction = false;

    try {
        const postExists = await Post.findOne({post_id: postId})

        if(!postExists) {
            return res.status(404).json({error: "Post not found"})
        }

        if(postExists.postOwner == loggedUser) {
            return res
            .status(403)
            .json({error: "Post owner cannot perform any action on post"})
        }

        if(postExists.expiration_time < currentTime) {
            return res.status(403).json({error: "Post expired for performing any action"})
        }

        if(action == 'like') {
            if (!postExists.like_users || !postExists.like_users.includes(loggedUser)) {
                await Post.findByIdAndUpdate(
                    postExists._id,
                    { $push: { like_users: loggedUser }, $inc: { likes_count: 1 } }
                );
            } else {
                userAlreadyPerformedAction = true;
            }
        }
```

```
    if(action =='dislike') {
        if (!postExists.dislike_users || !postExists.dislike_users.includes(loggedUser)) {
            await Post.findByIdAndUpdate(
                postExists._id,
                { $push: { dislike_users: loggedUser }, $inc: { dislikes_count: 1 } }
            );
        } else {
            userAlreadyPerformedAction = true;
        }
    }

    if(action == 'comment') {
        const newComment = {
            commentId: shortid.generate(),
            username: loggedUser,
            comment: commentBody
        }
        await Post.findByIdAndUpdate(
            postExists._id,
            {$push: {comments: newComment}, $inc: {comments_count: 1}}
        );

        return res.status(200).json({message: "Comment posted successfully."})
    }

    if (userAlreadyPerformedAction) {
        return res.status(400).json({ message: `User has already ${action === 'like' ? 'liked' : 'disliked'} the post` });
    }

    res.json({ message: `User has ${action === 'like' ? 'liked' : 'disliked'} the post` });
    if (userAlreadyPerformedAction) {
        return res.status(400).json({ error: `User has already ${action === 'like' ? 'liked' : 'disliked'} the post` });
    }

} catch (error) {
    console.error(error);
```

4. **Get a Post by ID:**

```
router.get('/getpostById', verifyToken, async (req, res) => {

    const {postId} = req.query

    try {
        const postExists = await Post.findOne({post_id: postId})

        if(!postExists) {
            return res.status(404).json({error: "Post not found"})
        }
        res.send(postExists)
    } catch (error) {
        res.status(400).send({message: error})
    }
})
```

5. **Get Most Active Post:**

```
router.get('/getMostActivePost', verifyToken, async (req, res)=> {

    const {topic} = req.query

    try {
        const postExists = await Post.aggregate([
            { $match: { post_topic: topic } },
            {
                $addFields: {
                    totalInteractions: { $add: ['$likes_count', '$dislikes_count'] },
                },
            },
            { $sort: { totalInteractions: -1 } },
            { $limit: 1 },
        ]);

        if (!postExists || postExists.length === 0) {
            return res.status(404).json({error: `No active posts found for the topic: ${topic}`})
        }

        res.json(postExists[0]);
      } catch (err) {
        res.status(500).json({ error: err });
    }
})
```

6. **Get Post Per Topic:**

```
router.get('/getPostPerTopic', verifyToken, async (req, res) => {

    const {topic} = req.query

    try {
        const postExists = await Post.find({post_topic: topic})
        console.log(postExists)
        if(!postExists) {
            return res.status(404).json({error: "Post not found"})
        }

        res.send(postExists)
    } catch(error) {
        res.status(400).send({message: error})
    }
})
```

7. **Get Expired Posts:**

```
router.get('/getExpiredPosts',verifyToken, async (req, res)=>{

    const { topic } = req.query;
    const currentTime = Date.now();

    try {
        const postExists = await Post.find({
            post_topic: topic,
            expiration_time: { $lt: currentTime }
        }).sort({ timestamp: -1}).exec();

        if (!postExists || postExists.length === 0) {
            return res.status(404).json({ error: `No expired posts found for the topic: ${topic}` });
        }

        res.send(postExists)

    } catch (error) {
        res.status(500).json({error: "Internal server error"})
    }
})
```

8. **Get All Posts:**

```
router.get('/getAllPosts', verifyToken, async (req, res) => {

    try {
        const posts = await Post.find()
        console.log("Posts are : ", posts)
        res.send(posts)
    } catch (err) {
        res.status(400).send({message: err})
    }
})
```

9.0 Conclusion

In conclusion, Piazza offers a feature-rich social media experience with secure user authentication, interactive post functionalities, and efficient data storage using MongoDB. The implementation of OAuth v2 ensures a secure environment for users to engage and share content.