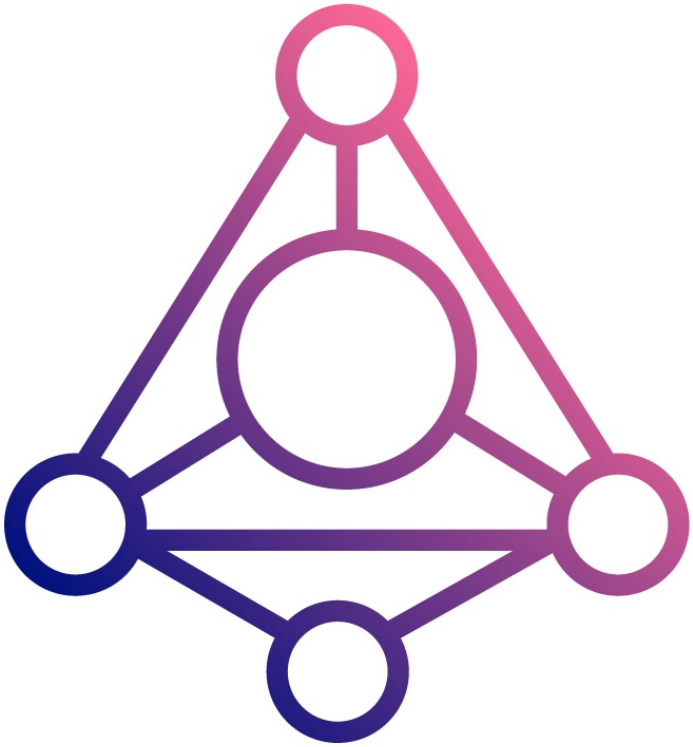
CLOUD COMPUTING

Course Work Report



Raabta.

Mishal Zulfiqar

[redacted]

Table of Contents

1. INTRODUCTION

1.1 REPORT OVERVIEW

This report explains in detail the concepts and software development methods that are used to build this course work project, Raabta. The report is written in a top-down approach, which starts with the high-level design of the system in section 2, moving onto cover each sub-system in detail. After explaining the design and implementation of the components, section 5 deals with the deployment pipelines used to host the application onto the cloud. Lastly, section 6 covers the testing of the application, where the 20 test cases are carried out and explained.

1.2 PROJECT OVERVIEW

Raabta is a cloud-native Software as a Service (SaaS) which provides its users a social media platform to interact through posts. Hence, the title 'Raabta' which is a word of the Urdu language, having meanings of Interaction and Connection.

1.3 TECHNOLOGIES USED

Backend: Node.js
Frontend: React.js
Database: MongoDB
Deployment: Docker, Kubernetes
Testing: System testing through React.js frontend

2. SYSTEM ARCHITECTURE

The architecture of Raabta is an overlap of the microservices architecture and the Single Page Application (SPA) architecture. Its architecture focuses on modularity and breaks down the web application into self-contained components of frontend and backend. All three components are loosely coupled and changes in the implementation of one do not affect the other.

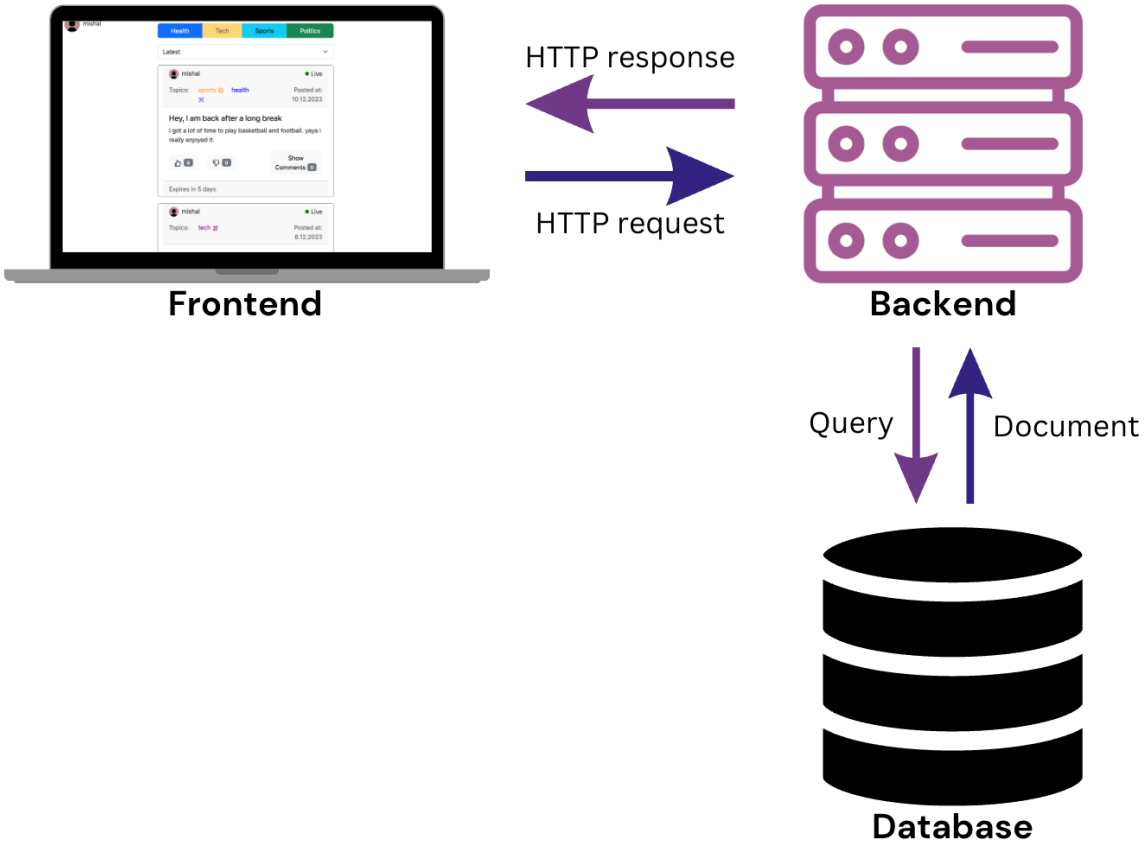


Figure 1: System Architecture

2.2 Frontend and Backend Interface

The Raabta frontend and backend interact only through APIs, exchanging data through JSON.

The frontend requests data from the server by sending a HTTP request with JSON data and receives back a JSON response. Based on this received JSON object from the server, the frontend changes its state and therein updating the content that is to be rendered onto the screen.

This loose coupling between the frontend and backend allows for greater flexibility, maintainability and separation of concern, where each component is abstracted and therefore can be built independently on any choice of technology.

This interaction between the two components is depicted in figure 2.

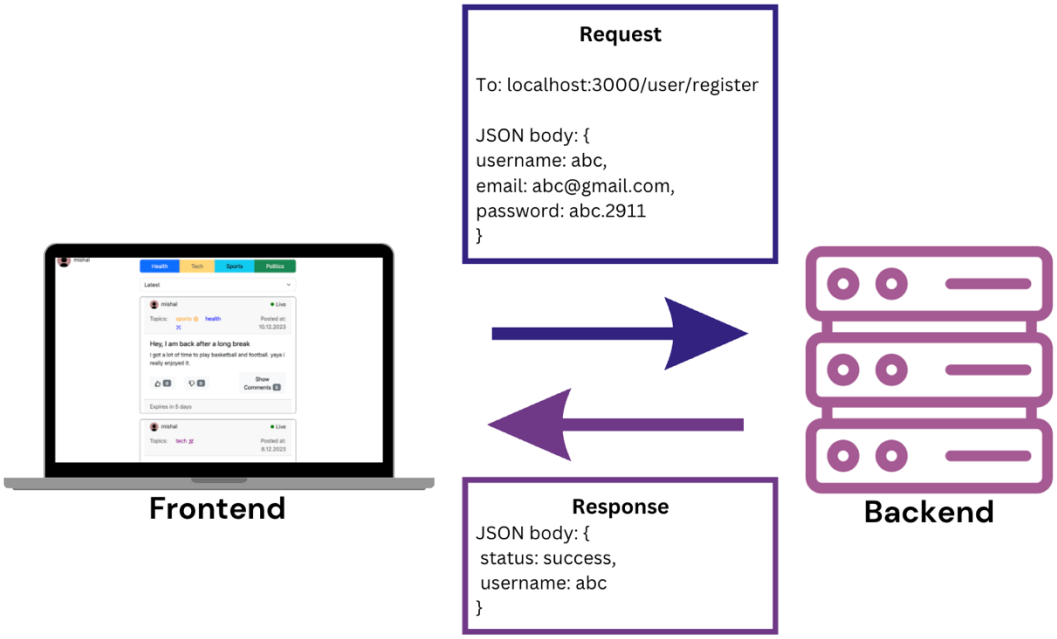


Figure 2: Interface between frontend and backend

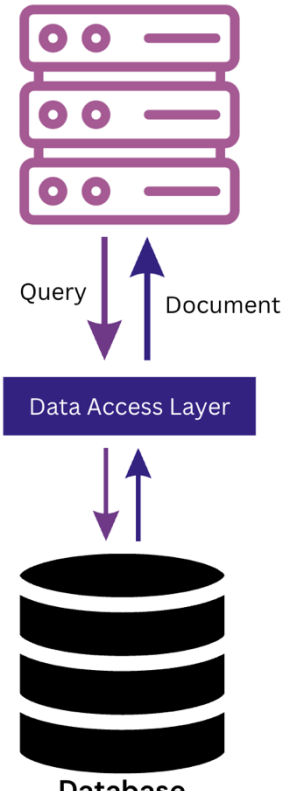


Figure 3: Interface between backend and database

2.2 Database and Backend Interface

The Raabta backend interacts with the database through an abstraction layer, called the Data Access Layer. In Raabta, this Data Access layer is provided by an Object Data Modelling (ODM) library.

The Backend interacts with the data access layer only and the interactions with the database are done under the hood.
Since Raabta uses a NoSQL database, this data access layer provides a higher-level, object-oriented abstraction, providing features such as schema enforcement, validations, the option of populating fields (similar to the concept of joins in Relational Databased) and more.

3. DEVELOPMENT TOOLS

This section covers the technologies that were used for developing the Raabta Application.

Visual Studio Code
The IDE used for the development of Raabta was Visual Studio Code because it is a lightweight and fast IDE that has a great support for JavaScript and node.js, also offering several useful and essential extensions. It also provides built-in integration for git which makes it easy to visually see the status of files as changes are made.

GIT
GIT is a version control system. It was used in the development of Raabta due to its features of providing the ability to push the application code to remote repositories and the ability to access code from different host machines. This feature of Git was used to deploy the application to the cloud.

JIRA
JIRA is a commonly used project management and collaboration tool. Its features such as visual representation of the development process, efficient tracking and categorization of tasks, prioritization and scheduling helped managed the workflow for Raabta.

4. IMPLEMENTATION DETAILS

4.1 DATABASE

Raabta uses a MongoDB which is a NoSQL database, as mentioned briefly in section 2.2 and stores data in form of documents. These documents in MongoDB are organised into collections, which corresponds to tables in relational databases.

The database in Raabta is used through the data access layer as discussed in section 2.2. The ODM library used for providing this abstraction is Mongoose. It streamlines the process of integrating the document-based MongoDB with JavaScript which is Object-Oriented. For this integration, mongoose maintains a strict schema which is followed for inserting documents in MongoDB, built on top of this mongoose then provides the functionality of creating models and making it possible to work with objects.

Querying the database in Raabta is also then carried out through various functions that mongoose provides and it populate function is used for performing joins in certain scenarios.

4.1.1 CONSIDERATIONS

Before designing Mongoose Schemas for Raabta, which would then become our MongoDB document structure, these factors were taken into consideration and the Schemas are based on these considerations. (Karlsson, n.d.)

- **Is the application read heavy or write heavy?**
  The application has little difference in the amount of approximated read and write queries, with leaning slightly onto read operations, because even though a user can write several posts, and comment on each post, the read queries on average will be slightly higher since the post wall has to always be populated with posts, increasing with every write operation.

- **What data is frequently accessed together?**
  Each time a post is rendered, the likes of the post are needed with it. The comments are loaded separately when the user asks to see the comments section of the post. Likes are never read on their own.

- **What are your performance considerations?**
  The user should be able to see the posts on each topic instantly to have a decent user experience therefore read operation needs to be fast. The write operations can afford a little delay (in seconds).

- **How will data grow?**
  The user documents will grow linearly as more users join the platform. The posts documents can on the other hand, have unbounded arrays of likes, dislikes and comments as growing number of users interact with each post.

4.1.2 MODEL DESIGNS

According to the considerations explained above, there are three models created with mongoose Schemas, which are depicted in the entity relationship diagram below.

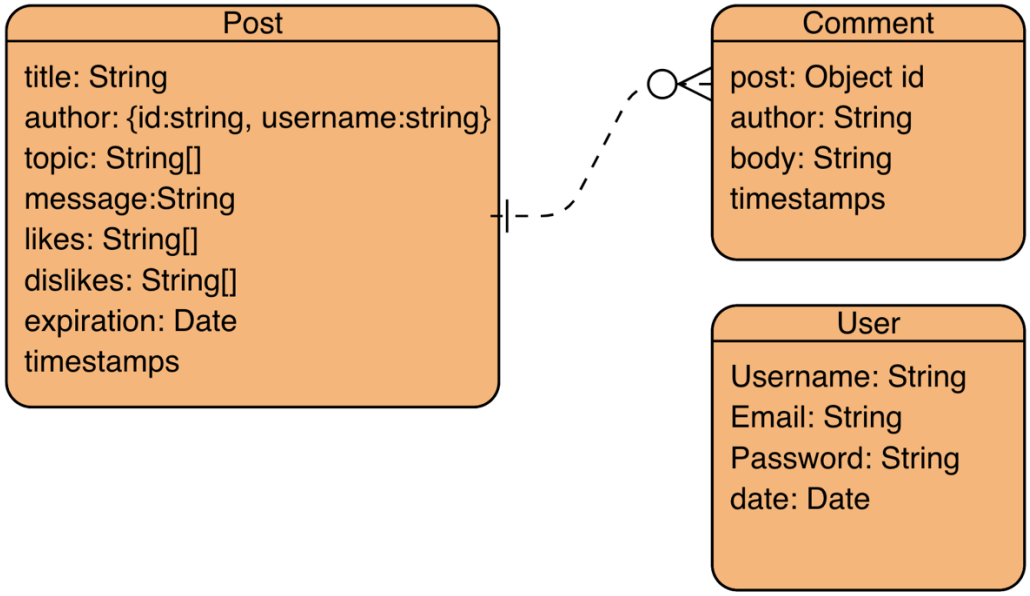


Figure 4:Entity Relationship Diagram

Post Schema
The complete Post Schema is given in Appendix A.

- The **title**, **message** and **expiration date** are one-to-one properties.
- The **author** stores two more properties, the ID and the username. This is better than directly storing the ref of the user because this would not cause the password to also be exposed with retrieval after a join is performed, moreover joins can be computationally expensive and therefore storing the two required fields would avoid performing a join.
- The **topic** is an array of strings and since can only be of the values [‘health’, ‘tech’, ‘sports’, ‘politics’], this is defined with the Enum property.
- The **likes** is simply an array of strings that stores the IDs of the users that have liked the post, in string. The reason for storing the IDs as a string as not as an Object ID is because we don’t need any other property of the user and therefore there is no need to populate or perform a join on the likes and users.
- The **dislikes** array is designed similarly.
- **Timestamps** option is included which adds two fields to each document (createdAt and updatedAt). Currently, only the createdAt is used, but updatedAt can be useful when a future feature of updating the post is to be implemented.

Relationship between Post and Comments

For implementing the relationship between comments and post, there were three available options:

- **Embedding comments objects within the post schema:** This meant storing all the comment information as object in the comments array of the post schema. Although such an implementation would avoid performing a join, it would have become difficult as the application scaled. As more users commented on each post, each document would have become excessively large, making the database unmaintainable.

- **Storing references within the post schema:** In this option, the reference IDs of the comment object would be stored in the comments array of the post schema. This option would require performing a join when the comments are to be read, and for writing a new post, a new comment object would have to be created and then afterwards, the ID of the created comment would have to be pushed into the comments array of the post.

- **Storing post reference within the comment schema:** This option stores the post reference inside the comment object. This makes both reading and writing comments easy since the post ID is sent in the URL. This is the option that has been implemented.

Comment Schema

The complete comment schema is given in Appendix B. It includes the post reference as discussed above, author which is of type string to store only the username of the author, this is because we only require the username to display alongside the respective comment under a post. The schema also includes the body which is of type string and the timestamps.

User Schema

The user schema is straightforward and has 4 one-to-one properties, the username, email and password are asked at the time of registration. The email and password are to be used for login. The date property is to keep track when the user was created and could be used for a future feature that could be implemented where the users could see how old their accounts are.

4.2 BACKEND

4.2.1 USER AUTHORIZATION

User authorization is managed through the usage of access tokens. A new user registers onto the application. If the data passes validation, the user is registered. Now, the user logs in with his email and password, if the data is valid and correct, the server generates a JSON web token (JWT) and sends it back to the client (frontend).

To perform any other operation, such as browsing posts, uploading a post, liking or disliking a post, a valid JWT needs to be sent with the request to the browser, otherwise the request will be denied, and an authorization error will be sent back to the client. On receiving this error, the client will redirect the user to the login page. This process can be understood by the following diagram.

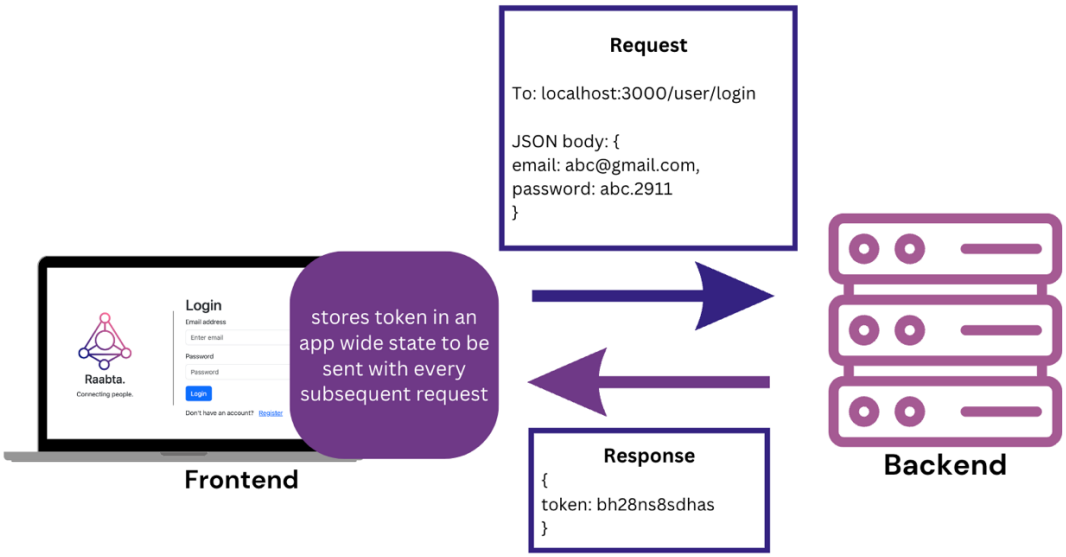


Figure 5: User Authorization Process

4.2.2 RESTFUL API DEVELOPMENT

All the endpoints of the Raabta server are enumerated in the table below. The API endpoints are designed using RESTFUL approach.

MODEL	TASK	HTTP Method	API
User	Register User	POST	/user/register
	Login User	POST	/user/login
Post	Add Post	POST	/posts
	Get Posts by Topic	GET	/posts/:topic
	Like a post	POST	/posts/:postid/like
	Dislike a post	POST	/posts/:postid/dislike
Comment	Add comment on a post	POST	/posts/comments/:postid
	Get all comments on a post	GET	/posts/comments/:postid

- The application always displays post with respect to the topic and therefore, there is no API to get all posts regardless of the topic.

- To add and get comments, /posts/comments/:postid is used instead of /posts/:postid/comments which is more intuitive, because of code readability, maintainability and separation of concern since it makes the extraction of the comments' APIs into another routes file possible. So, if, in future work, more API endpoints need to be developed to delete and edit a comment, it could be done in a readable manner.

4.2.3 HANDLING INTERACTIONS

In Raabta interactions on a post include liking a post, disliking a post and adding a comment. These are to be completed only if the post is not expired. To ensure this, a middleware is applied to all the API endpoints for these interactions, which sees if the post has not expired. If it is expired, an error is sent back, otherwise the interaction is completed and reflected onto the frontend. For each interaction, as discussed earlier, the client sends the JWT token within the header of the request to be able to perform the interaction.

4.2.4 USER INPUT VALIDATIONS

The user inputs which are sent to the server with a request by the client are validated before performing any database operation. These validations are done through JOI library, which is a JavaScript library that provides all essential validation methods.

A schema is designed using Joi with all the constraints, so when the user input is compared with the validation schema, all the constraints are validated, and an appropriate message is sent back to the client.

The Joi Validation Schemas are created for when a user enters data for registration, logging in, adding a new post and adding a new comment and are given in the Appendix C, D, E and F respectively.

All validations are self-explanatory, and the exceptions are explained below.

- To ensure that a post can have multiple topics from the 4 options of ['health', 'tech', 'sports', 'politics'], the following validations were used.

```
topic: joi.array().items(joi.string().valid("politics", "health",
"sports", "tech").required()).required()
```

  Joi.array() tells Joi that topic is an array, items() define how each item of the array should look like, which is that it should be a string and valid() ensures that each item is only from the options given. Topic is required but it cannot be empty so required is used inside the items() as well.

- To ensure that the expiration date of the post has not passed, at the time the post is getting added, the following validations were used.

```
expiration: joi.date().greater("now").required()
```

It is to note that expiration of a post in Raabta can only be defined as a date and not a specific time, therefore a user cannot add a post with the expiration date of the same day.

4.3 FRONTEND

The frontend of Raabta is built on React.js which is a JavaScript library for building dynamic User Interfaces as Single Page Applications. Since, frontend is out of the scope of the project, it is not covered in detail and can be found within the uploaded scripts. However, the relevant implementations are enumerated below.

- Once the user login is successful and a token is received, it is stored in the app wide state (context) of the frontend.
- The fetch library is used to send HTTP requests to the Raabta backend. With every request, the token from the context is attached to the request.
- If an authorization error is sent from the server, the token is cleared from the context and the user is redirected to the login page.
- To connect to the backend, to the container on VM or to the Kubernetes Cluster, only the proxy property within the package.json is to be set to the IP of the server.

5. DEPLOYMENT

5.1 DOCKER

To host the application onto the cloud, Docker is used. Docker is a tool used to containerize an application for it to be portable enough to be run on any host machine. There are two ways to perform this task: building an image locally, pushing to DockerHub, pulling from the host VM and pushing the code to GitHub, pulling the code in the host VM, building an image on the VM. The second deployment pipeline was followed, since to install Docker on the VM is more convenient that installing it locally.

The following commands were used to deploy Raabta onto the Google Cloud Platform using Docker:

- **Setting up the Virtual Machine**: In this step the VM instance is created, a sudo user is created, Docker is installed, and the directory is switched to the created user's home directory.
- **Cloning repository from Github:** The application code is first pushed to Github, from there it is then cloned to the home directory of the sudo user.
- **Add a Dockerfile:** A Dockerfile is then to be created in the folder of the cloned repository. The following code was added to the Dockerfile for Raabta.

```
FROM node:lts-alpine
RUN apk add --update nodejs npm
ENV DB_URL mongodb+srv://mishalzulfiqar2911:<password>
@cluster0.ktmbatl.mongodb.net/raabta?retryWrites=true
&w=majority
ENV TOKEN_SECRET mishalisadmin2911
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8000
ENTRYPOINT ["npm", "start"]
```

  Two ENV variables are added DB_URL and TOKEN_SECRET. (For privacy, password is replaced by *<password>* in the DB_URL). Since the node modules are not included with the code that is cloned, it is necessary to add *RUN npm install,* and that to0 after setting the WORKDIR.

- **Build the image** using the docker image build command.
- **Run the container** by using the docker container run command with the publish tag to connect the port 80 of the container with the port 8000 of the application.
- **Connect the frontend** with the application on the VM by changing the proxy property in the package.json of the frontend to the external IP of the VM to forward the requests to the containerized application.

5.2 KUBERNETES

To provide the ability to easily scale our cloud application as the traffic on the application grows, Kubernetes is used. To deploy Raabta application on a Kubernetes cluster, the following steps were followed.

- **Push the built image to DockerHub** by first logging into the previously created sudo user on the VM.
- **Create a cluster:** A cluster was created on the Google Kubernetes Engine with the following configurations:
    - Name: raabta-cluster
    - No of nodes: 1 [Nodes were set to 1, since 110 pods can run on one node and currently the application is only in its prototype phase. It can be increased in the future as the application scales]
    - All other settings were left to the default values.
- **Connect the Google Cloud Shell** to the GKE cluster by the connect command that can be found on the three dots on the right side of the cluster definition. This way we get access into the cluster and can set up configurations for deploying pods into the node.
- **Create a deployment config file** by creating a file called raabta-cluster.yaml that configures the cluster settings and the following code was entered into the file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: raabta-deployment
  labels:
    app: raabta
spec:
  replicas: 5
  selector:
    matchLabels:
      app: raabta
  template:
    metadata:
```

```
    labels:
      app: raabta
  spec:
    containers:
    - name: raabta
      image: mishalz/raabta:1
      imagePullPolicy: Always
      ports:
      - containerPort: 8000
```

The replicas were set to 5 since it is asked of the coursework. This will create 5 pods into the one node of the cluster.

- **Apply the cluster configuration** to set up a GKE cluster by using the ***kubectl apply*** command.

  The created pods can be seen by the ***kubectl get pods*** command. The output of the command below tells us that the 5 pods have been created and are now running.

```
mishal_zulfiqar2911@cloudshell:~ (cloud-class-401110)$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
raabta-deployment-7d66b4d985-5tbhk  1/1     Running   0          8h
raabta-deployment-7d66b4d985-7vthk  1/1     Running   0          8h
raabta-deployment-7d66b4d985-gflqt  1/1     Running   0          8h
raabta-deployment-7d66b4d985-nksrf  1/1     Running   0          8h
raabta-deployment-7d66b4d985-tfglr  1/1     Running   0          8h
```

- **Create a load balancer**: To define a load balancer that would divide the traffic between the 5 pods, a service config file is created with the name raabta-service.yaml. The following code is added to the service config file.

```
apiVersion: v1
kind: Service
metadata:
  name: raabta-service
  labels:
    app: raabta-service
spec:
```

```
type: LoadBalancer
ports:
- name: http
  port: 80
  protocol: TCP
  targetPort: 8000
selector:
  app: raabta
sessionAffinity: None
```

- **Apply the service config file** by again using the ***kubectl apply*** command on the service config file.
- **Get the complete cluster information** by the ***kubectl get services*** command.

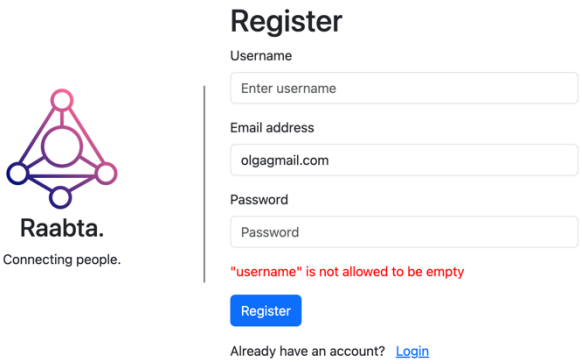
```
mishal_zulfiqar2911@cloudshell:~ (cloud-class-401110)$ kubectl get services
NAME             TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes       ClusterIP      10.96.0.1      <none>        443/TCP        8h
raabta-service   LoadBalancer   10.96.11.221   34.70.3.216   80:31514/TCP   8h
```

The Kubernetes is the cluster that was created, and raabta-service is the load balancer as can be seen in the TYPE column. It is to note that the cluster doesn't have an external IP since it cannot be directly accessed and can only be accessed through the load balancer. The external IP of the load balancer is where the requests from the frontend are forwarded by setting the external IP to the proxy property in the package.json of the frontend.

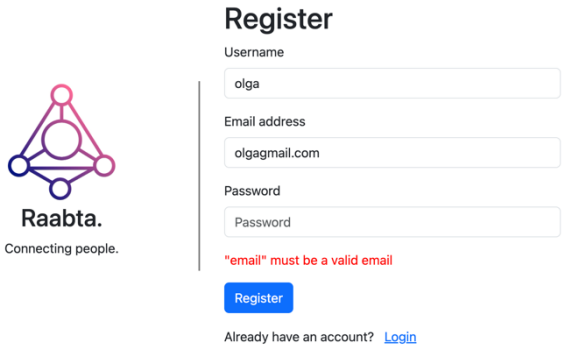
6. TESTING

TEST CASE 1: Olga, Nick, Mary, and Nestor register and are ready to access the Raabta API.

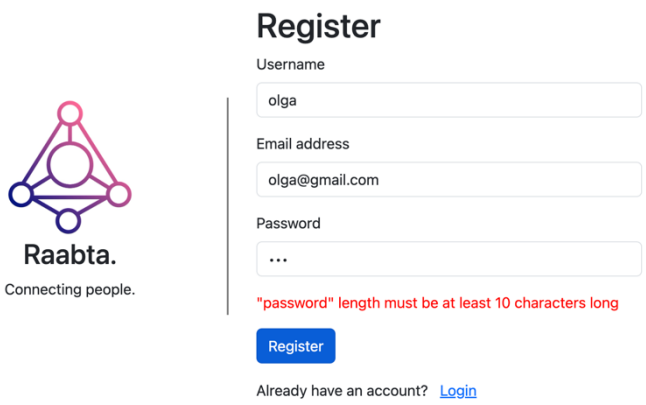
Input: Data is entered at the register form of the application frontend



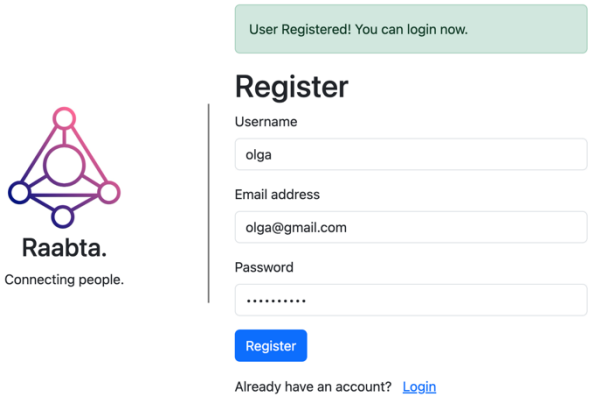
Checking Validation 1



Checking Validation 2



Checking Validation 3



When the user successfully registers

All four users are similarly registered and can be seen in the MongoDB Atlas, as shown below.

```
_id: ObjectId('65762de8db60e2eb6408a802')
username: "Mary"
email: "mary@gmail.com"
password: "$2a$05$4ZsmKpYnb5A2GTiCIHGHy.69Sucl3EqOCRsVL0eReiSP5cjGIDNDe"
date: 2023-12-10T21:30:16.008+00:00
__v: 0
```

```
_id: ObjectId('65762dfcdb60e2eb6408a805')
username: "Nestor "
email: "nestor@gmail.com"
password: "$2a$05$N88pT5sA5/FOMl2V3bqZZO8Ejwa.onHnAAkQUecDUWzDfjozUVAHa"
date: 2023-12-10T21:30:36.385+00:00
__v: 0
```

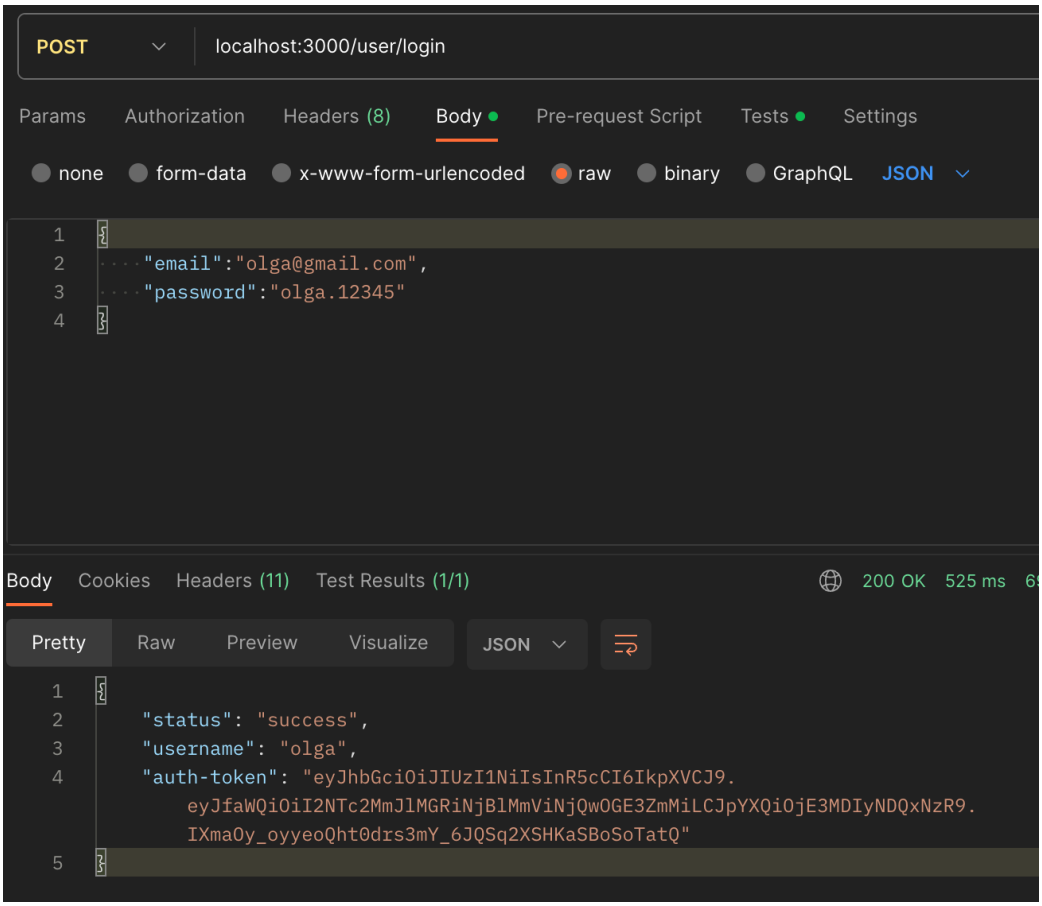
```
_id: ObjectId('65762be0db60e2eb6408a7fc')
username: "olga"
email: "olga@gmail.com"
password: "$2a$05$qKrr.fE2XFBbbp76uKuM.OoNggmF.3mdofzW8eABvMw3XQwYKZB4m"
date: 2023-12-10T21:21:36.656+00:00
__v: 0
```

```
_id: ObjectId('65762dd8db60e2eb6408a7ff')
username: "Nick"
email: "nick@gmail.com"
password: "$2a$05$TY2XwoeuiV/CcZzfZ1Jxf.7iN/jk8ErZjA7Gd2QHqFmOpChyK68nS"
date: 2023-12-10T21:30:00.139+00:00
__v: 0
```


Note that the passwords are hashed using Bcrypt.js before storing them into the database.

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

For this test case, one user, for instance Olga is shown as logging in and to show the token value in JSON, Postman request is shown.



In the frontend, when the user clicks the login button after entering correct data, the request is sent to this API endpoint, and with response, the auth-token and the username is extracted from the response and set to the app-wide state of the application to be accessed by any component.

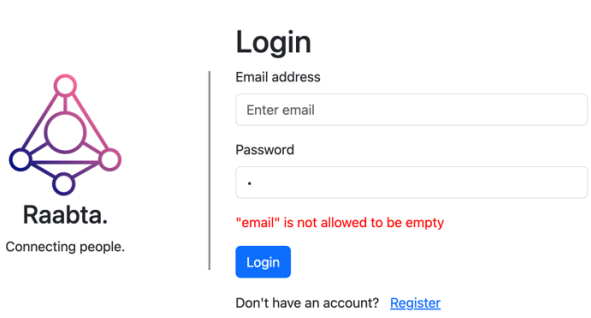


Figure 6a: Checking Validation 1

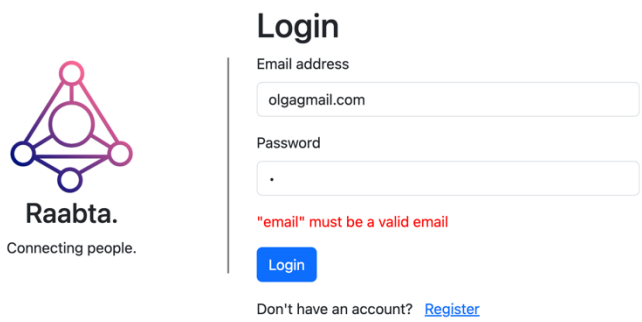


Figure 6b: Checking Validation 2

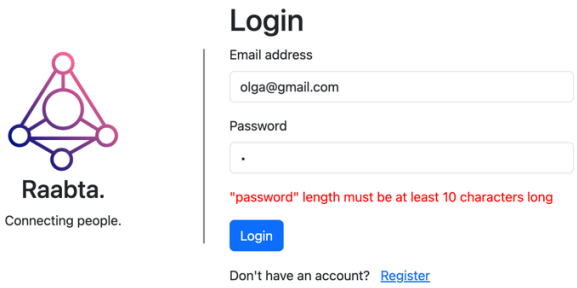


Figure 6c: Checking Validation 3

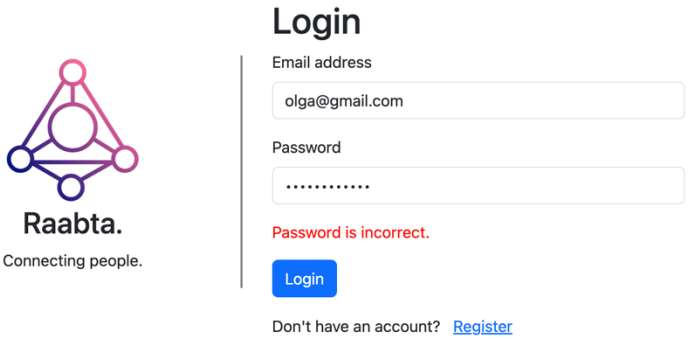


Figure 6d: Checking Validation 4

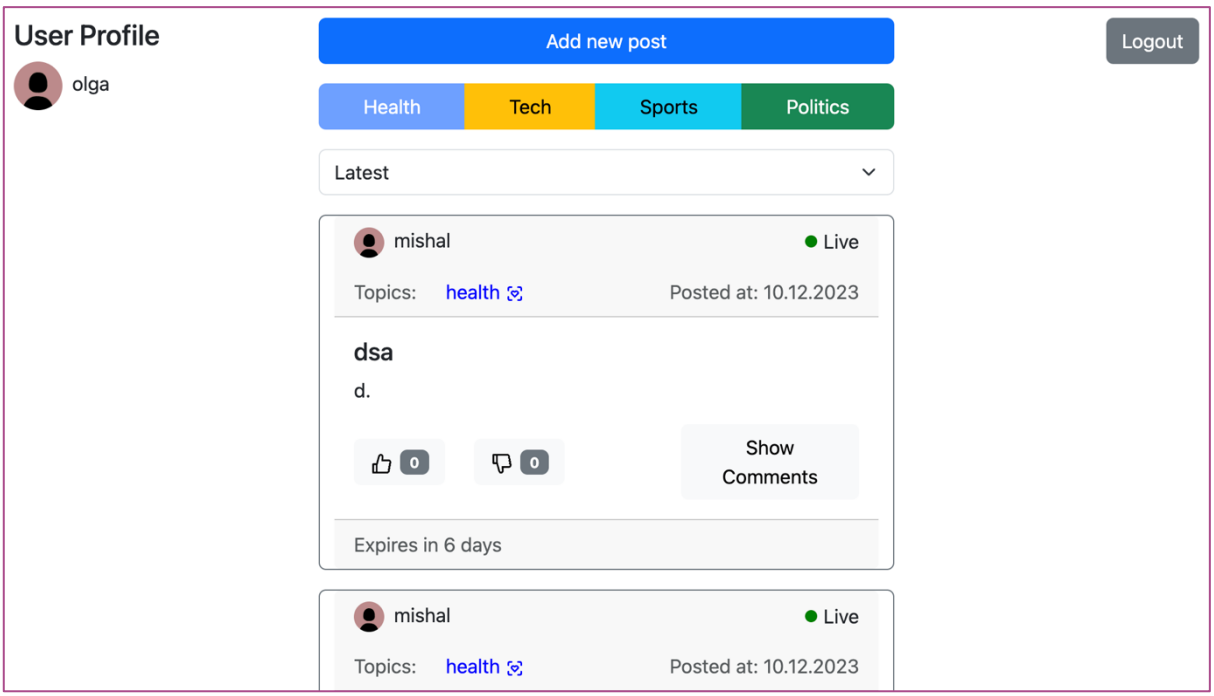
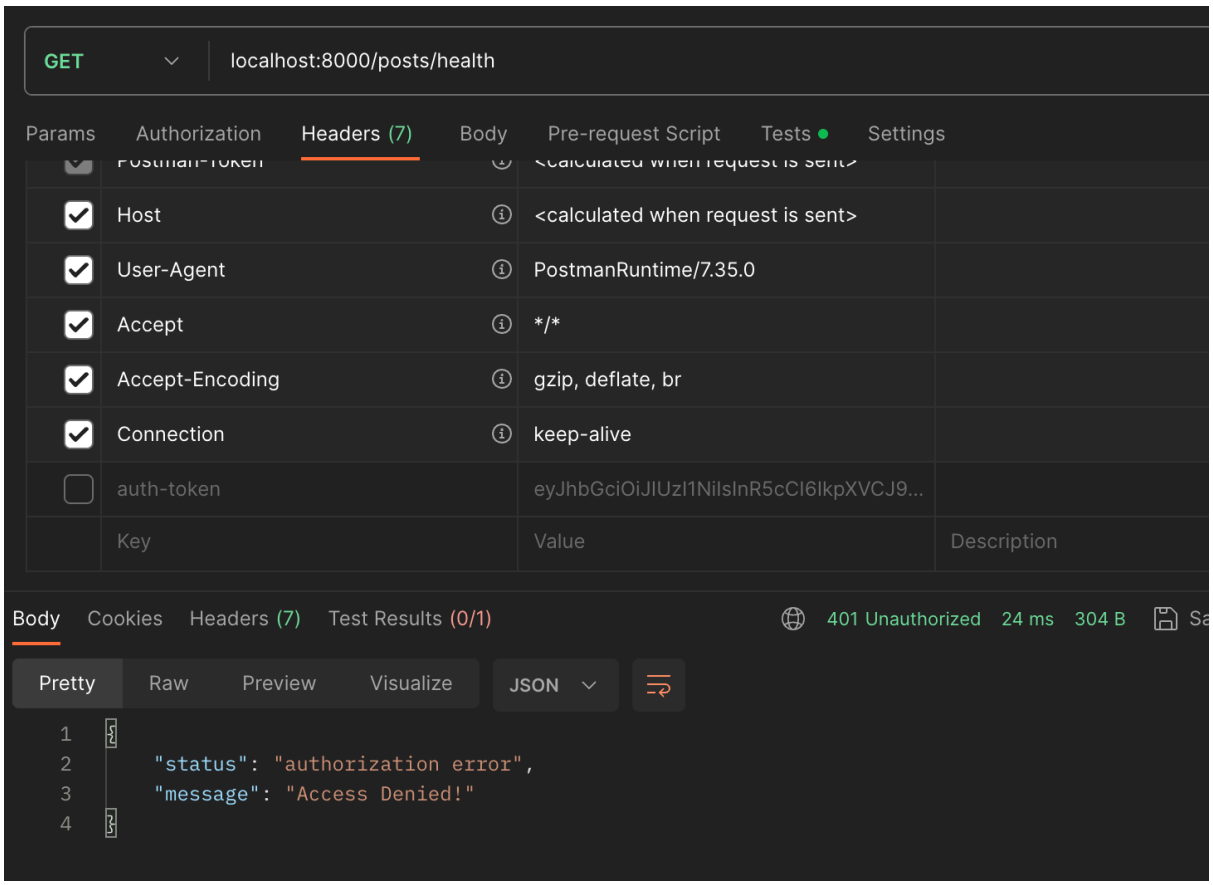


Figure 7: When the user successfully logs in.

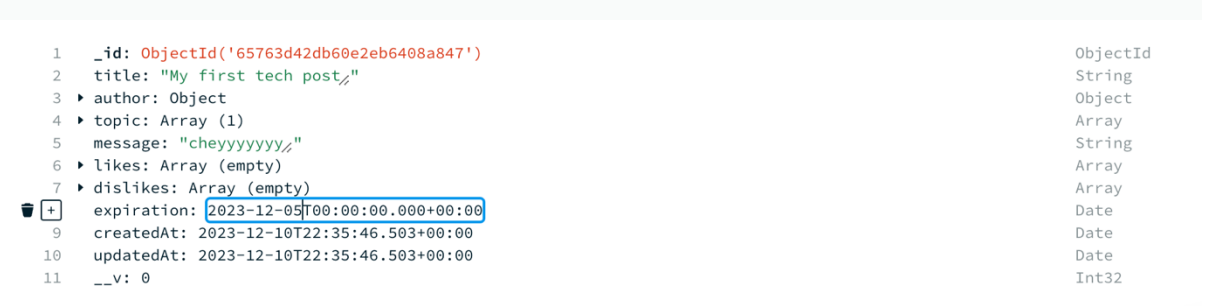
Test Case 3: Olga makes a call to the API without using her token. This call should be unsuccessful as the user is unauthorised.



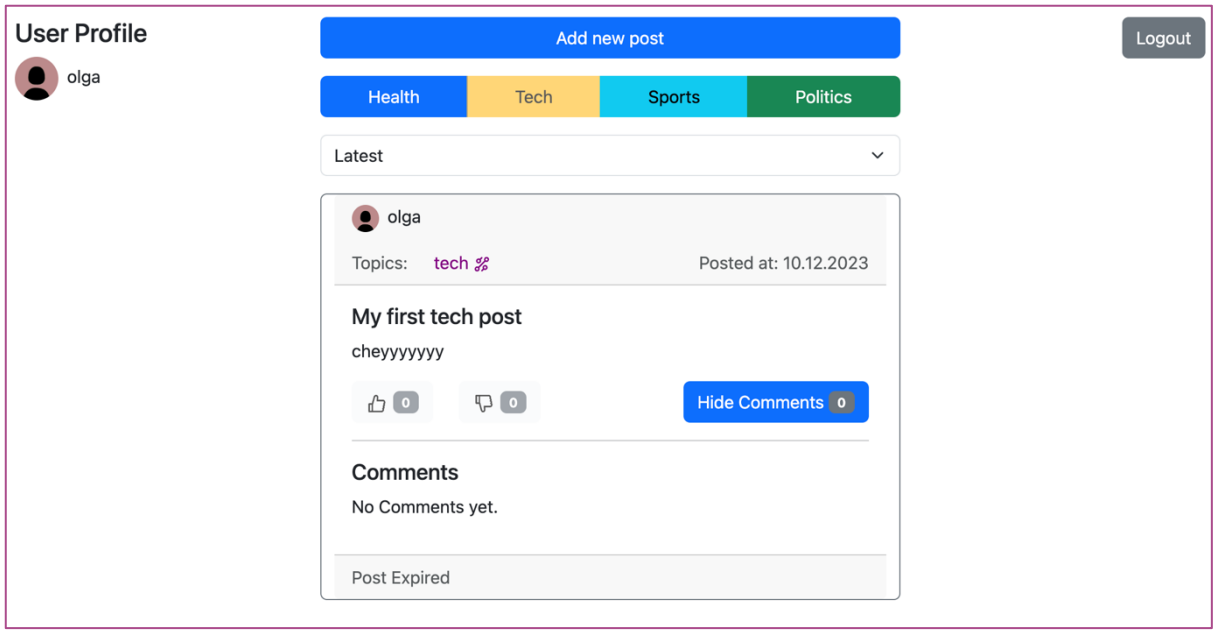
In frontend, when the user sends a request without a token, the user is directed to the login page.

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).

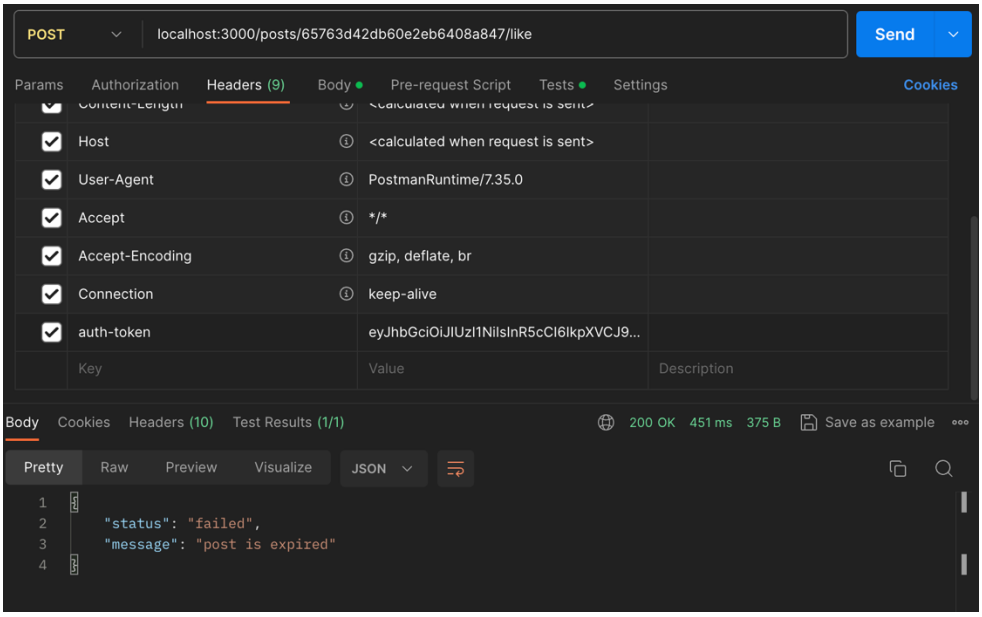
Raabta only takes date as the expiration date and does not consider time in minutes. To show this, a post will be entered by the MongoDB Atlas with an expiry date of a passed date (5th Dec 2023)



Olga added a post with 05-12-2023 as expiration date.

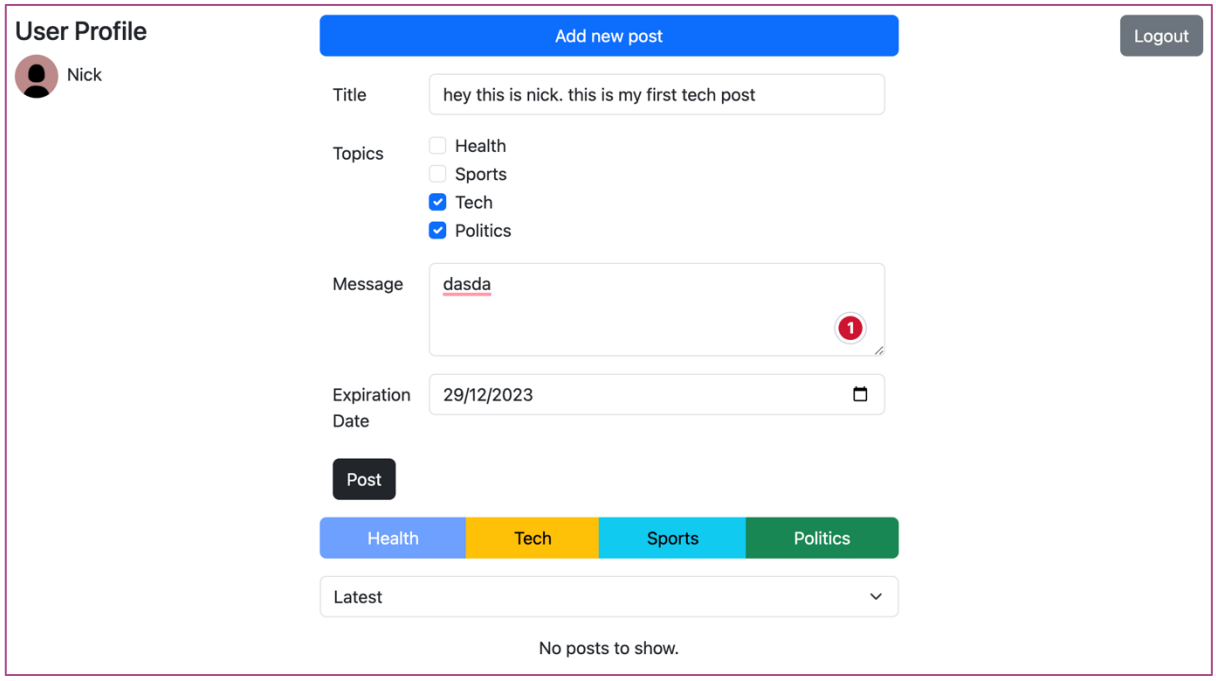


In the frontend, for an expired post all options to interact are set to disabled.

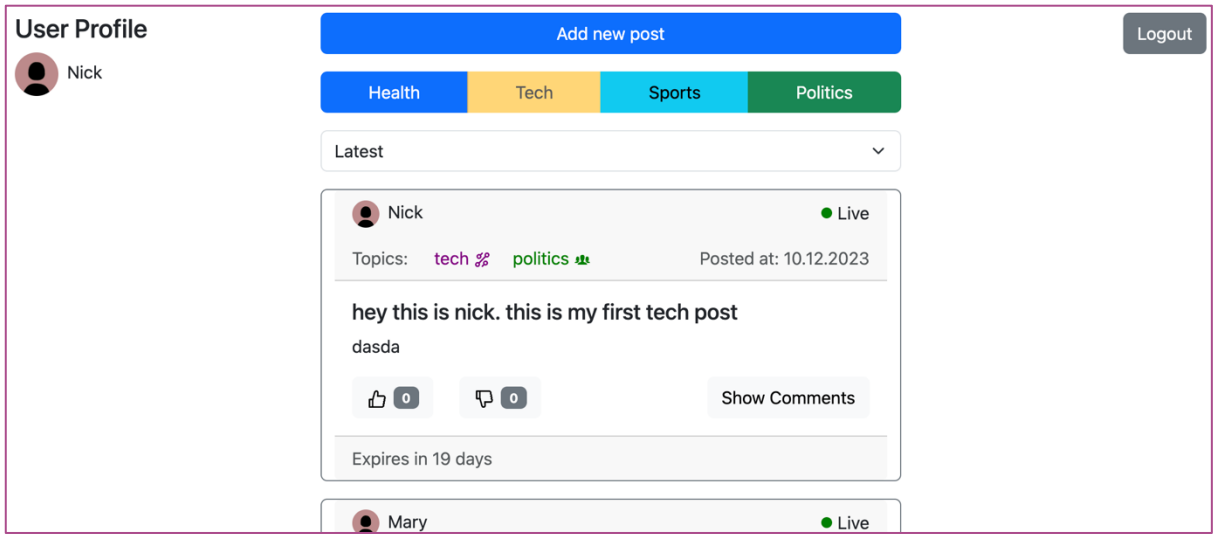


The request is sent to like the just created expired post.

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

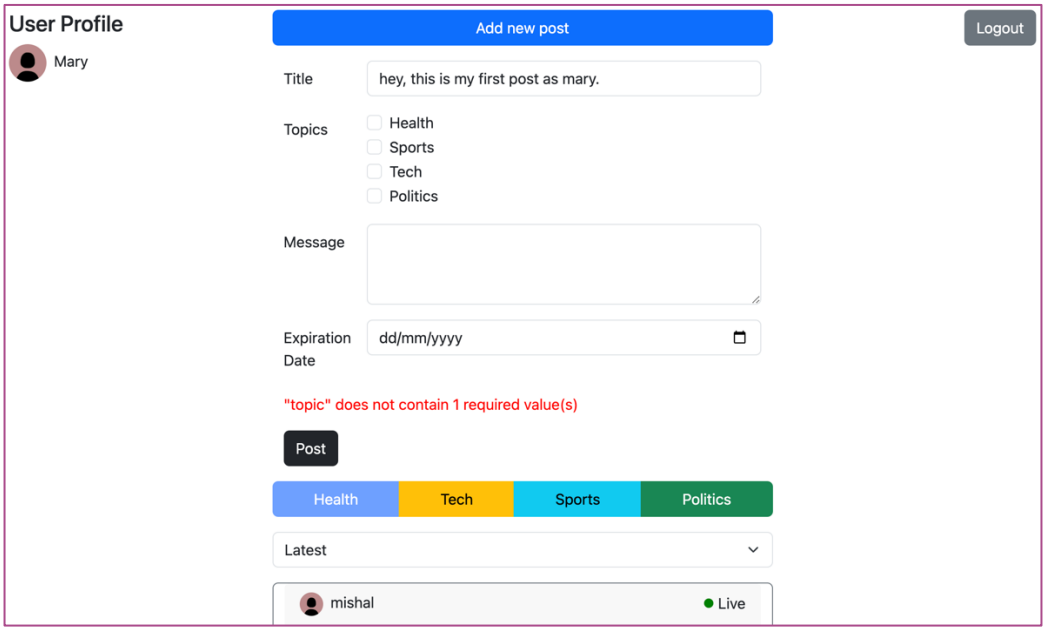


Nick enters his post data and clicks submit.

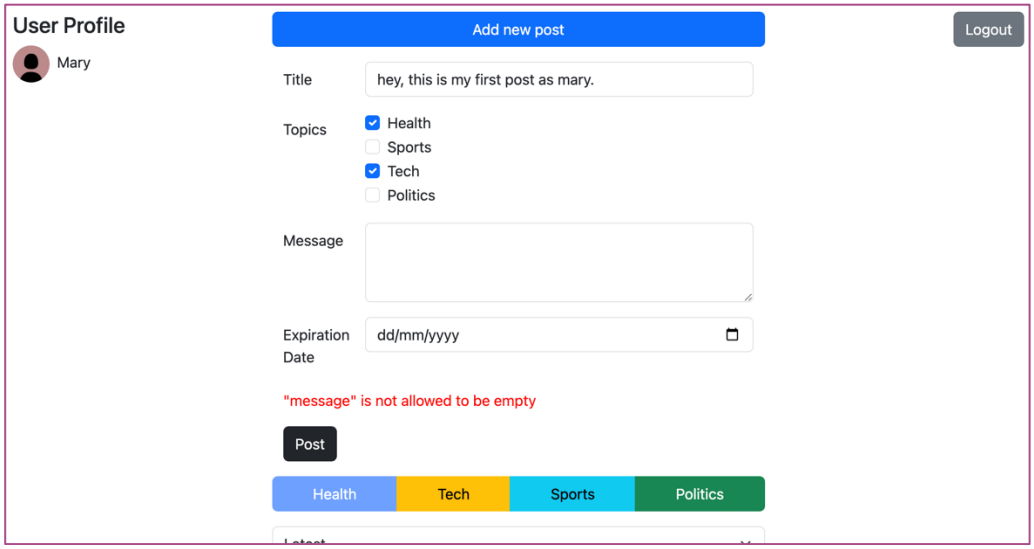


The post is uploaded and is shown on the wall.

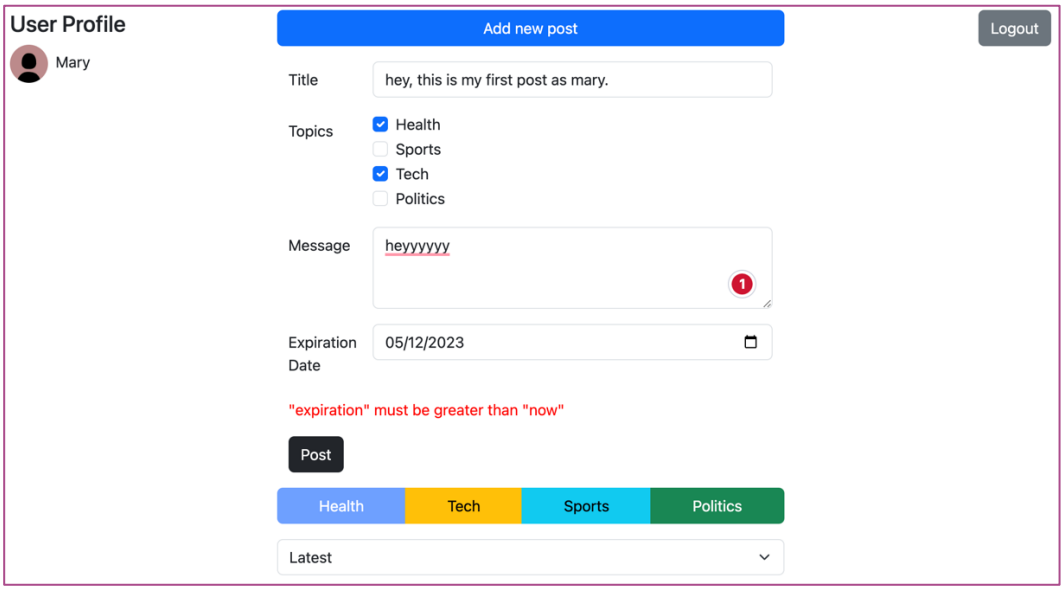
Test Case 5: Mary posts a message in the Tech topic with an expiration time using her token.



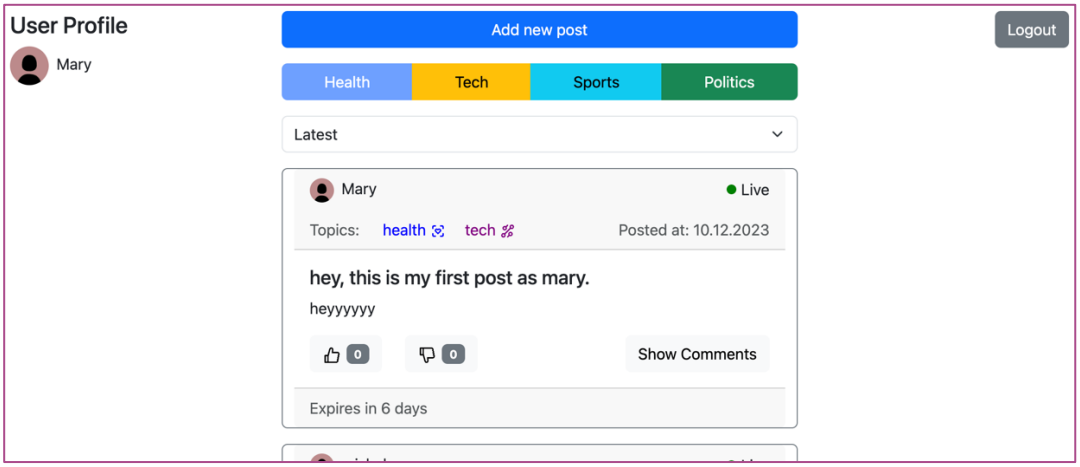
Checking Validations for posts



Checking Validations for posts

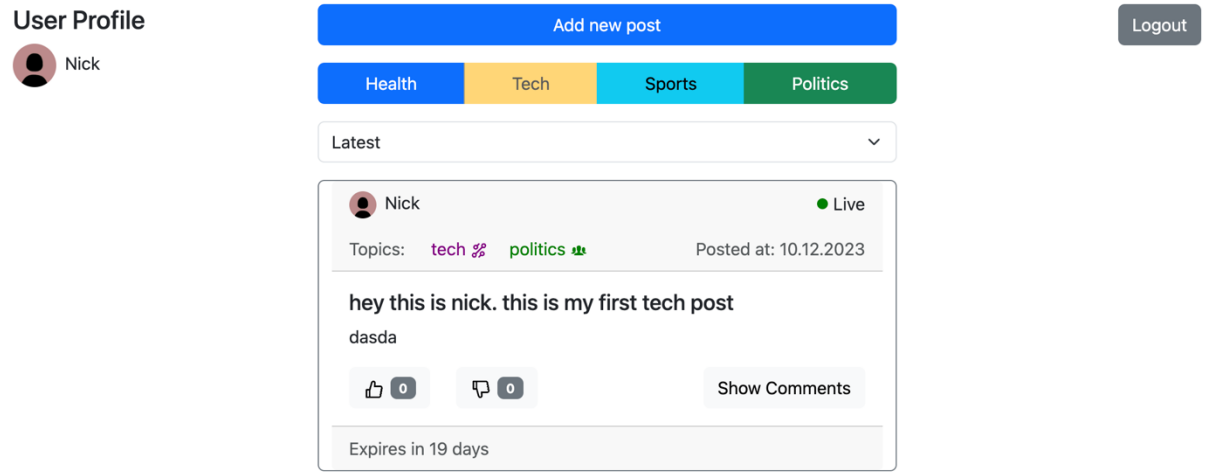


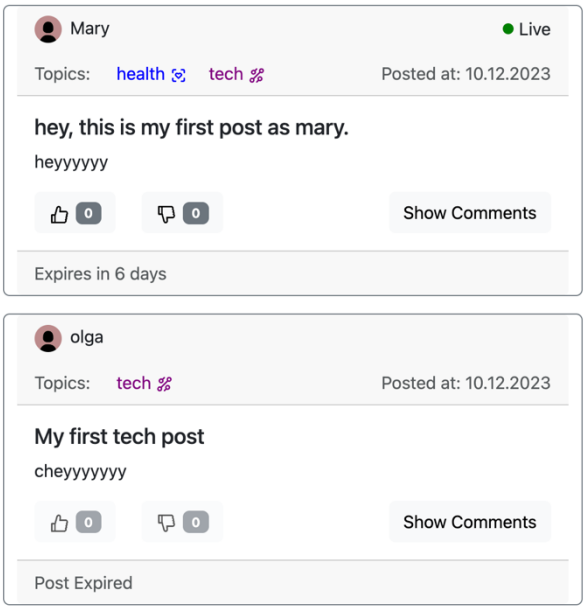
Checking Validations for posts



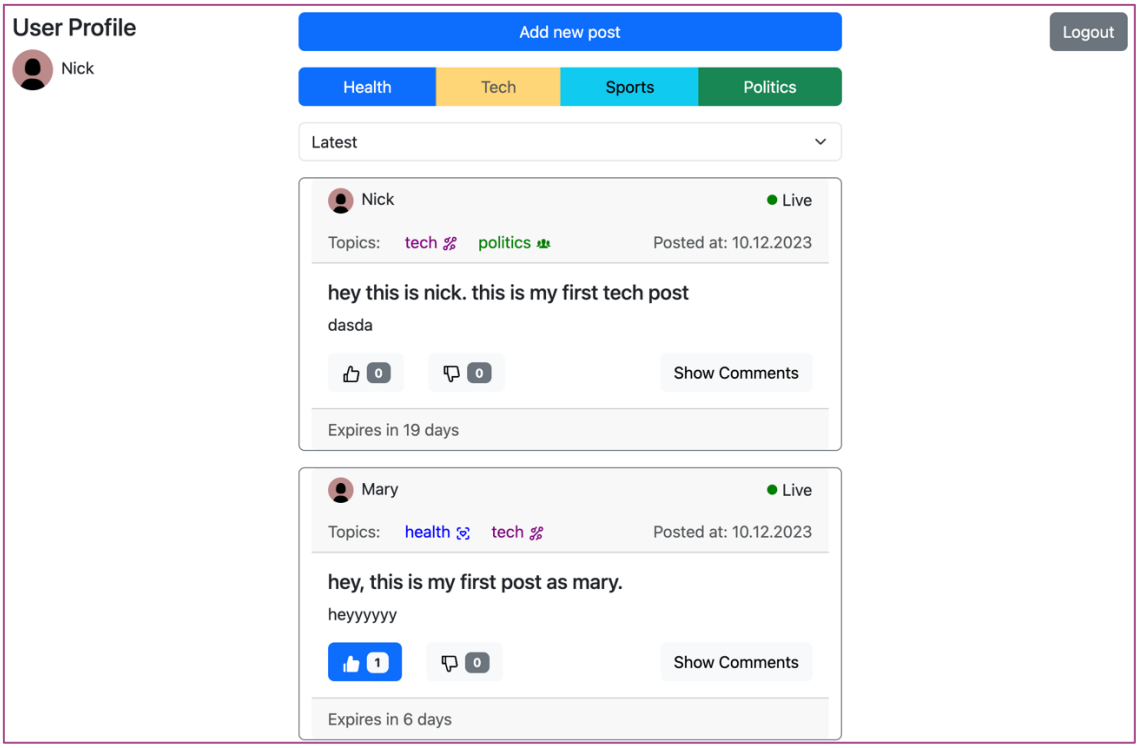
The post is uploaded when all the data is valid.

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.

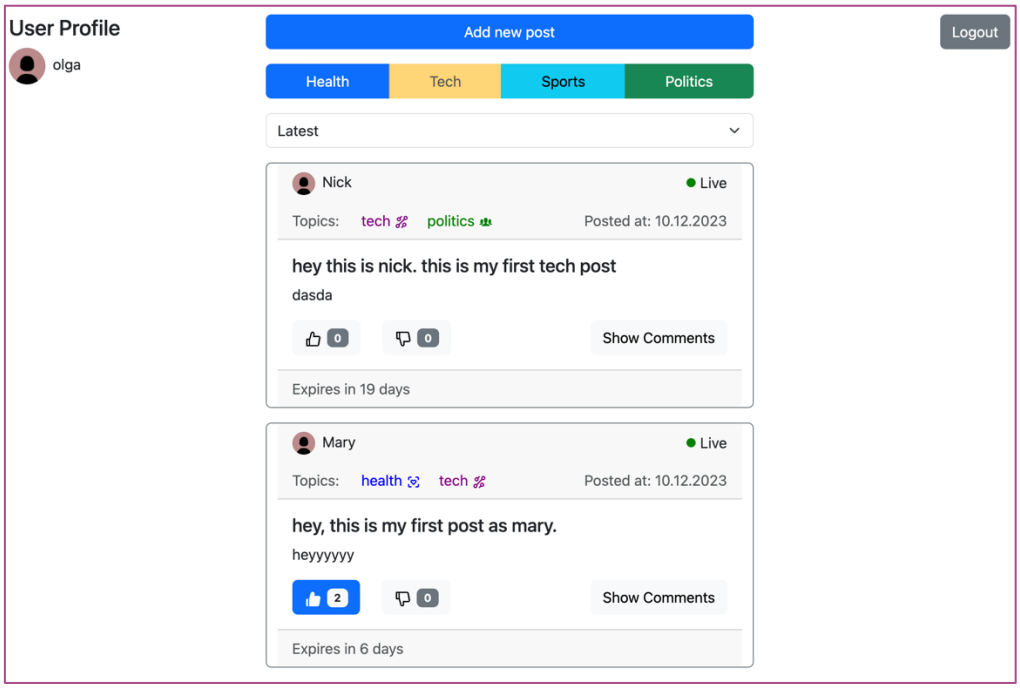




Test Case 8: Nick and Olga "like" Mary's post on the Tech topic.

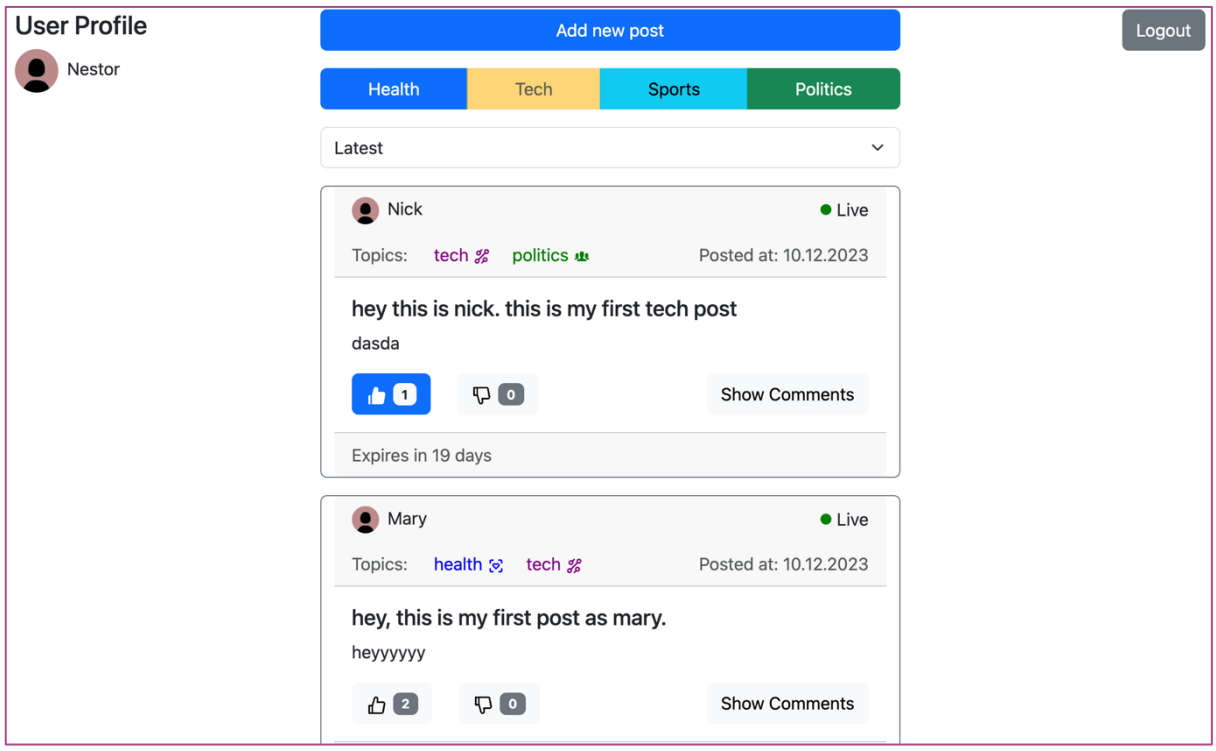


Nick Likes mary's post and can be seen by a filled like icon.

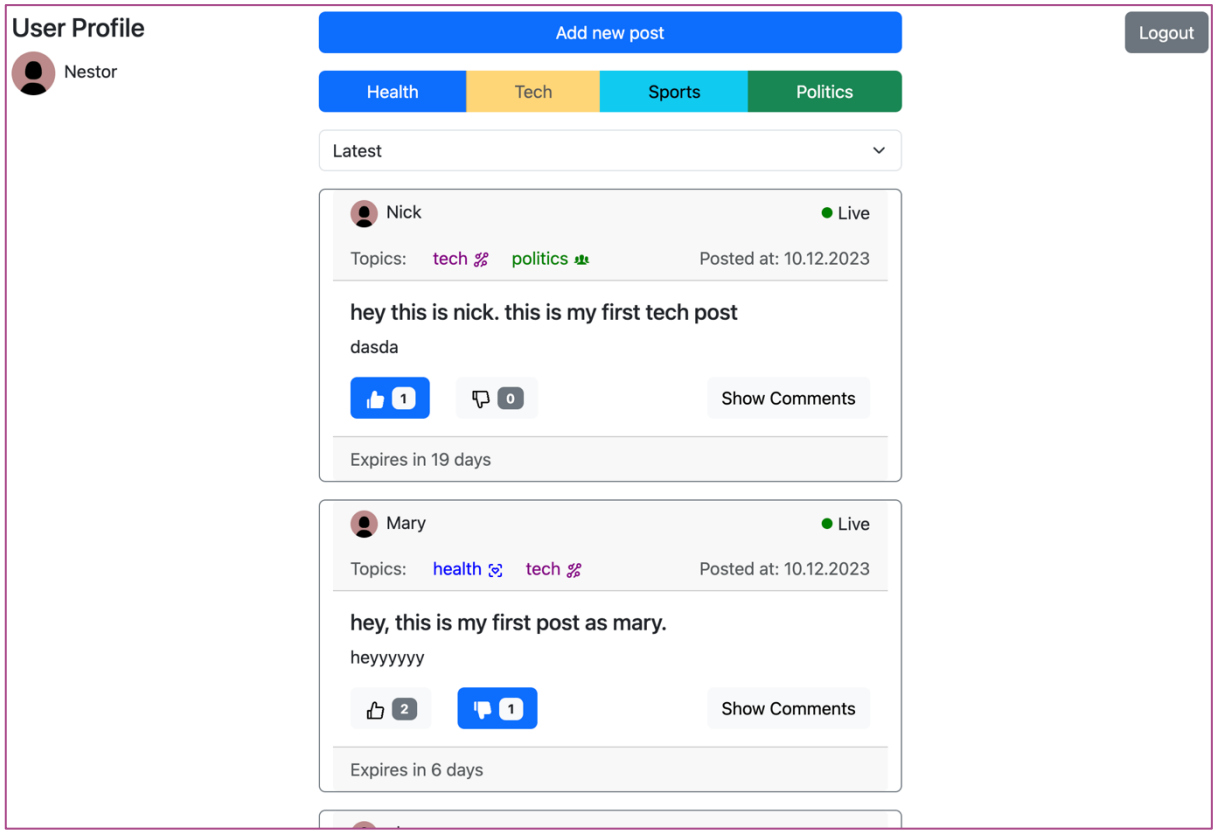


Olga Likes Mary's post and can be seen by a filled like icon.

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

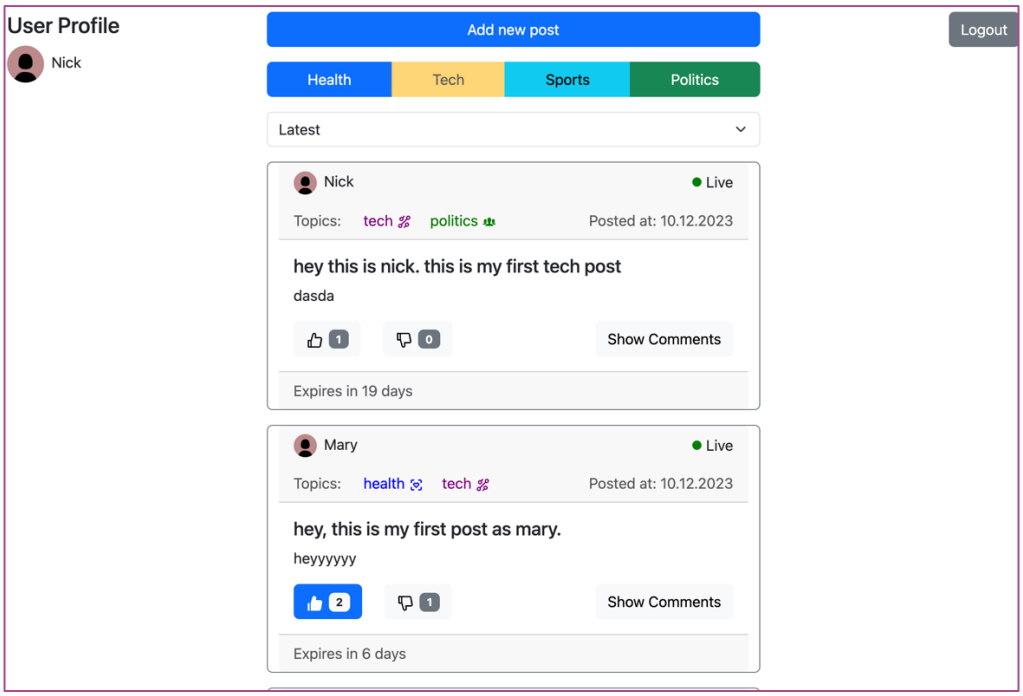


Nestor Likes Nick’s post and can be seen by a filled like icon.

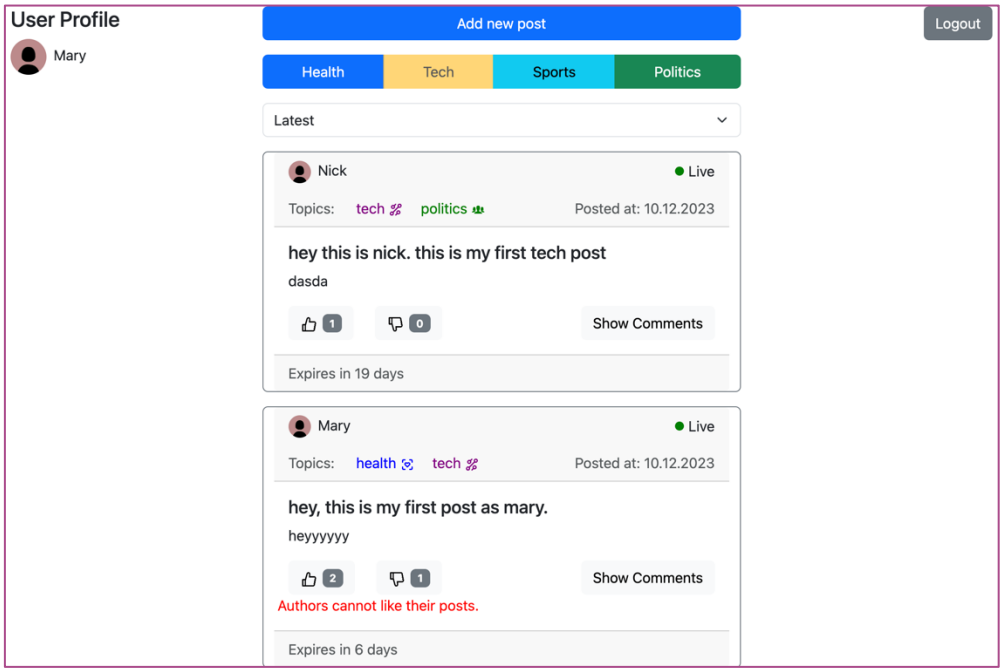


Nestor dislikes Mary’s post and can be seen by a filled dislike icon.

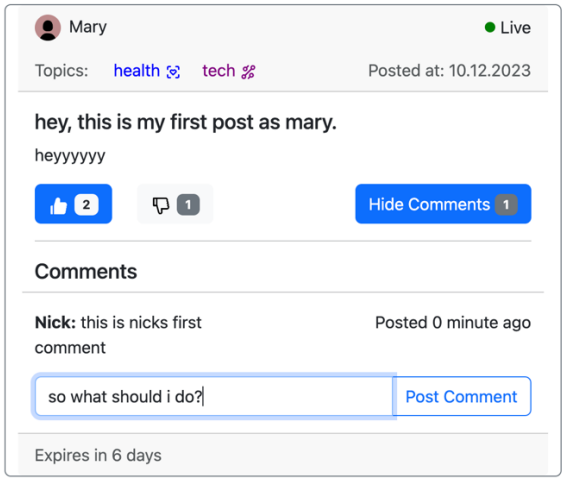
Test Case 10: Nick browses all the available posts on the Tech topic; at this stage, he can see the number of likes and dislikes for each post (Mary has two likes and one dislike, and Nick has one like). There are no comments made yet.



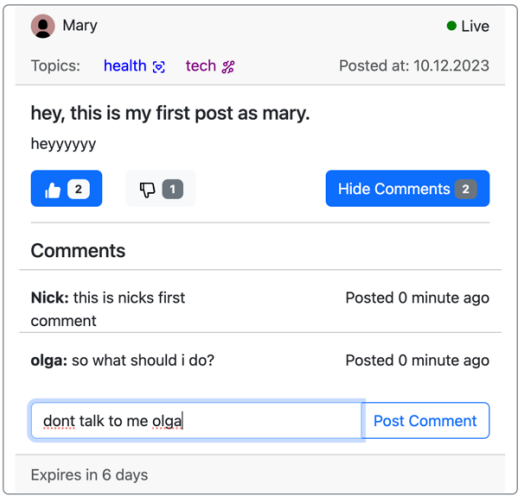
Test Case 11: Mary likes her post on the Tech topic. This call should be unsuccessful; in Piazza, a post owner cannot like their messages.



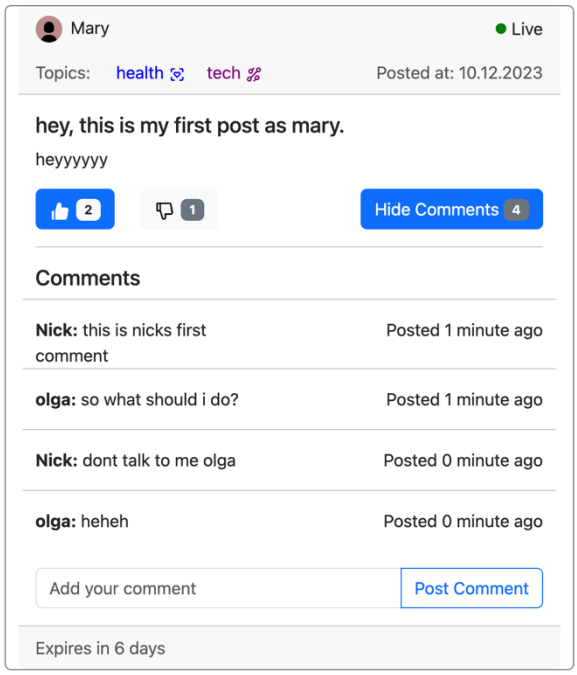
Test Case 12: Nick and Olga comment on Mary's post on the Tech topic in a round-robin fashion (one after the other, adding at least two comments each).



Olga adds his first comment.

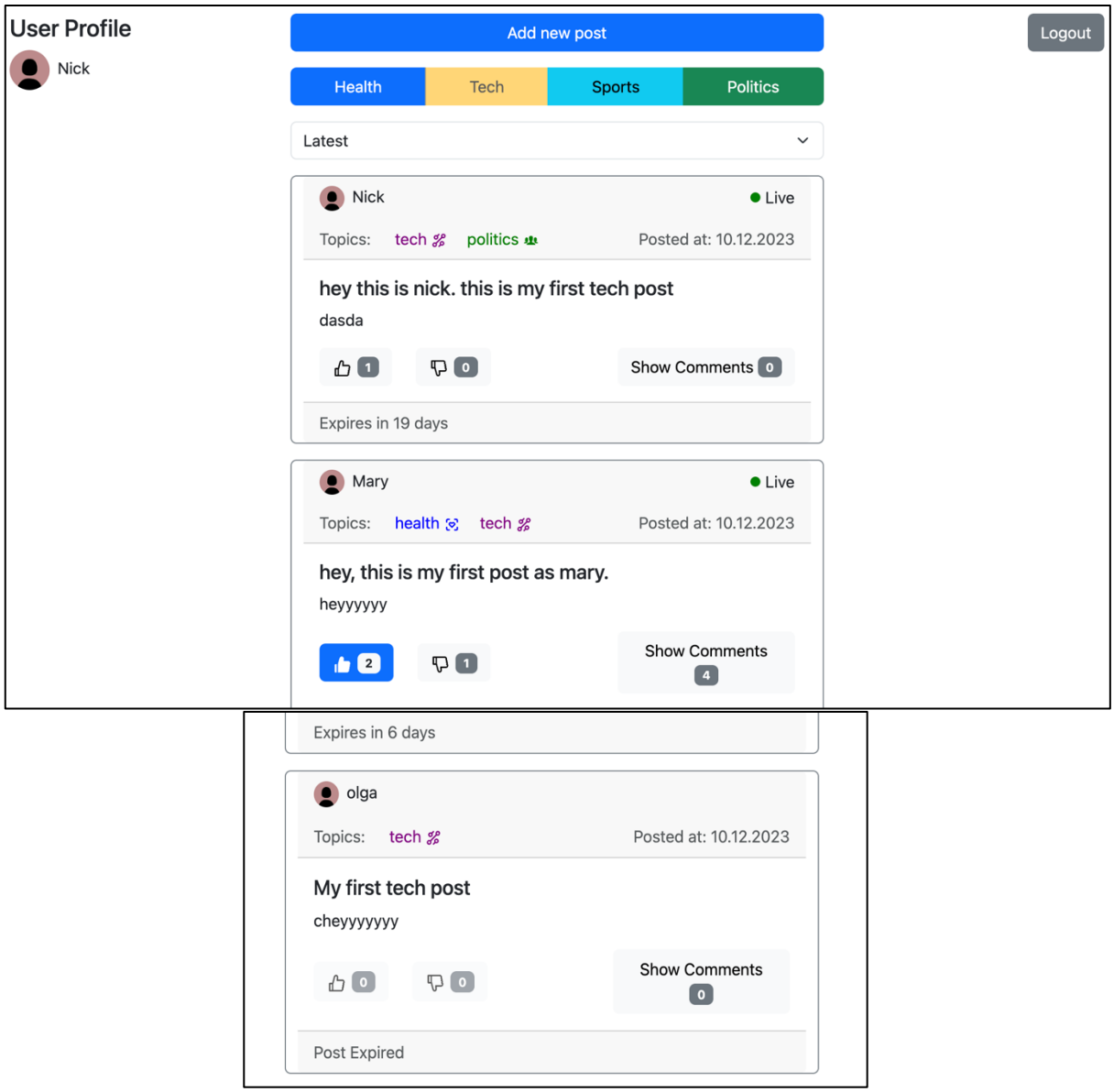


Nick adds his second comment.

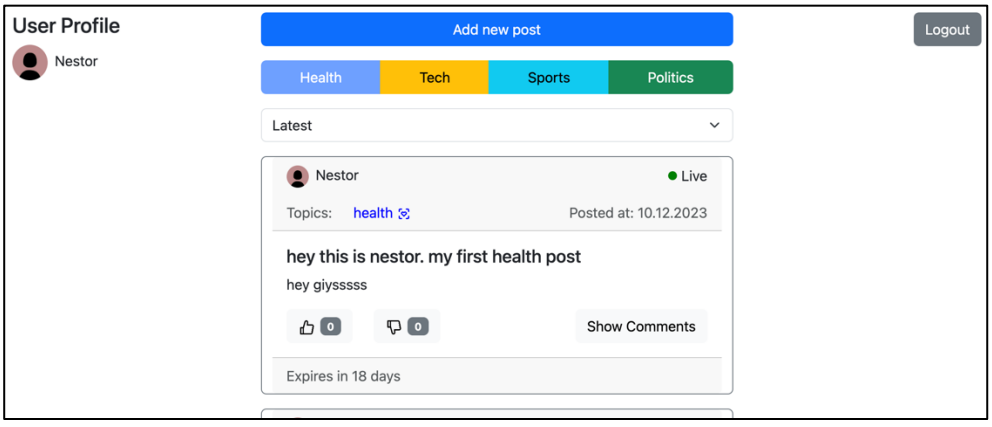


All the 4 comments.

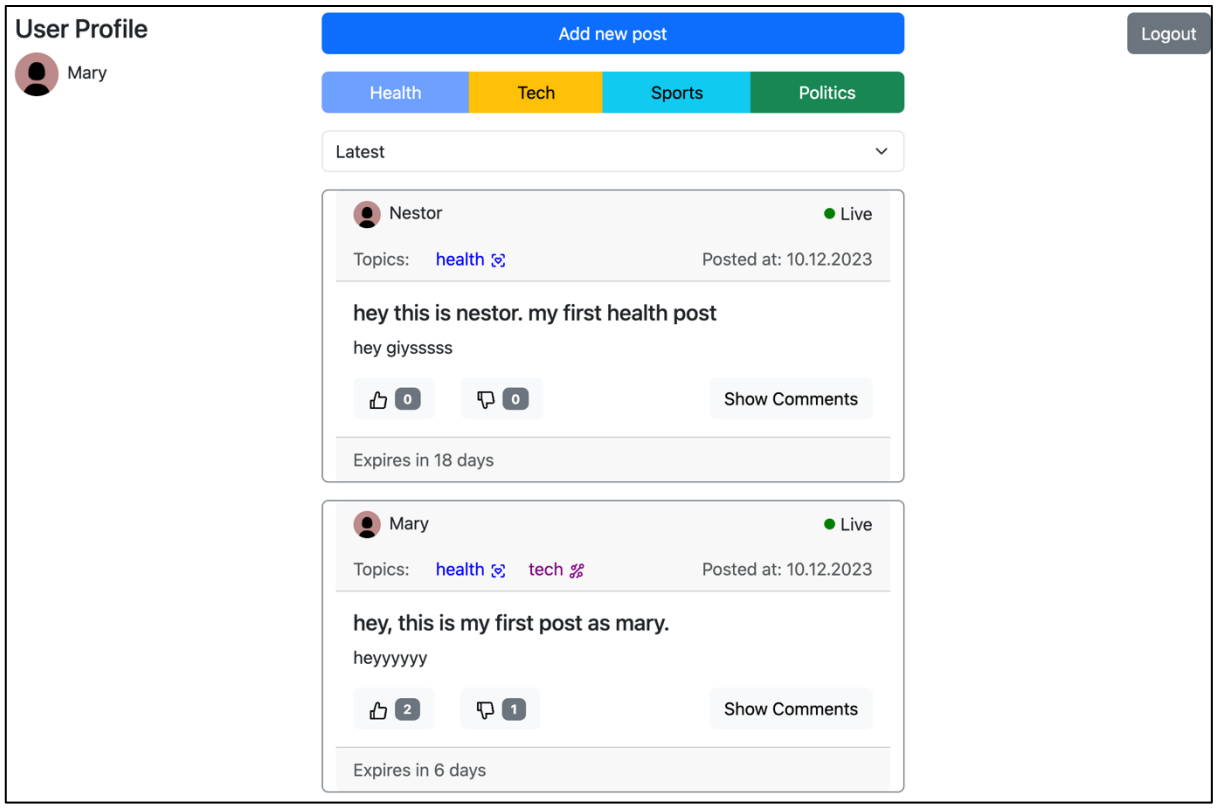
Test Case 13: Nick browses all the available posts in the Tech topic; at this stage, he can see the number of likes and dislikes of each post and the comments made.



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

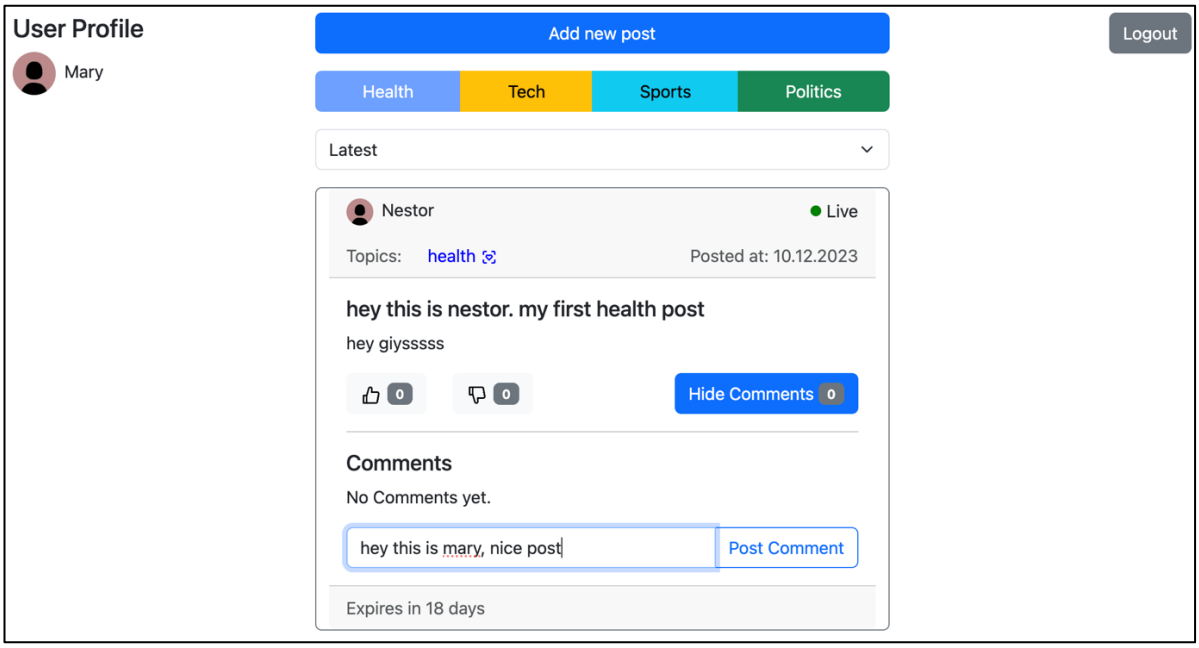


Test Case 15: Mary browses all the available posts on the Health topic; at this stage, she can see only Nestor's post.

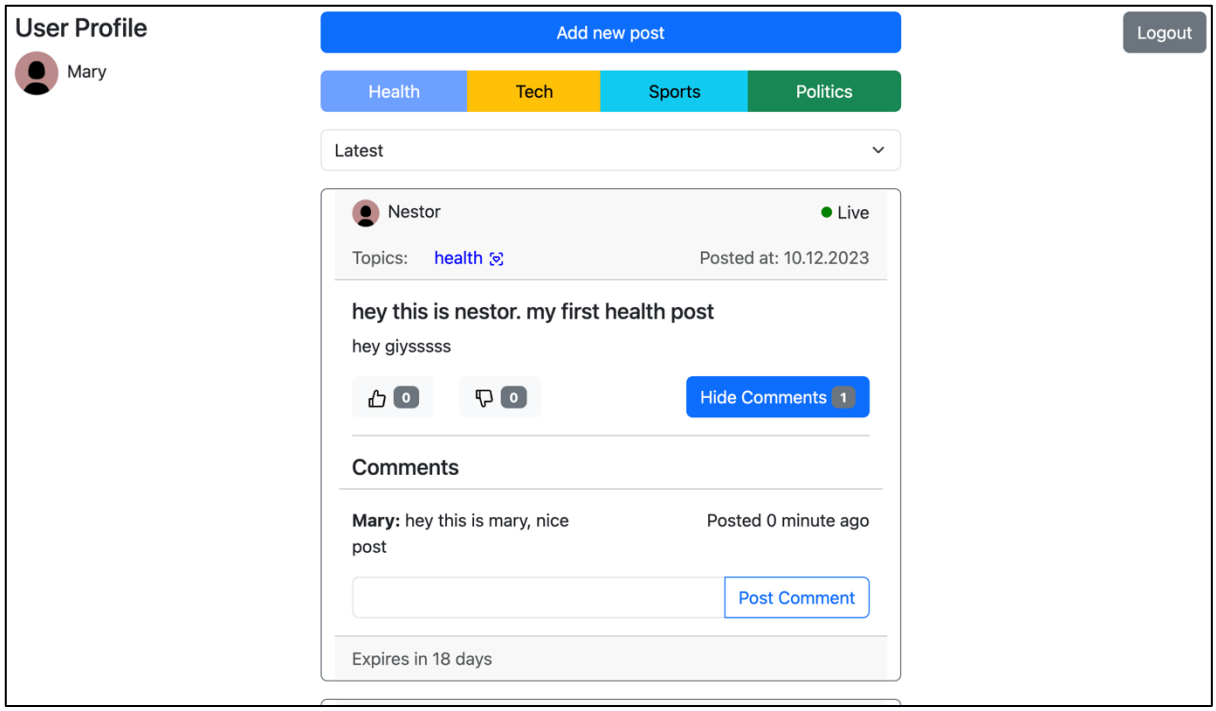


Mary can see two posts since in test case 5, Mary chose two topics to show that multiple topics can be selected.

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



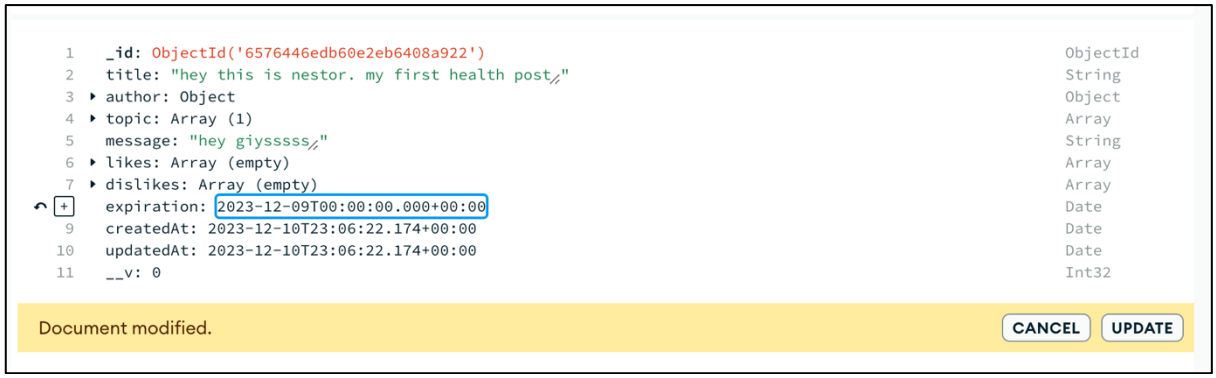
Mary typing comment of Nestor's health post.



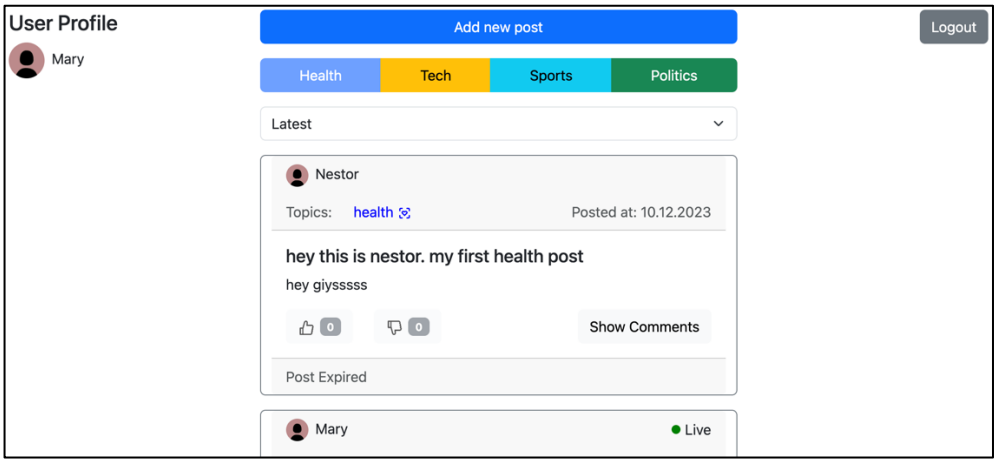
The comment is posted.

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

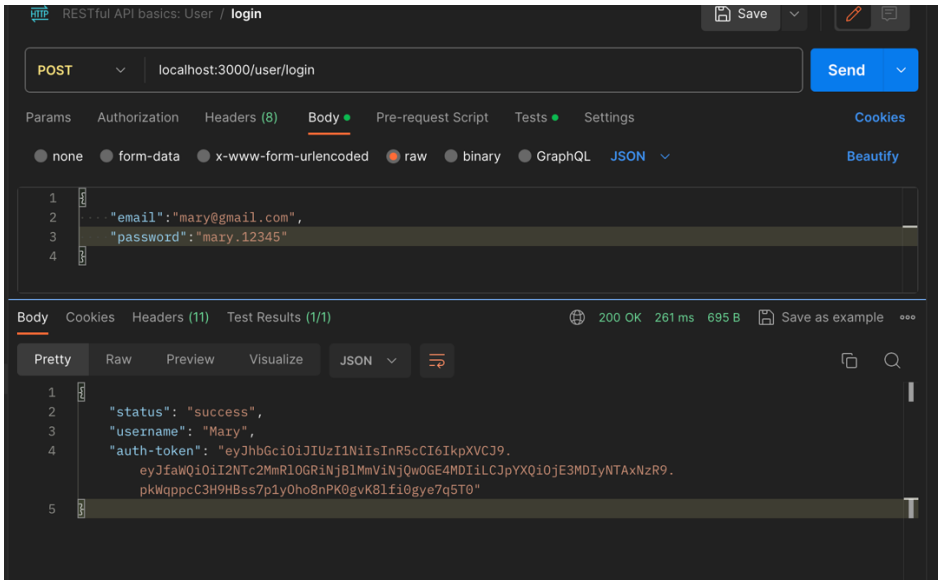
For this, MongoDB Atlas shall be used to change the expiry of nestor's message. Both frontend and postman request are attached to show the outcome of the request since in the frontend the option to like a post is disabled.



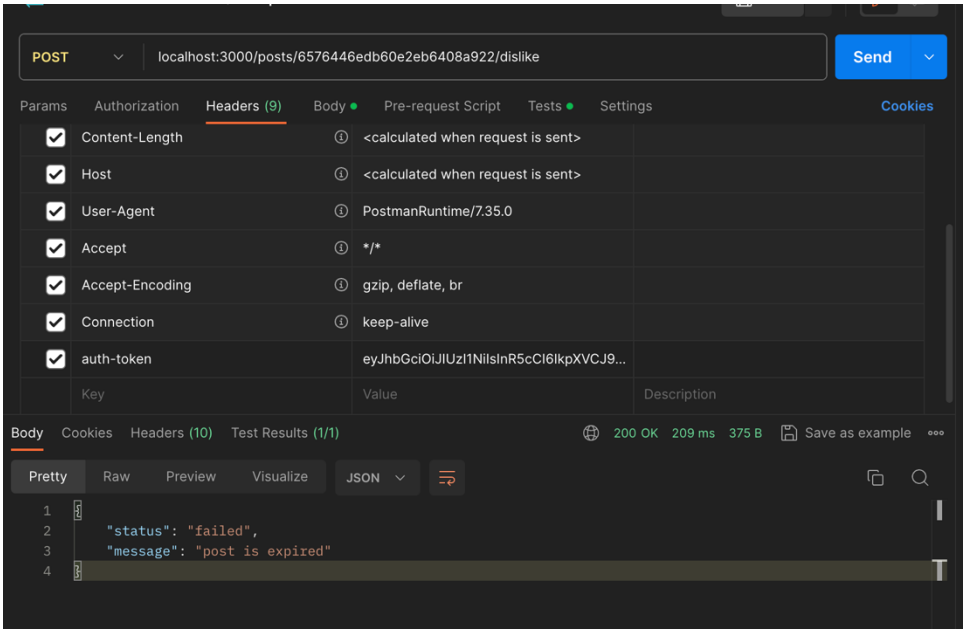
Changing the expiration date of Nestor's post to 9-12-23



The like and dislike buttons are disabled.

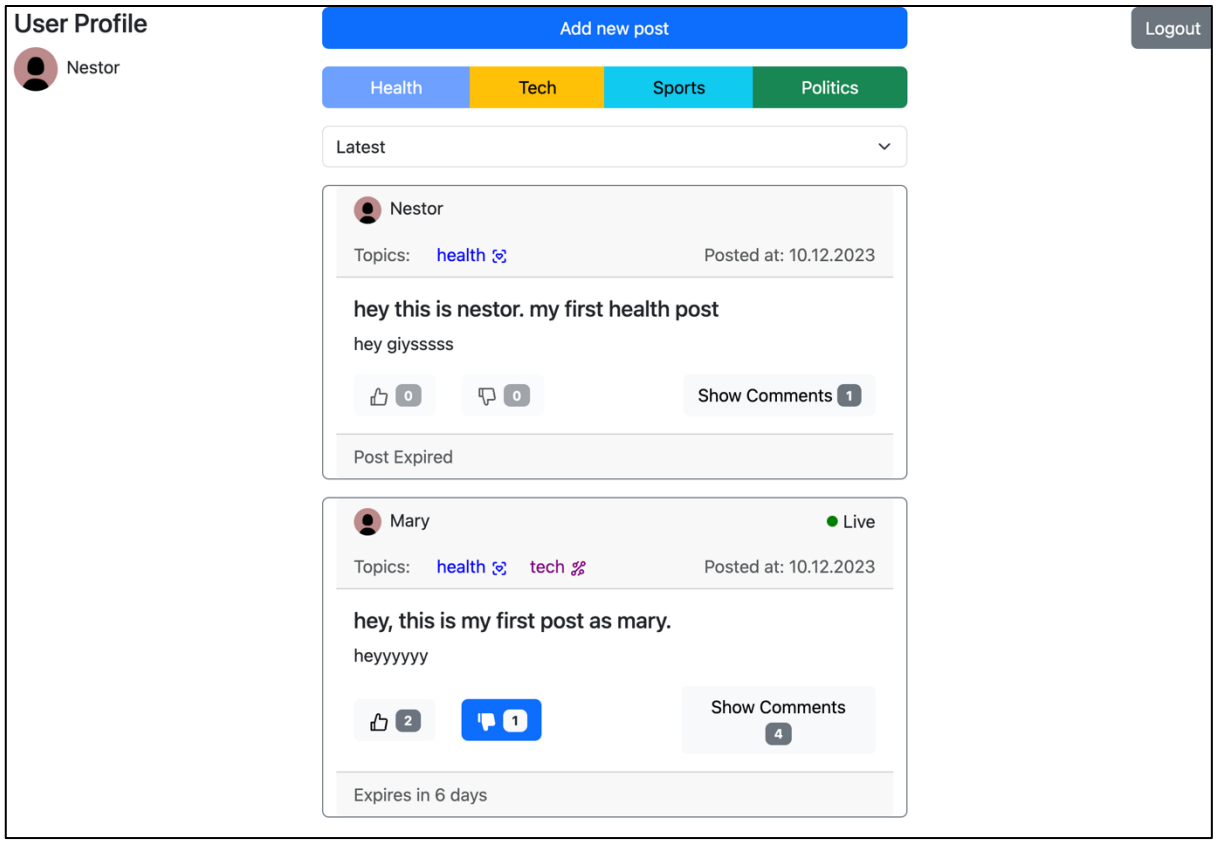


Logging in as Mary to get the auth-token.



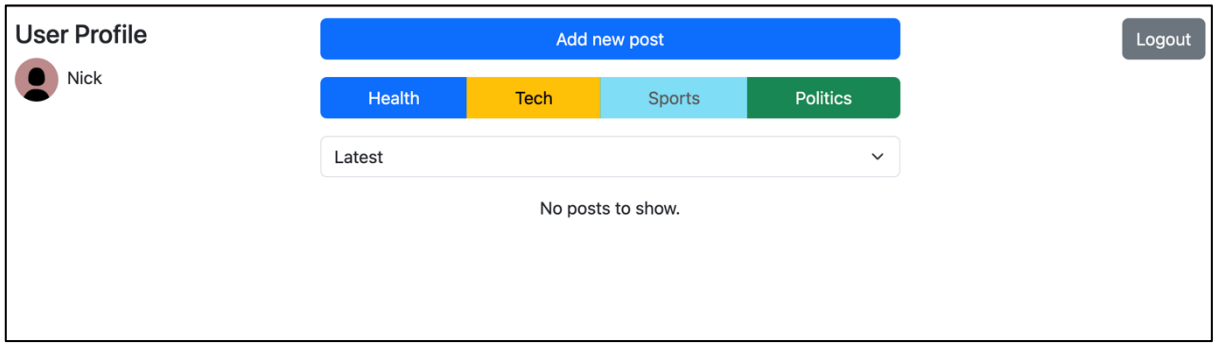
Shows the error that post is expired.

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).

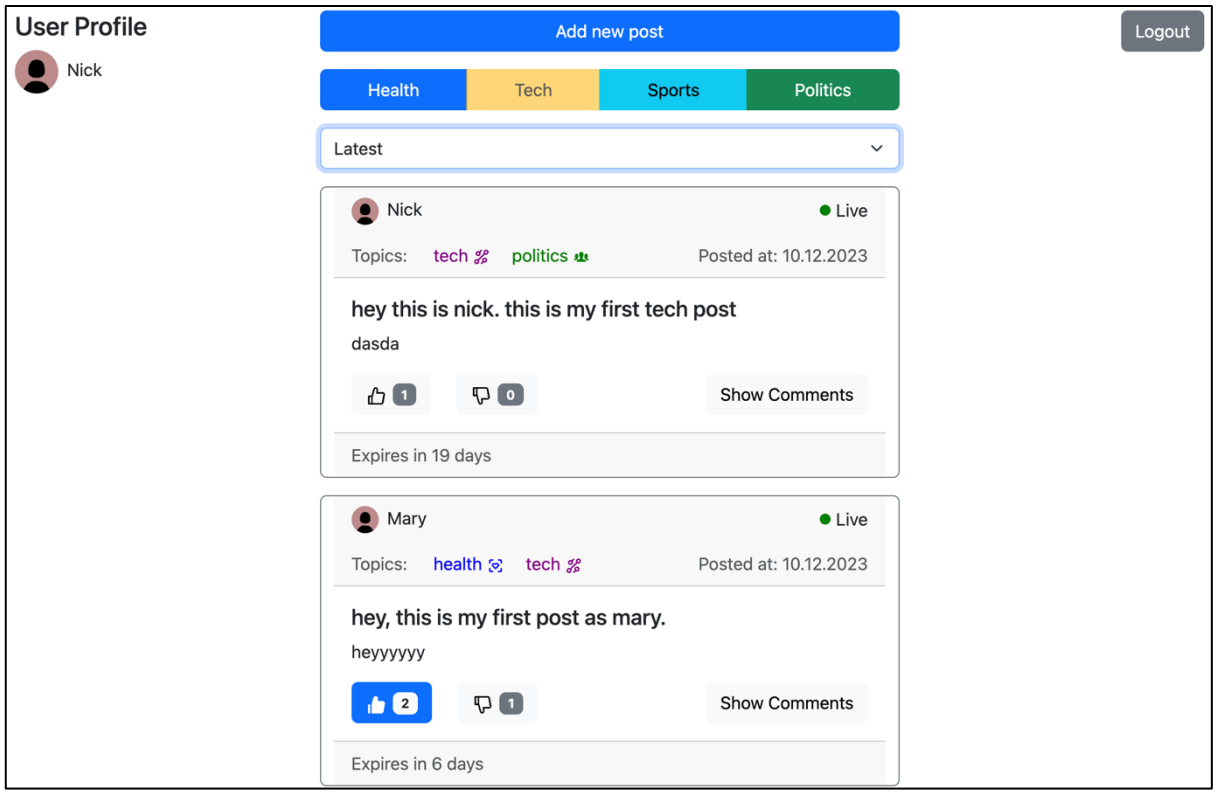


Mary can see two posts since in test case 5, Mary chose two topics to show that multiple topics can be selected.

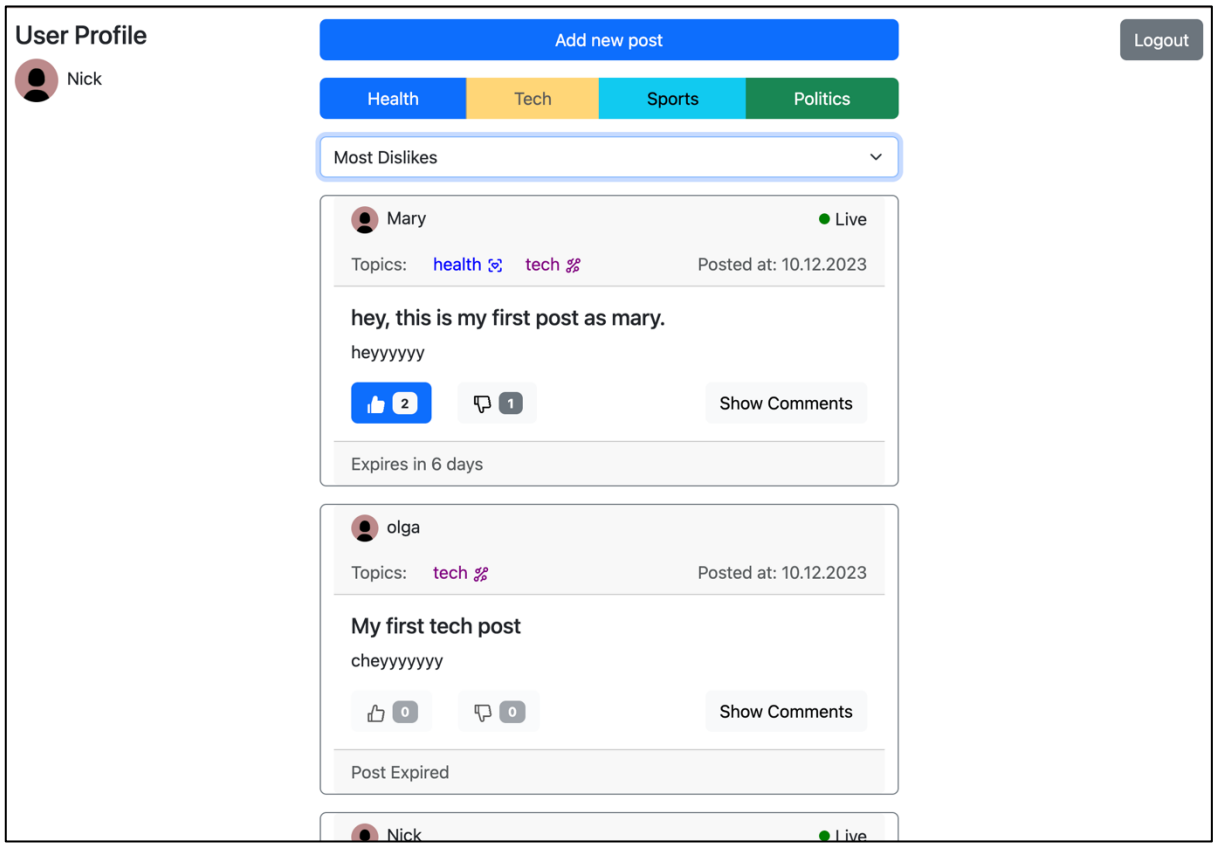
Test Case 19: Nick browses all the expired messages on the Sports topic. These should be empty.



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.



Sorted by latest.



Sorted by the maximum number of dislikes.

REFERENCES

Karlsson, J. (n.d.). *MongoDB Schema Design Best Practices*. Retrieved from MongoDB: https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

Logo Credits: "Designed by macrovector / Freepik"

APPENDICES

Appendix A

```
  {
    title: {
      type: String,
      required: true,
    },
    author: {
      id: { type: String, required: true },
      username: { type: String, required: true },
    },
    topic: [
      {
        type: String,
        lowercase: true,
        enum: ["politics", "health", "sports", "tech"],
        required: true,
      },
    ],
    message: {
      type: String,
      required: true,
    },
    likes: [String], //stores the ids in string, of the users that liked the post
    dislikes: [String], //stores the ids in string, of the users that disliked the
post
    expiration: {
      type: Date,
      required: true,
    },
  },
  { timestamps: true }
```

Appendix B

```
  {
    post: { type: mongoose.Types.ObjectId, ref: "Post" },
    author: {
      type: String,
      required: true,
    },
    body: {
      type: String,
      required: true,
    },
  },
  { timestamps: true }
```

Appendix C

```
const schemaValidation = joi.object({
  username: joi.string().required().min(3).max(256),
  email: joi.string().required().email(),
  password: joi.string().required().min(10).max(1024),
  });
```

Appendix D

```
const schemaValidation = joi.object({
  email: joi.string().required().email(),
  password: joi.string().required().min(10).max(1024),
});
```

Appendix E

```
  const postSchema = joi.object({
    title: joi.string().required(),
    topic: joi.array()
      .items(joi.string().valid("politics", "health", "sports", "tech").required())
      .required(), //topic is an array of strings, where each string can only be
from the options: ["politics", "health", "sports", "tech"]
    message: joi.string().required().trim().min(1), //the message cannot be empty
or contain only spaces
    author: joi.string().required(),
    expiration: joi.date().greater("now").required(), //the expiration date must be
greater than the current date
  });
```

Appendix F

```
const commentSchema = joi.object({
    body: joi.string().trim().required().min(1), //the body cannot be empty and can
be as long as required
    author: joi.string().required(), //the username of the author is required
  });
```