

## Table of Contents

Phase A: Set up your MiniWall project and install necessary libraries .....	2
NodeJS dependencies .....	2
File Structure .....	2
URL Endpoints .....	2
Phase B: Enforcing authentication/verification functionalities.....	3
Phase C: Development of the MiniWall RESTful API.....	4
Phase D: Development of the MiniWall testing cases .....	7
TC 1: Olga, Nick, and Mary register in the application and access the API.....	7
TC 2: Olga, Nick, and Mary will use the OAuth v2 authorization service to get their tokens.....	10
TC 3: Olga calls the API (any endpoint) without using a token. This call should be unsuccessful as the user is unauthorized.....	13
TC 4: Olga posts a text using her token .....	14
TC 5: Nick posts a text using his token .....	16
TC 6: Mary posts a text using her token.....	17
TC 7: Nick and Olga browse available posts in reverse chronological order in the MiniWall; there should be three posts available. Note: that we do not have any likes yet .....	18
TC 8: Nick and Olga comment Mary's post in a round-robin fashion (one after the other) .....	20
TC 9: Mary comments her post. This call should be unsuccessful; an owner cannot comment owned posts.....	21
TC 10: Mary can see posts in reverse chronological order (newest posts are on the top as there are no likes yet) .....	22
TC 11: Mary can see the comments for her posts .....	23
TC 12: Nick and Olga like Mary's posts.....	23
TC 13: Mary likes her post. This call should be unsuccessful; an owner cannot like their post.....	26
TC 14: Mary can see that there are two likes in her posts.....	27
TC 15: Nick can see the list of posts, since Mary's post has two likes it is shown at the top.....	28
TC 16: Extra functionality .....	29
Phase E: Deploy your MiniWall project into a GCP VM using Docker.....	31
Phase F: Document your MiniWall solution in a technical report .....	34
Phase G: Submit quality scripts.....	34

## Phase A: Set up your MiniWall project and install necessary libraries

### NodeJS dependencies

- bcryptjs: Used to encrypt/decrypt user passwords
- body-parser: Used to parse JSON to display on the response
- dotenv: Used to read .env file which stores sensitive information
- express: Used to create the API
- joi: Used to validate POST requests
- jsonwebtoken: Used to create a token to verify user login
- mongoose: Used to connect to MongoDB
- nodemon: Used to restart the serve automatically

### File Structure

- Repo folder
  - models -- directory that holds DB structures
    - Posts.js -- posts schema
    - Users.js -- users schema
  - node\_modules -- NodeJS packages
    - all package dependencies here
  - routes -- directory that holds URIs
    - auth.js -- login / register routes
    - posts.js -- all routes that involve the posts
  - validations -- directory that holds authentications/verifications
    - validation.js -- checks for correct inputs
    - verifyToken.js -- checks for web token
  - .env -- secret file for sensitive information
  - .gitignore -- file to have GitHub ignore files to sync with
  - app.js -- main entry point of the app
  - package.json -- NodeJS default
  - package-lock.json -- NodeJS default

### URL Endpoints

There are two main endpoints being used for the specific schemas: user and posts. These are the first two after <http://localhost:3000>

```
app.use('/posts', postsRoute)
app.use('/user', authRoute)
```

Next, there are two final endpoints for the user section. Those two endpoints are register and login.

```
router.post(path: '/login')
```

```
router.post( path: '/register'
```

As for the posts section, this one is a bit more involved since it follows the CRUD structure. Posts/ has both a POST and GET, which is for all of the posts on the MiniWall app.

```
router.post( path: '/'
```

```
router.get( path: '/'
```

The following endpoints deal with individual posts.

```
router.get( path: '/:postId'
```

```
router.patch( path: '/:postId'
```

```
router.delete( path: '/:postId'
```

In regard to likes, I had decided to go with a PATCH of the postId where it adds the username to a list of likes. There is also a PATCH to dislike the post.

```
router.patch( path: '/:postId/like'
```

```
router.patch( path: '/:postId/unlike'
```

In regard to comments, this one was a bit harder to think about implementing due to inexperience with Nonrelational Databases. I originally had planned to have another schema for comments in which the comment ID would be referenced in the post schema as a foreign key. So the best way I could think of doing the comments was to do a PATCH on the postId. I know the comments are suppose to also follow the CRUD structure, but I am really unsure how to do this with the NoSQL structure.

```
router.patch( path: '/:postId/comment'
```

## Phase B: Enforcing authentication/verification functionalities

With the use of the jsonwebtoken dependency, users are given a token when logging in to use the API. That token is then used in the header to authenticate the continued use of the API. On top of token authentication, the API uses joi to verify requirements for adding data to the database, such as email is in correct format, password length, post and comment length, etc. All of these will be shown as part of the test cases in Phase D.

Those two dependencies can only go so far. Within each PATCH function (such as for likes and comments) and the DELETE function, I had added additional code to verify that the correct user is logged in to be able to interact with the post they are trying to interact with. This is done by matching the username, found by using the token, to the owner of the post. Phase D has an additional test case included to show these additional checks.

```
const loginValidation = (data) : any => {  
  const schemaValidation : ObjectSchema<any> = joi.object( schema: {  
    email:joi.string().required().min( limit: 6).max( limit: 256).email(),  
    password:joi.string().required().min( limit: 3).max( limit: 1024)  
  })  
  return schemaValidation.validate(data)  
}
```

```
const registerValidation = (data) : any => {  
  const schemaValidation : ObjectSchema<any> = joi.object( schema: {  
    username:joi.string().required().min( limit: 3).max( limit: 256),  
    email:joi.string().required().min( limit: 6).max( limit: 256).email(),  
    password:joi.string().required().min( limit: 3).max( limit: 1024)  
  })  
  return schemaValidation.validate(data)  
}
```

```
const postValidation = (data) : any => {  
  const schemaValidation : ObjectSchema<any> = joi.object( schema: {  
    title:joi.string().required().min( limit: 1).max( limit: 1024),  
    text:joi.string().required().min( limit: 1).max( limit: 1024)  
  })  
  return schemaValidation.validate(data)  
}
```

## Phase C: Development of the MiniWall RESTful API

The MiniWall API has two database models: users and posts. Users stores the username, password, email, and date registered of a person who signs up to use the API. Posts stores the title of the post, the main text of the post, the owner (or who created) of the post, date created, an array for the likes (the array stores the username but then when GET is called, it shows a count called numLikes), and an array for the comments (stored as username who commented, date the comment was made, and the text of the comment). Phase D shows the different validations used (besides the ones from joi pictured above) when accessing the API incorrectly. I have tried to include as many pictures as possible to cover

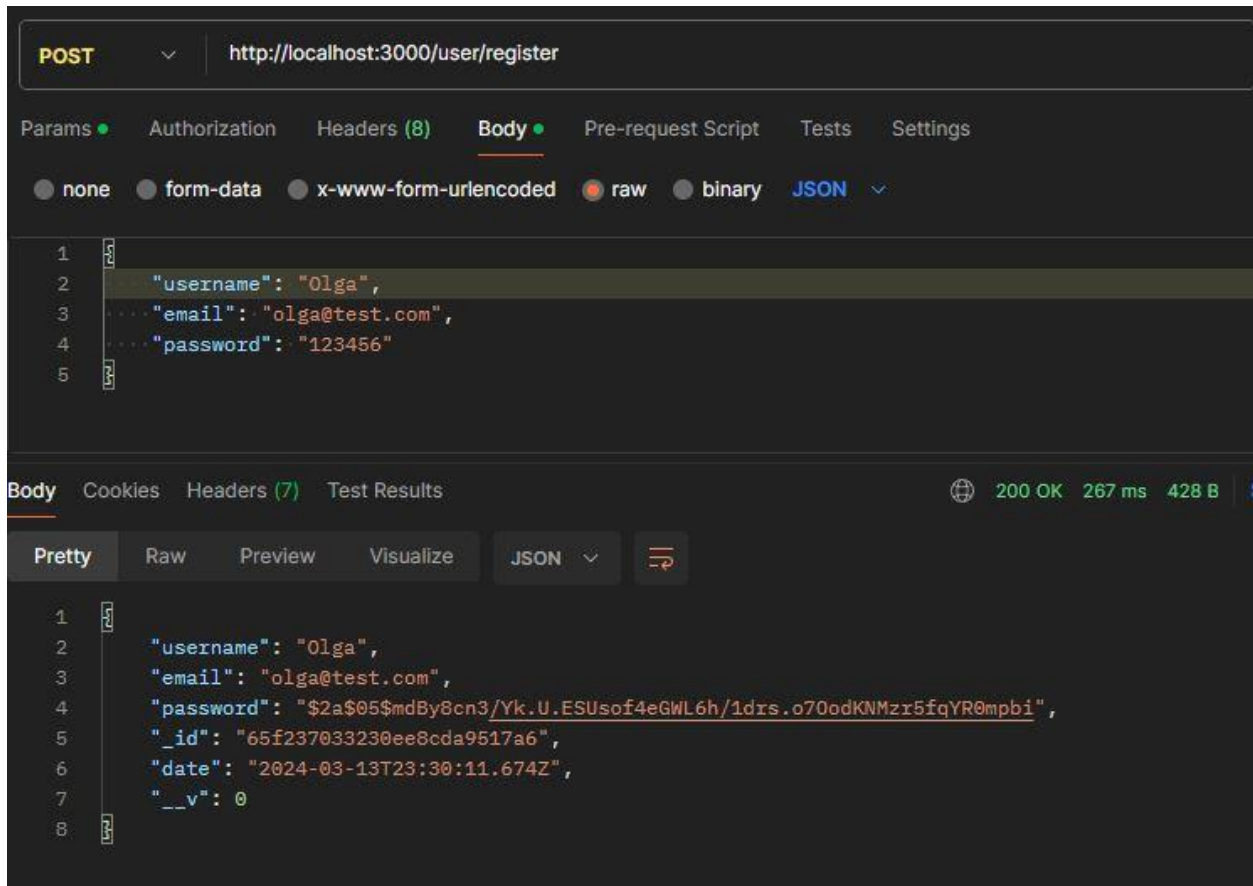
everything, but I am sure I have missed a thing or two. Thus, having a look at the code in the github folder will give a better idea of what the API looks like.

```
const userSchema = mongoose.Schema( definition: {  
  username:{  
    type:String,  
    require:true,  
    min:3,  
    max:256  
  },  
  email:{  
    type:String,  
    require:true,  
    min:6,  
    max:256  
  },  
  password:{  
    type:String,  
    require:true,  
    min:3,  
    max:1024  
  },  
  date:{  
    type:Date,  
    default:Date.now  
  }  
})
```

```
const postSchema = mongoose.Schema( definition: {  
  title:{  
    type:String,  
    require:true,  
    min:1,  
    max:1024  
  },  
  date:{  
    type:Date,  
    default:Date.now  
  },  
  owner:{  
    type:String,  
    require:true  
  },  
  text:{  
    type:String,  
    require:true,  
    min:1,  
    max:1024  
  },  
  likes:[{  
    type:String,  
  }],  
  comments:[{  
    date:{  
      type:Date,  
      default:Date.now  
    },  
    username:{  
      type:String,  
      require:true  
    },  
    comment:{  
      type:String,  
      require:true,  
      min:1,  
      max:1024  
    }  
  }  
  ],  
})
```

## Phase D: Development of the MiniWall testing cases

TC 1: Olga, Nick, and Mary register in the application and access the API





**POST** ⌵ `http://localhost:3000/user/register`

Params ● Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ⌵

```
1 {  
2   "username": "Olga",  
3   "email": "olga@test.com",  
4   "password": "123456"  
5 }
```

Body Cookies Headers (7) Test Results 🌐 400 Bad Request 96 ms 277 B

Pretty Raw Preview Visualize **JSON** ⌵ ≡

```
1 {  
2   "message": "User already exists"  
3 }
```

**POST** ⌵ `http://localhost:3000/user/register`

Params ● Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ⌵

```
1 {  
2   "username": "Mary",  
3   "email": "marytest.com",  
4   "password": "123456"  
5 }
```

Body Cookies Headers (7) Test Results 🌐 400 Bad Request 5 ms 289 B

Pretty Raw Preview Visualize **JSON** ⌵ ≡

```
1 {  
2   "message": "\"email\" must be a valid email"  
3 }
```





**POST** `http://localhost:3000/user/register`

Params • Authorization Headers (8) **Body** • Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binary **JSON** ▾

```
1 {
2   "username": "Mary",
3   "email": "mary@test.com",
4   "password": "12"
5 }
```

**Body** Cookies Headers (7) Test Results  **400 Bad Request** 7 ms 312 B

Pretty Raw Preview Visualize **JSON** ▾ 


```
1 {
2   "message": "\"password\" length must be at least 3 characters long"
3 }
```


**POST** `http://localhost:3000/user/register`

Params • Authorization Headers (8) **Body** • Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binary **JSON** ▾

```
1 {
2   "username": "M",
3   "email": "mary@test.com",
4   "password": "123456"
5 }
```

**Body** Cookies Headers (7) Test Results  **400 Bad Request** 4 ms 312 B

Pretty Raw Preview Visualize **JSON** ▾ 

```
1 {
2   "message": "\"username\" length must be at least 3 characters long"
3 }
```

TC 2: Olga, Nick, and Mary will use the oAuth v2 authorization service to get their tokens

POST ▼ http://localhost:3000/user/login

Params • Authorization Headers (8) **Body •** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary **JSON ▼**

```
1 {  
2   "email": "olga@test.com",  
3   "password": "123456"  
4 }
```

Body Cookies Headers (8) Test Results 🌐 200 OK 94 ms 565 B

Pretty Raw Preview Visualize **JSON ▼**

```
1 {  
2   "auth-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3     eyJfawQiOiI2NWYyMzcwMzMyMzBlZThjZGE5NTE3YTUiLCJpYXQiOiE3MTAzNzMwODV9.  
       tp0JuhYIXpoWnlyVsBG5zJCRJFHhKn-yojg_Ms4ncPE"
```

POST ▼ http://localhost:3000/user/login

Params • Authorization Headers (8) **Body •** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary **JSON ▼**

```
1 {  
2   "email": "nick@test.com",  
3   "password": "123456"  
4 }
```

Body Cookies Headers (8) Test Results 🌐 200 OK 96 ms 565 B

Pretty Raw Preview Visualize **JSON ▼**

```
1 {  
2   "auth-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3     eyJfawQiOiI2NWYyMzg0MTMyMzBlZThjZGE5NTE3YWQiLCJpYXQiOiE3MTAzNzMwMDF9.  
       U4HgTfDYwYdqz6wn0MBSjYepTcF7qbWQqV21o59lIWg"
```

**POST** `http://localhost:3000/user/login`

Params • Authorization Headers (8) **Body** • Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ▾

```
1 {
2   "email": "mary@test.com",
3   "password": "123456"
4 }
```

**Body** Cookies Headers (8) Test Results 200 OK 93 ms 565 B

Pretty Raw Preview Visualize **JSON** ▾

```
1 {
2   "auth-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2NWYyMzgzNDMyMzBlZThjZGE5NTE3YWElLCJpYXQiOiE3MTAzNzMyMzZ9.P5h_wGb2_XK-55i48t_mY20ek830EK_DKWgnlwHm-tg"
3 }
```

**POST** `http://localhost:3000/user/login`

Params • Authorization Headers (8) **Body** • Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ▾

```
1 {
2   "email": "marytest.com",
3   "password": "123456"
4 }
```

**Body** Cookies Headers (7) Test Results 400 Bad Request 4 ms 289 B

Pretty Raw Preview Visualize **JSON** ▾

```
1 {
2   "message": "\"email\" must be a valid email"
3 }
```

**POST** ▼ `http://localhost:3000/user/login`

Params ● Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ▼

```
1 {
2   "email": "mary@test.com",
3   "password": "12345"
4 }
```

**Body** Cookies Headers (7) Test Results 🌐 400 Bad Request 92 ms 279 B

**Pretty** Raw Preview Visualize **JSON** ▼ ≡

```
1 {
2   "message": "Password is incorrect"
3 }
```

**POST** ▼ `http://localhost:3000/user/login`

Params ● Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ▼

```
1 {
2   "email": "andy@test.com",
3   "password": "123456"
4 }
```

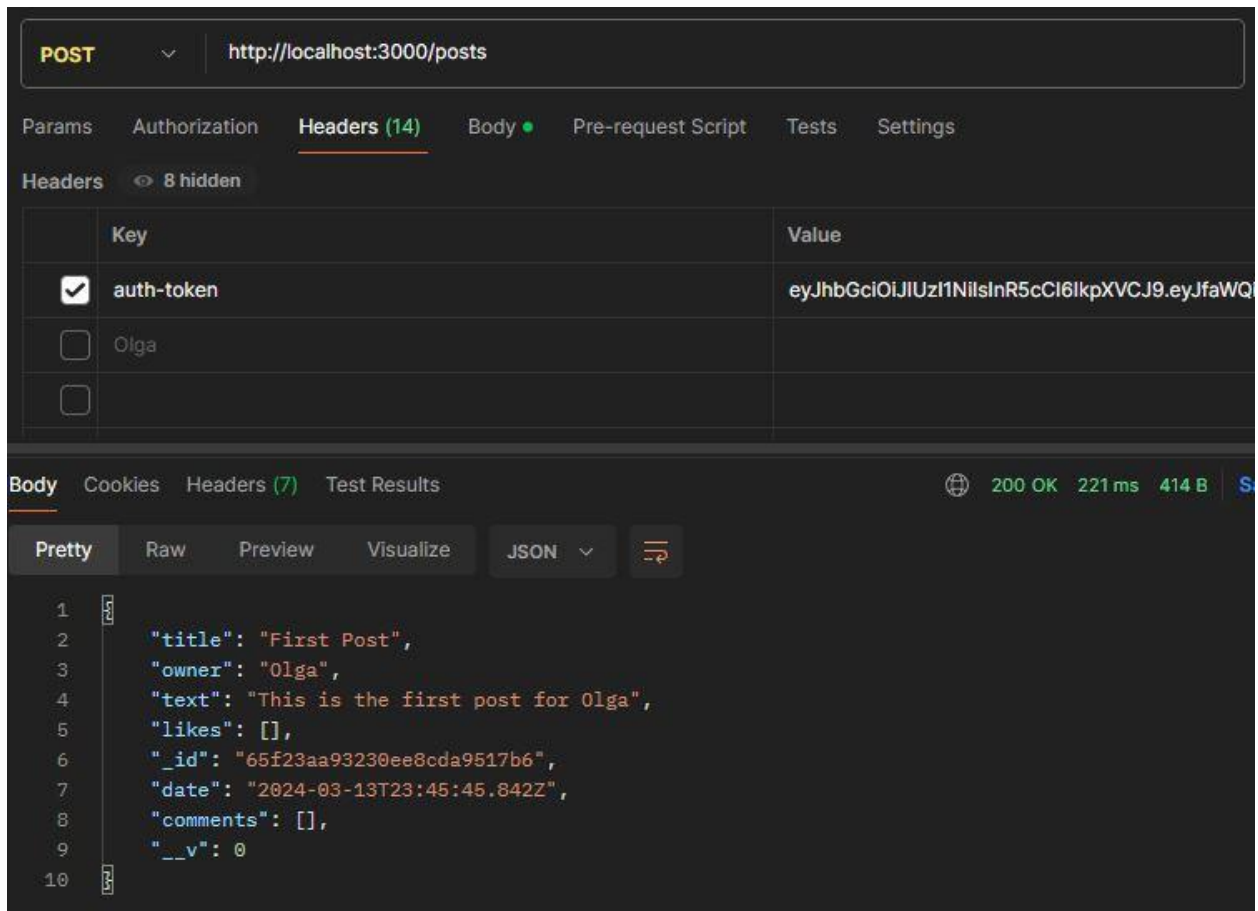
**Body** Cookies Headers (7) Test Results 🌐 400 Bad Request 90 ms 278 B

**Pretty** Raw Preview Visualize **JSON** ▼ ≡

```
1 {
2   "message": "User does not exists"
3 }
```

The screenshot shows the Chrome DevTools Network tab with a POST request to `http://localhost:3000/posts`. The request status is **401 Unauthorized**. The Headers tab is active, displaying an `auth-token` header. The Body tab shows a JSON response: `{ "message": "Access denied" }`.

TC 4: Olga posts a text using her token



**POST** ⌵ `http://localhost:3000/posts`

Params Authorization Headers (14) **Body** • Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ⌵

```
1  {
2    "title": "",
3    "text": "This is the first post for Olga"
4  }
```

**Body** Cookies Headers (7) Test Results 🌐 400 Bad Request 4 ms 294 B

**Pretty** Raw Preview Visualize **JSON** ⌵ ⇌

```
1  {
2    "message": "\"title\" is not allowed to be empty"
3  }
```

**POST** ⌵ `http://localhost:3000/posts`

Params Authorization Headers (14) **Body** • Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ⌵

```
1  {
2    "title": "First Post",
3    "text": ""
4  }
```

**Body** Cookies Headers (7) Test Results 🌐 400 Bad Request 4 ms 293 B

**Pretty** Raw Preview Visualize **JSON** ⌵ ⇌

```
1  {
2    "message": "\"text\" is not allowed to be empty"
3  }
```



## TC 5: Nick posts a text using his token

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/posts
- Body Type:** JSON
- Request Body:**

```
1 {
2   "title": "First Post",
3   "text": "This is the first post for Nick"
4 }
```
- Response:** 200 OK, 192 ms, 414 B
- Response Body (Pretty):**

```
1 {
2   "title": "First Post",
3   "owner": "Nick",
4   "text": "This is the first post for Nick",
5   "likes": [],
6   "_id": "65f23b673230ee8cda9517bd",
7   "date": "2024-03-13T23:48:55.613Z",
8   "comments": [],
9   "__v": 0
10 }
```

## TC 6: Mary posts a text using her token

**POST** ▼ `http://localhost:3000/posts`

Params Authorization Headers (14) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ▼

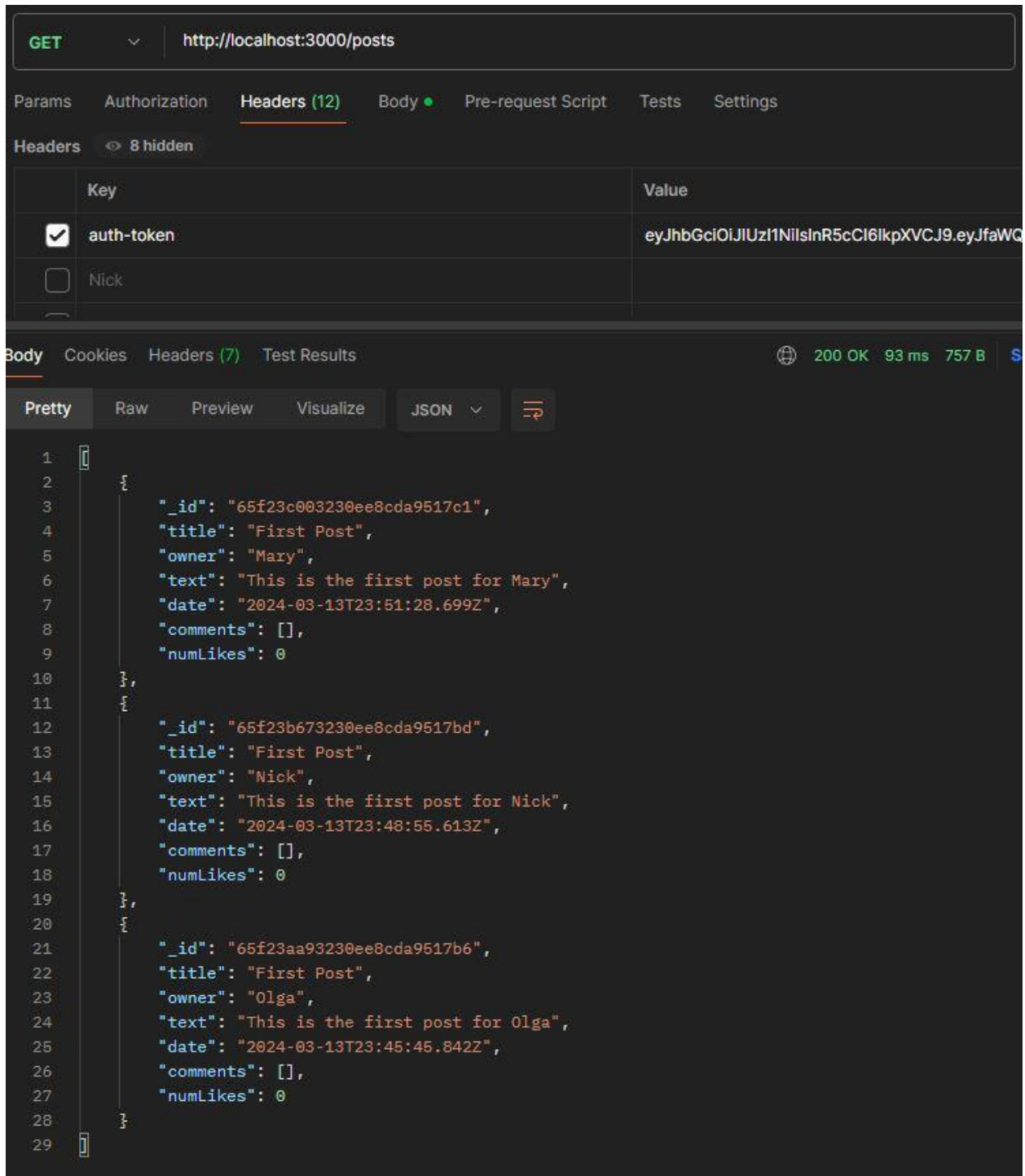
```
1  {
2    "title": "First Post",
3    "text": "This is the first post for Mary"
4  }
```

**Body** Cookies Headers (7) Test Results 🌐 200 OK 191 ms 414 B

Pretty Raw Preview Visualize **JSON** ▼ 🔍

```
1  {
2    "title": "First Post",
3    "owner": "Mary",
4    "text": "This is the first post for Mary",
5    "likes": [],
6    "_id": "65f23c003230ee8cda9517c1",
7    "date": "2024-03-13T23:51:28.699Z",
8    "comments": [],
9    "__v": 0
10 }
```

TC 7: Nick and Olga browse available posts in reverse chronological order in the MiniWall; there should be three posts available. Note: that we do not have any likes yet



19

[illegible]

20

A screenshot of the Postman application interface. The top bar shows a PATCH request to http://localhost:3000/posts/65f23c003230ee8cda9517c1/comment. Below the URL bar are tabs for Params, Authorization, Headers (10), Body, Pre-request Script, Tests, and Settings. The Headers tab is selected, displaying a table with two headers: Key and Value. The first header is auth-token with a checked checkbox, and its value is eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQ... The second header is Mary with an unchecked checkbox. At the bottom, there's a status bar showing a 400 Bad Request response, 184 ms duration, and 287 B body size.

The screenshot shows a web browser with a REST client interface. The top bar indicates a GET request to `http://localhost:3000/posts`. Below the address bar, there are tabs for Params, Authorization, Headers (10), Body, Pre-request Script, Tests, and Settings. The Headers tab is selected, showing a table with two headers: `auth-token` (checked) and `Mary` (unchecked). The Body tab is also visible, showing a JSON response. The response is a JSON array of three posts, each with fields like `_id`, `title`, `owner`, `text`, `date`, `comments`, and `numLikes`. The first post is owned by Mary, the second by Nick, and the third by Olga. The comments array for the first post contains two comments from Nick and Olga. The numLikes field is 0 for all posts.

Key	Value
<input checked="" type="checkbox"/> <code>auth-token</code>	<code>eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJfjfaWQ</code>
<input type="checkbox"/> <code>Mary</code>	

```
{
  "id": "65f23c003230ee8cda9517c1",
  "title": "First Post",
  "owner": "Mary",
  "text": "This is the first post for Mary",
  "date": "2024-03-13T23:51:28.699Z",
  "comments": [
    {
      "username": "Nick",
      "comment": "Nick's comment",
      "id": "65fdacf2dc53623b18d3d7d8",
      "date": "2024-03-22T16:08:18.756Z"
    },
    {
      "username": "Olga",
      "comment": "Olga's comment",
      "id": "65fdad65dc53623b18d3d7df",
      "date": "2024-03-22T16:10:13.326Z"
    }
  ],
  "numLikes": 0
},
{
  "id": "65f23b673230ee8cda9517bd",
  "title": "First Post",
  "owner": "Nick",
  "text": "This is the first post for Nick",
  "date": "2024-03-13T23:48:55.613Z",
  "comments": [],
  "numLikes": 0
},
{
  "id": "65f23aa93230ee8cda9517b6",
  "title": "First Post",
  "owner": "Olga",
  "text": "This is the first post for Olga",
  "date": "2024-03-13T23:45:45.842Z",
  "comments": [],
  "numLikes": 0
}
```



TC 11: Mary can see the comments for her posts

See TC10

TC 12: Nick and Olga like Mary's posts

The screenshot shows a REST client interface with the following details:

- Method:** PATCH
- URL:** `http://localhost:3000/posts/65f23c003230ee8cda9517c1/like`
- Headers:** 10 headers are listed, with 8 hidden. The visible headers are:

Key	Value
<input checked="" type="checkbox"/> auth-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQ9...
<input type="checkbox"/> Nick	
- Body:** The response body is shown in JSON format:

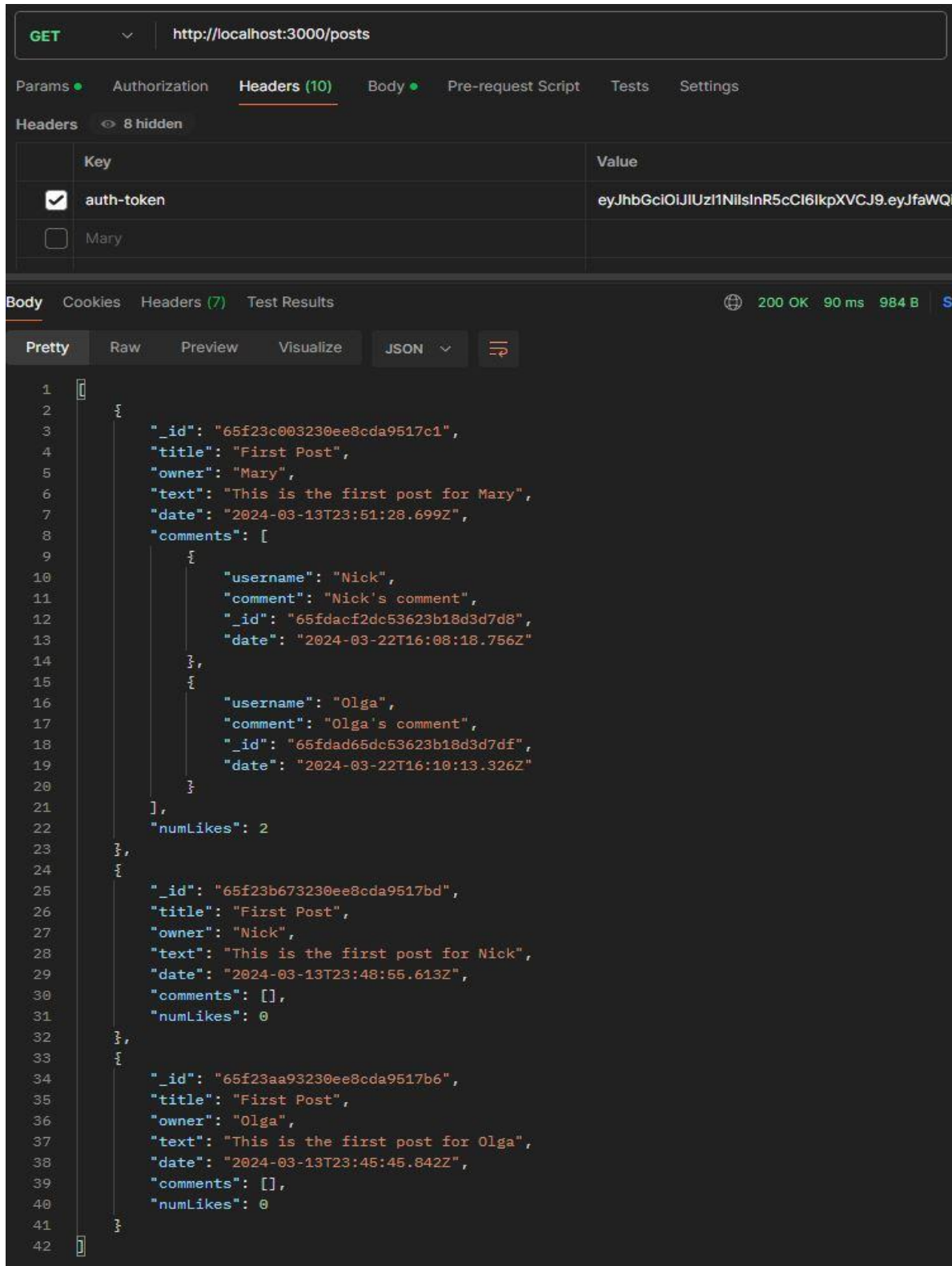
```
1 {  
2   "acknowledged": true,  
3   "modifiedCount": 1,  
4   "upsertedId": null,  
5   "upsertedCount": 0,  
6   "matchedCount": 1  
7 }
```
- Status:** 200 OK, 366 ms, 327 B

The screenshot shows the Postman application interface. At the top, the URL bar displays 'http://localhost:3000/posts/65f23c003230ee8cda9517c1/like' with a 'PATCH' method selected. Below the URL bar, there are tabs for 'Params', 'Authorization', 'Headers (10)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers' tab is currently selected, showing a table with two headers: 'Key' and 'Value'. The first row has a checked checkbox, the key 'auth-token', and the value 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQ'. The second row has an unchecked checkbox, the key 'Nick', and an empty value field. Below the headers, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Body' tab is selected, showing a JSON response: {"message": "User already liked the post"}. At the bottom right, the status bar shows a '400 Bad Request' status, '267 ms' time, and '285 B' size.



[illegible]

TC 14: Mary can see that there are two likes in her posts



TC 15: Nick can see the list of posts, since Mary's post has two likes it is shown at the top

GET http://localhost:3000/posts

Params Authorization Headers (10) Body Pre-request Script Tests Settings

Headers 8 hidden

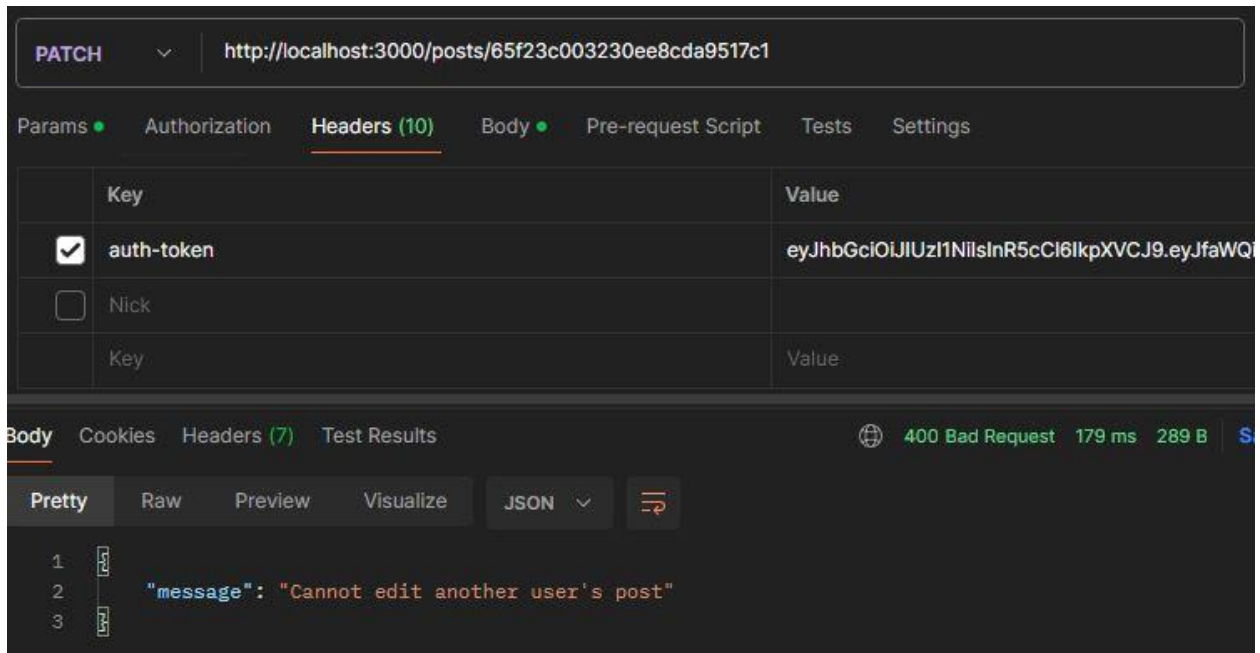
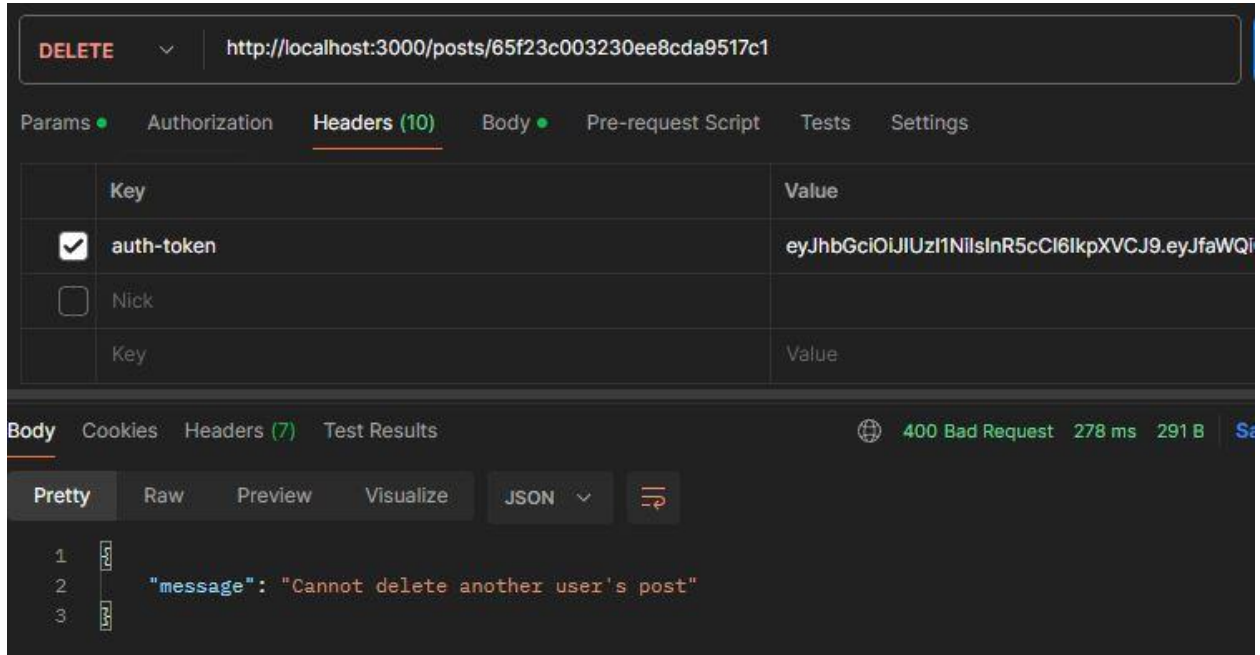
Key	Value
<input checked="" type="checkbox"/> auth-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJfaWQiOiJ1b3RlbnQ1IiwiaWF0IjoxNzE5MjM0MjY5Lm51bnQ1
<input type="checkbox"/> Nick	

Body Cookies Headers (7) Test Results 200 OK 90 ms 984 B

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "_id": "65f23c003230ee8cda9517c1",
4     "title": "First Post",
5     "owner": "Mary",
6     "text": "This is the first post for Mary",
7     "date": "2024-03-13T23:51:28.699Z",
8     "comments": [
9       {
10        "username": "Nick",
11        "comment": "Nick's comment",
12        "_id": "65fdacf2dc53623b18d3d7d8",
13        "date": "2024-03-22T16:08:18.756Z"
14      },
15      {
16        "username": "Olga",
17        "comment": "Olga's comment",
18        "_id": "65fdad65dc53623b18d3d7df",
19        "date": "2024-03-22T16:10:13.326Z"
20      }
21    ],
22     "numLikes": 2
23   },
24   {
25     "_id": "65f23b673230ee8cda9517bd",
26     "title": "First Post",
27     "owner": "Nick",
28     "text": "This is the first post for Nick",
29     "date": "2024-03-13T23:48:55.613Z",
30     "comments": [],
31     "numLikes": 0
32   },
33   {
34     "_id": "65f23aa93230ee8cda9517b6",
35     "title": "First Post",
36     "owner": "Olga",
37     "text": "This is the first post for Olga",
38     "date": "2024-03-13T23:45:45.842Z",
39     "comments": [],
40     "numLikes": 0
41   }
42 }
```

## TC 16: Extra functionality





GET http://localhost:3000/posts

Params • Authorization Headers (10) Body • Pre-request Script Tests Settings

Headers 8 hidden

	Key	Value
<input checked="" type="checkbox"/>	auth-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiJ1b3R5IiwiaWF0IjoiYXNjaWQ
<input type="checkbox"/>	Nick	

Body Cookies Headers (7) Test Results 200 OK 93 ms 1.11 KB

Pretty Raw Preview Visualize JSON

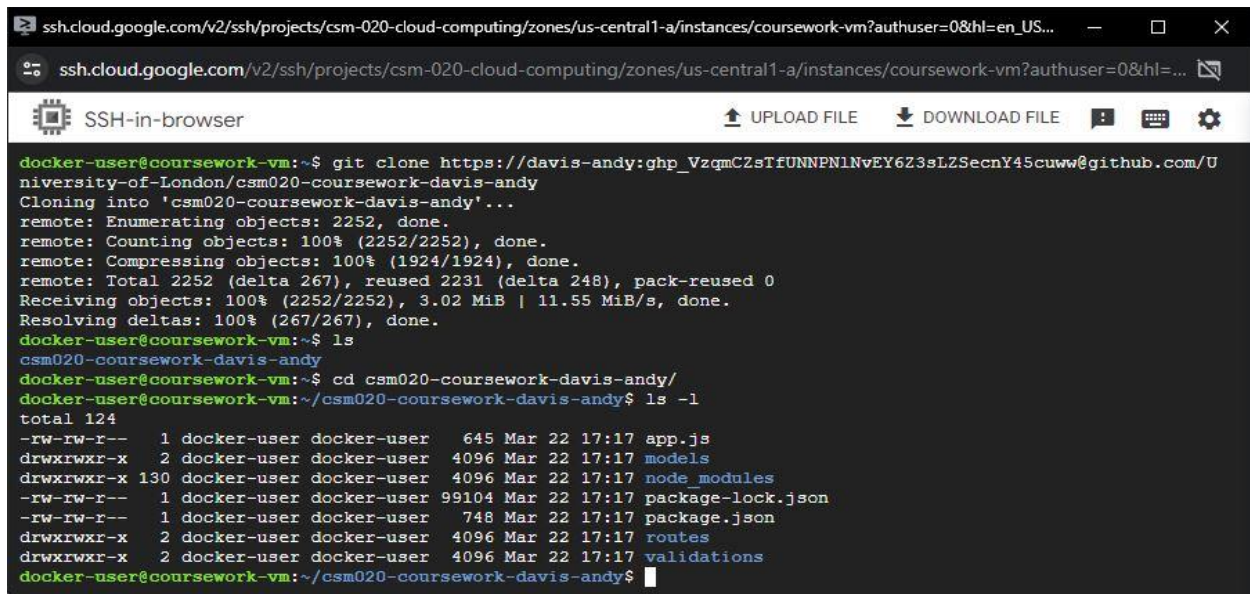
```
1 {
2   {
3     "_id": "65f23c003230ee8cda9517c1",
4     "title": "First Post",
5     "owner": "Mary",
6     "text": "This is the first post for Mary",
7     "date": "2024-03-13T23:51:28.699Z",
8     "comments": [
9       {
10        "username": "Nick",
11        "comment": "Nick's comment",
12        "_id": "65fdacf2dc53623b18d3d7d8",
13        "date": "2024-03-22T16:08:18.756Z"
14      },
15      {
16        "username": "Olga",
17        "comment": "Olga's comment",
18        "_id": "65fdad65dc53623b18d3d7df",
19        "date": "2024-03-22T16:10:13.326Z"
20      }
21    ],
22    "numLikes": 2
23  },
24  {
25    "_id": "65fdafc2dc53623b18d3d800",
26    "title": "second post",
27    "owner": "Nick",
28    "text": "second post",
29    "date": "2024-03-22T16:20:18.661Z",
30    "comments": [],
31    "numLikes": 0
32  },
33  {
34    "_id": "65f23b673230ee8cda9517bd",
35    "title": "First Post",
36    "owner": "Nick",
37    "text": "This is the first post for Nick",
38    "date": "2024-03-13T23:48:55.613Z",
39    "comments": [],
40    "numLikes": 0
41  },
42  }
```

## Phase E: Deploy your MiniWall project into a GCP VM using Docker

First thing that was needed to deploy a Docker image was to clone the GitHub repository into the VM. On top of that, the .env file would have to be manually transferred over (best practices not to have it in the repo). For this module, though, I have the .env in the repo only because there is nothing I really care to hide within that file. It is not shown in the first image below, but the file does exist on the VM.

Next up is creating the Docker file and then building an image and running a container. The second image below shows the Docker image being built first, but the last bit shows the actual Docker file as well as the Docker container up and running.

Finally, third image below shows access to the VM using its IP address.



```
ssh.cloud.google.com/v2/ssh/projects/csm-020-cloud-computing/zones/us-central1-a/instances/coursework-vm?authuser=0&hl=en_US...
ssh.cloud.google.com/v2/ssh/projects/csm-020-cloud-computing/zones/us-central1-a/instances/coursework-vm?authuser=0&hl=en_US...
SSH-in-browser
UPLOAD FILE
DOWNLOAD FILE

docker-user@coursework-vm:~$ git clone https://davis-andy:ghp_VzqmCZsTfUNNPn1NvEY6Z3sLZSecnY45cuww@github.com/University-of-London/csm020-coursework-davis-andy
Cloning into 'csm020-coursework-davis-andy'...
remote: Enumerating objects: 2252, done.
remote: Counting objects: 100% (2252/2252), done.
remote: Compressing objects: 100% (1924/1924), done.
remote: Total 2252 (delta 267), reused 2231 (delta 248), pack-reused 0
Receiving objects: 100% (2252/2252), 3.02 MiB | 11.55 MiB/s, done.
Resolving deltas: 100% (267/267), done.
docker-user@coursework-vm:~$ ls
csm020-coursework-davis-andy
docker-user@coursework-vm:~$ cd csm020-coursework-davis-andy/
docker-user@coursework-vm:~/csm020-coursework-davis-andy$ ls -l
total 124
-rw-rw-r-- 1 docker-user docker-user 645 Mar 22 17:17 app.js
drwxrwxr-x 2 docker-user docker-user 4096 Mar 22 17:17 models
drwxrwxr-x 130 docker-user docker-user 4096 Mar 22 17:17 node_modules
-rw-rw-r-- 1 docker-user docker-user 99104 Mar 22 17:17 package-lock.json
-rw-rw-r-- 1 docker-user docker-user 748 Mar 22 17:17 package.json
drwxrwxr-x 2 docker-user docker-user 4096 Mar 22 17:17 routes
drwxrwxr-x 2 docker-user docker-user 4096 Mar 22 17:17 validations
docker-user@coursework-vm:~/csm020-coursework-davis-andy$
```

```
ssh.cloud.google.com/v2/ssh/projects/csm-020-cloud-computing/zones/us-central1-a/instances/coursework-vm?authuser=0&hl=en_US&projectNumber=753132805405&useAdminP...
ssh.cloud.google.com/v2/ssh/projects/csm-020-cloud-computing/zones/us-central1-a/instances/coursework-vm?authuser=0&hl=en_US&projectNumber=75313280540...
SSH-in-browser
[+] Building 2.9s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 150B
=> [internal] load metadata for docker.io/library/alpine:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/4] FROM docker.io/library/alpine:latest@sha256:rc5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4ff27761f8e1ad6b
=> [internal] load build context
=> => transferring context: 14.19MB
=> CACHED [2/4] RUN apk add --update nodejs npm
=> [3/4] COPY . /src
=> [4/4] WORKDIR /src
=> exporting to image
=> => exporting layers
=> => writing image sha256:7bcba5552dc7bf28310b82a6c8ca43f9b68ae800688c74729d3f55def00cac1b
=> => naming to docker.io/library/miniwall-image:1
docker-user@coursework-vm:~/csm020-coursework-davis-andy$ docker container run -d --name miniwall --publish 80:3000 miniwall-image:1
8284eaeabd4fa92650e4f3d50589c5f3ad4aa3a36d25c6947b52b226ffc89754
docker-user@coursework-vm:~/csm020-coursework-davis-andy$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
8284eaeabd4f   miniwall-image:1  "node ./app.js"         4 seconds ago  Up 2 seconds  0.0.0.0:80->3000/tcp, :::80->3000/tcp  miniwall
docker-user@coursework-vm:~/csm020-coursework-davis-andy$ cat Dockerfile
FROM alpine
RUN apk add --update nodejs npm
COPY . /src
WORKDIR /src
EXPOSE 3000
ENTRYPOINT ["node", "./app.js"]
docker-user@coursework-vm:~/csm020-coursework-davis-andy$
```

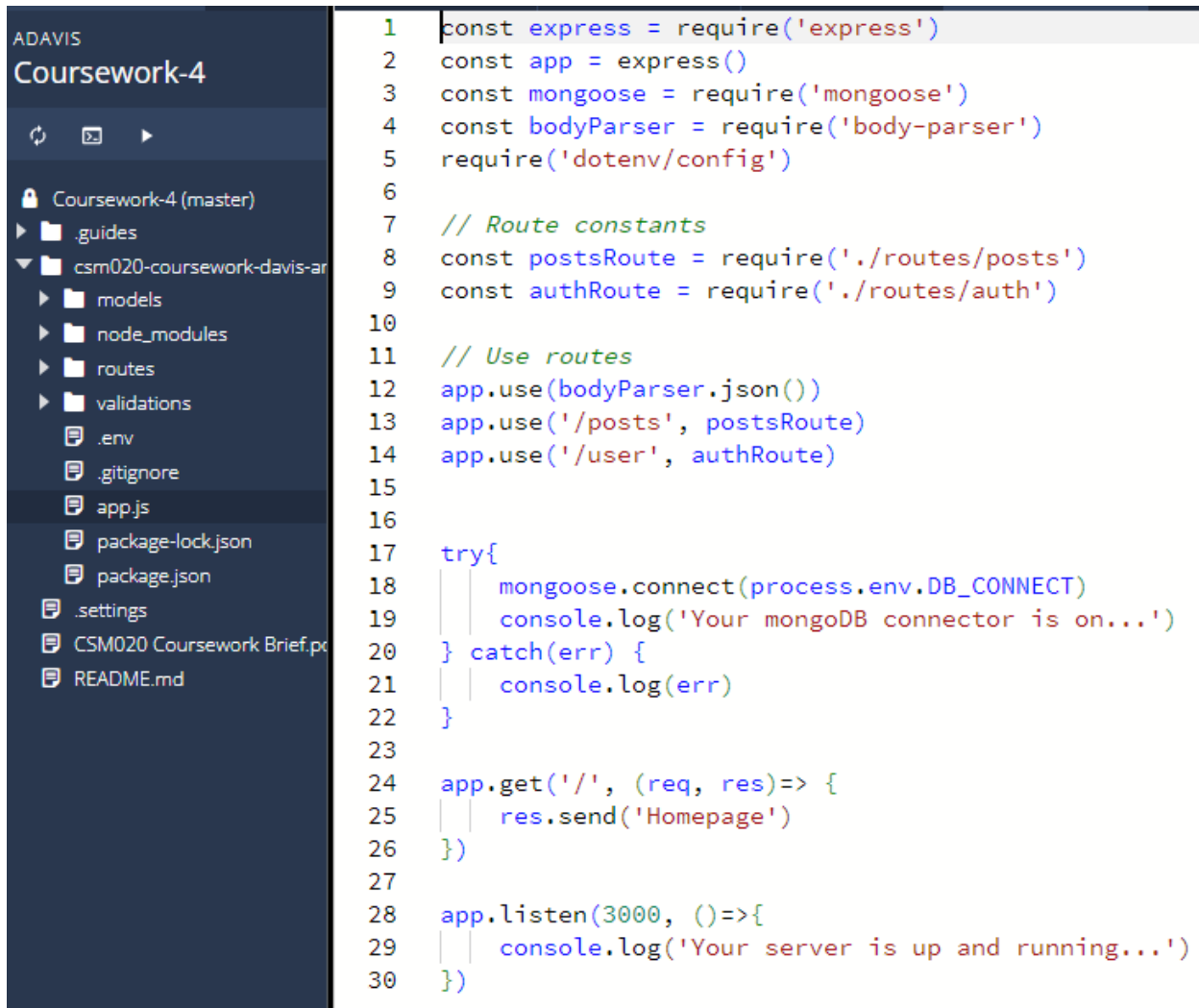
33

## Phase F: Document your MiniWall solution in a technical report

I am not the best at writing reports, especially if there is no template to follow, so I am hoping that this report will be satisfactory. I just structured this to follow each phase listed in the brief. Part of the struggle of writing this report is the need to show what is going on. I feel like just having the pictures and GitHub is plenty to go off on, and a report seems a bit redundant, but I understand where this is coming from and why it must be made. Seeing as this is my first time creating an API, one in NodeJS nonetheless, getting into the weeds and developing past what was given in the labs would have been way beyond my capabilities. There are already a few things implemented that took way longer than they should have, especially since a non-relational database is tricky to deal with. One of those issues was how to deal with the likes on a post. The final solution I came up with was to have an array that just adds the username of the person liking the post. This caused me to go down a rabbit hole of figuring out adding and removing items into an array and then making sure to not create duplicates. Once that was done, I now had to figure out how to count those likes in order to sort the posts, which was another rabbit hole of sorting by likes first and then by date created. In the end, it may be fairly basic and just a minor extension from the labs, but I think it covers all the needs of this coursework and is functional with a good coverage of test cases.

## Phase G: Submit quality scripts

The code has been submitted into Codio (cloned from the GitHub repository) and has been written to what I would consider to be of very good quality.



```
1 const express = require('express')
2 const app = express()
3 const mongoose = require('mongoose')
4 const bodyParser = require('body-parser')
5 require('dotenv/config')
6
7 // Route constants
8 const postsRoute = require('./routes/posts')
9 const authRoute = require('./routes/auth')
10
11 // Use routes
12 app.use(bodyParser.json())
13 app.use('/posts', postsRoute)
14 app.use('/user', authRoute)
15
16
17 try{
18   mongoose.connect(process.env.DB_CONNECT)
19   console.log('Your mongoDB connector is on...')
20 } catch(err) {
21   console.log(err)
22 }
23
24 app.get('/', (req, res)=> {
25   res.send('Homepage')
26 })
27
28 app.listen(3000, ()=>{
29   console.log('Your server is up and running...')
30 })
```