MiniWall API Documentation

2993 Words

By Kai Bowers

Table of Contents

1. Introduction

This technical report is a documentation of MiniWall, a cloud based Software as a Service (SaaS) API tailored for seamless sharing and interaction within a digital community. This report serves as a detailed account of conceptualizing, developing, and deploying MiniWall, encapsulating the essential aspects of cloud SaaS development.

Additionally, this report provides insights into the utilization of the API, enabling readers to integrate MiniWall functionalities into their own projects.

The code for this project can be found at https://github.com/University-of-London/csm020-coursework-KaiBowers99

2. Setup

The application was developed using NodeJS and the following libraries:

Name	Version
bcryptjs	^2.4.3
body-parser	^1.20.2
dotenv	^16.4.2
express	^4.18.2
joi	^17.12.1
jsonwebtoken	^9.0.2
mongoose	^8.1.2
nodemon	^3.0.3
supertest	^6.3.4
jest	^29.7.0

All libraries can be installed by using the command “npm install” within the main directory.

2.1 Database Connection

The project requires a MongoDB connection. For this, create a .env file in the main directory of the project containing “DB_CONNECTOR” and “TOKEN_SECRET”.

```
DB_CONNECTOR=mongodb+srv://myconnection
TOKEN_SECRET=token
```

2.2 Run

To run use the following command within the projects main directory:

```
npm start
```

for testing, use:

```
npm test
```

(!) Testing directly creates content on the database. To run the test twice, remove data created by the initial test run.

3. Project Structure

The MiniWall project contains four primary components: the API, the frontend, the app.js, and the testing section. The API splits into four subsections: models, routes, validations and verification. All key files are listed here:

App.js

This is the main file of the server and is located in the primary directory.

API Models

The “models” folder contains the relevant mongoose schemas for the project.

`/models/postmodel.js`

`/models/threadsmodel.js`

`/models/usermodel.js`

API Routes + Frontend

The “routes” folder contains all routes for the API, and a subfolder for the frontend.

`/routes/auth.js`

`/routes/posts.js`

`/routes/threads.js`

`/routes/frontend/dashboard.js`

API Validations + Verifications

Validations and verifications serve to maintain security and formatting.

`/validations/validation.js`

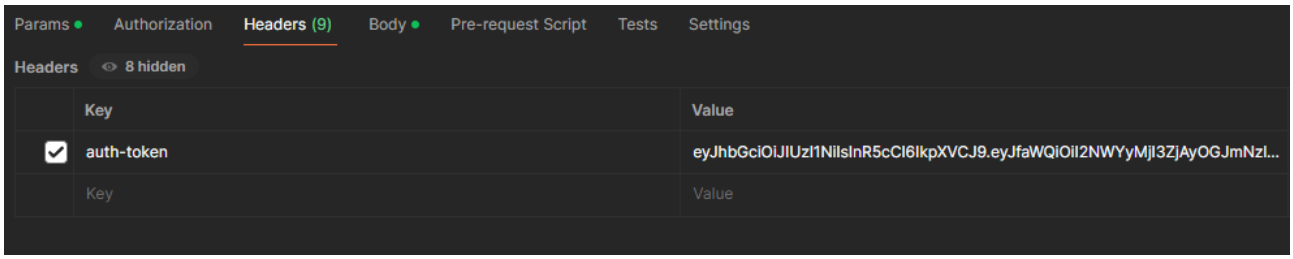
`/verification/verifyDataExists.js`

`/verification/verifyToken.js`

4. Routing

This section presents a catalogue of all endpoints (chapter 4.2) and responses (chapter 4.1 + 4.2) the server provides. Each entry includes a URL, endpoint type, expected JSON body, and a list of possible JSON formatted responses.

All endpoints except **`/api/user/register`** and **`/api/user/login`** require the auth-token to be set in the header of the request. The auth-token can be acquired using the login endpoint.



4.1 Generic Server Responses

All endpoints include generic responses, which are listed here to reduce duplication in the comprehensive endpoints list in chapter 3.2. The comprehensive list of endpoints will only feature the responses that are specific to each endpoint.

<u>Authentication</u>

Hashes for both the userID and username are included within the authentication token. This measure increases security by reducing identity theft risks.

All error responses listed in this chapter adhere to the following JSON format :
`{ “message”: output }`

If the user has not correctly authenticated but attempts to access information using any endpoint, the following server responses occur:

Code	Output	Notes
401	`Access denied, missing token.`	The user has not provided a auth-token in the header.
401	`Invalid token, ID does not match.`	auth-token provided, but incorrect userID.
401	`Invalid token, usernames do not match.`	auth-token provided, but incorrect username.
401	`Access denied, auth-token expired.`	auth-token provided, but invalid for other reasons.

All responses from this point on are only returned after the authentication checks, so they do not leak information a non-user shouldn't be able to see.

Unknown Server Error

If at any point a error occurs which isn't explicitly designed to be caught, the following error is returned:

Code	Output	Notes
500	`{ “message”: output }`	`The internal server error will be printed in the output field.`

PostID

the following responses can appear with all endpoints using “`:PostID`”:

Code	Output
400	`Post ID's must have a length of 24.`
404	`Post does not exist.`

ThreadID

the following responses can appear with all endpoints using "`:ThreadID`":

Code	Output
400	`Thread ID's must have a length of 24.`
404	`Thread does not exist.`

Comments in Threads

All Endpoints which contain both "`:ThreadID`" and "`:CommentID`" with an incorrect `CommentID` will output one of the following responses:

Code	Output
400	`CommentID's must have a length of 24.`
404	`Comment not found within the Thread.`

Incorrect Request Body

All endpoints will output a error if the JSON Body is incorrectly defined. The Error message will be in the following form.

Code	Output
400	`Message: ...`

The validation of string formats are done with the Joi package. The "message" field will be filled with the error generated by Joi.

4.2 Comprehensive list of all endpoints

This is a complete list of all routes provided by the server. The responses provided here serve as supplementary options to those previously mentioned above.

In this section, the outputs vary in format; while some adhere to the standard 'message' format like above, others refer to mongoose models such as comments, posts, and threads. Mongoose models in this chapter are marked with stars (*). Outputs that follow the “message” format are explicitly denoted using quotation marks.

The models are printed using JSON with their fields as described in chapter 5. “PostID”, “CommentID”, “ThreadID” refer to the “_id” field provided by mongoose in every model as a default.

The requirements outlined in this chapter are shown using their Joi formatting code. [Square brackets] in this and following chapters are used to denote Arrays.

All outputs using Code 200 should be considered successful and requests which do not have a body defined do not require one. All fields are required.

1. Register a new user

POST **/api/user/register**

Body

Field	Type	Requirements
`email`	`String`	min(6).max(256).email()
`username`	`String`	min(3).max(256)
`password`	`String`	min(6).max(1024)

Unique responses

Code	Output	Notes
`400`	“`User already exists.`”	This message appears if a user with the same name or email has already signed up.
`200`	`Usermodel*`	

2. Login to a existing account

POST **/api/user/login**

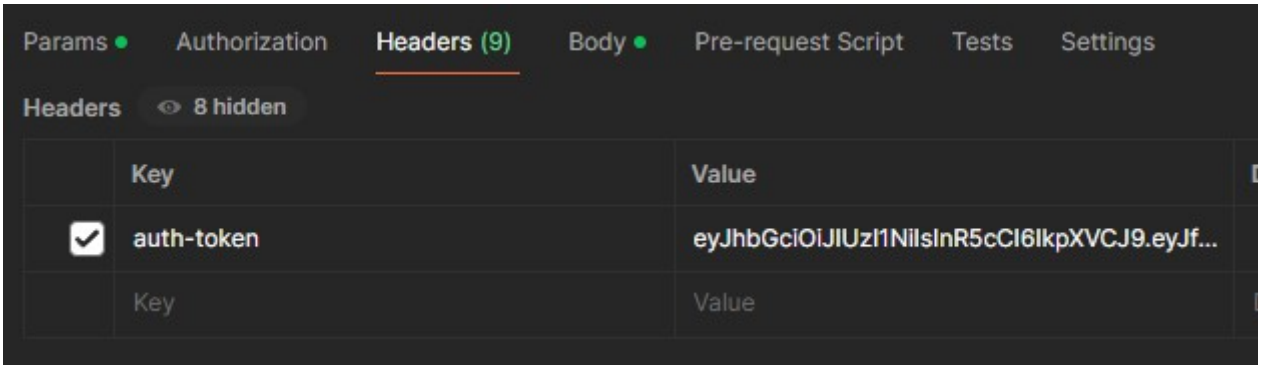
Body

Field	Type	Requirements
`email`	`String`	min(6).max(256).email()
`password`	`String`	min(6).max(1024)

Unique responses

Code	Output	Notes
`400`	`"User doesn't exist."`	The provided credentials cannot be found in the database.
`400`	`"incorrect password"`	The provided password is incorrect.
`200`	`String containing auth-token`	Success! The output provides the auth-token for further usage of the service.

From this point on, the auth-token must be provided in the header.



3. Create a new post

POST **/api/posts/new**

Body

Field	Type	Requirements	Notes
`Title`	`String`	min(6).max(256)	The post's title
`Description`	`String`	min(6).max(1024)	The post's content

Unique responses

Code	Output
`200`	`Postmodel*`

4. Read all posts

GET **/api/posts/**

The posts returned here are sorted primarily by likes and secondarily by newest.

Unique Responses

Code	Output
`200`	`[Postmodel*]`

5. Read post by PostID

GET **/api/posts/:postID**

by providing /api/posts/ with a ID, a singular post can be viewed.

Unique Responses

Code	Output
`200`	`Postmodel*`

6. Update a post's title, description and owner data using a PostID

PATCH **/api/posts/:postID**

Body

Field	Type	Requirements	Notes
`Title`	`String`	min(6).max(256)	The Post's title
`Description`	`String`	min(6).max(1024)	The Post's description

Unique responses

Code	Output
`200`	`Postmodel*`

7. Add Comment to Post By PostID

PATCH **/api/posts/:postID/comment**

Adding a comment to a post will create a new thread under the post with the comment included. If there are already comments under the post, the comment is added to the existing main thread of the post.

Body

Field	Type	Requirements
`text`	`String`	min(6).max(1024)

Unique responses

Code	Output	Notes
400	“You cannot comment on your own post!"	If you attempt to comment on a post you have created, you receive this error.
200	Threadmodel*	

8. Like Post by PostID

PATCH **/api/posts/:postID/like**

Unique responses

Code	Output
400	"You can't like this post - you are the owner!"
400	“You have already liked this post!”
200	Postmodel*

9. Un-Like Post by PostID

PATCH **/api/posts/:postID/unlike**

The term “unlike” refers to removing a like.

Unique responses

Code	Output
400	"You can't Un-like (or Like) this post - you are the owner!"
400	"You haven't liked this post yet!"
200	Postmodel*

10. Delete Post By PostID

DELETE **/api/posts/:postID**

Unique responses

Code	Output
200	"Success!"

11. Read by ThreadID

GET **/api/threads/:threadID**

Unique responses

Code	Output
200	Threadmodel*

12. Read Comment within Thread using ID's

GET **/api/threads/:threadID/:commentID/**

Unique responses

Code	Output
404	"Comment not found within the post”
200	Threadmodel*

13. Add Comment by ThreadID

PATCH **/api/threads/:threadID/comment/**

Unique responses

Code	Output
200	`Threadmodel*`

14. Reply to a comment (allows commenting on comments recursively)

PATCH **/api/threads/:threadID/:commentID/comment**

Unique responses

Code	Output
200	`Threadmodel*`

15. Like a comment by commentID

PATCH **/api/threads/:threadID/:commentID/like**

Unique responses

Code	Output
400	`"You can't like your own comment"`
400	`"You have already liked this comment"`
200	`"Comment liked successfully"`

16. Remove a like from a comment using ID's

PATCH **/api/threads/:threadID/:commentID/unlike**

Unique responses

Code	Output
400	"You have not liked this comment"
200	"Comment unliked successfully"

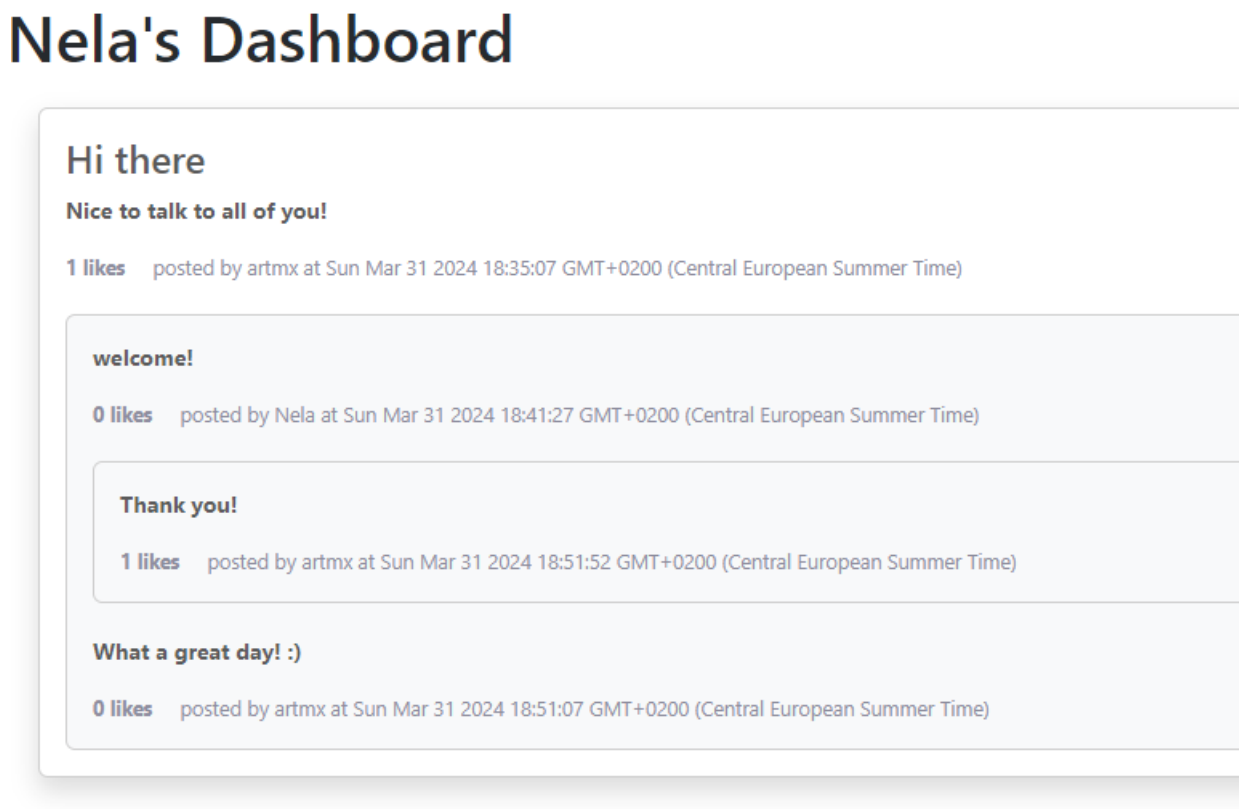
17. Frontend Dashboard

GET **/Dashboard**

This read-only dashboard requires the user to first log in using the API. The screenshot below shows a example dashboard of a user.

Unique responses

Code	Output	Notes
200	HTML	The output can be viewed directly in browser.



5. Database Models

MiniWall uses MongoDB as its database management system. The database consists of three primary models: Users, Threads, and Posts. These structures are outlined below.

Fields marked with a star (*) are not shared with API users. This could be due to confidentiality concerns or because these fields are utilized for internal logic purposes.

All models contain a generated id (_id), which is provided by the mongoose library.

5.1 User Model

The user model acts as a repository of users, with each entry representing a distinct user.

Field Name	Type	Properties	Notes
Username	String	Required, min:3, max:256	
Email*	String	Required, min:6, max:256	
Password*	String	Required, min:6, max:256	hashed representation of the password
Timestamp*	Date	default:Date.now	Account creation date

5.2 Post Model

The post model contains all information about a post made by a user.

Field Name	Type	Properties	Notes
Title	String	Required, min:6, max:256	
Description	String	Required, min:6, max:1024	The content/text of this post.
OwnerName	String	Required, min:6, max:256	The name of the user which posted this.
OwnerId*	String	Required, min:6, max:256	The ID of the user which posted this
UserIDsLiked*	[String]	default: []	A list of all user ID's which liked this post. Used internally for removing likes/only allowing one like per user.
Timestamp	Date	default:Date.now	Time of posting
ThreadID	String	default:"Empty"	When the first user comments on this post, a new thread is created, which stores the information about the post's comment section. The “_id” of the thread is then stored here.
Likes	Int	virtual	calculated dynamically using the length of UserIDsLiked

5.3 Thread and Comment Models

The thread and comment models store all discussion-related data associated with a post. These models handle comments on posts as well as comments on other comments. This design choice was made to enable a recursive comment section with a hierarchy of comments.

Thread Schema

Field Name	Type	Properties	Notes
`OriginalPostID`	`String`	`Required, min:6, max:256`	Stores the original Post this comments belongs to, even if this is a thread under another thread.
`Comments`	`[commentSchema]`	`default: []`	Comments in this thread which are on this level of the comment hierarchy.

Comment Schema

Field Name	Type	Properties	Notes
`Text`	`String`	`required`	Content
`Timestamp`	`Date`	`default: Date.now`	Date when the comment was posted
`PostedByName`	`String`	`required`	Username of poster
`SubthreadID`	`String`	`default:"Empty"`	If someone comments on this comment, a new thread is started and its Id is stored here. This allows for tracking through the recursive comment hierarchy.
`UserIDsLiked*`	`[String]`	`default: []`	Array of userID's who liked this post. Used internally for removing likes/only allowing one like per user.
`Likes`	`Int`	`virtual`	calculated dynamically using UserIDsLiked's length

6. User Authentication and Verification

When users try to access MiniWall, they must provide their credentials using a auth-token. The server verifies their identity by checking for the existence of the user and confirming the password's correctness using bcryptjs. This involves comparing the hashed password stored in the mongoose database with the provided hash. If the credentials are valid, the server generates a JSON Web Token (JWT) using the jsonwebtoken library.

Once the user is successfully authenticated and the JWT is generated, the token is returned the client. It contains encrypted user information such as the ID and username, and it's signed with a unique secret key only known by the server.

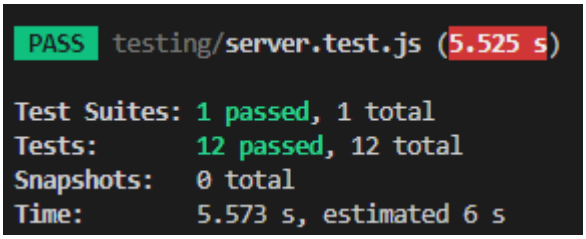
Subsequently, when the user attempts to access additional resources within MiniWall, they include this JWT in the HTTP headers of their requests as described in Chapter 3.

7.Testing

In this section, I will present the requested test cases from 1 to 15, along with documentation that includes screenshots of the test implementation, the frontend (via `/Dashboard`), the results for API users and the mongoose database.

All code screenshots are from the server.test.js file using supertest as the testing framework. The tests require a database connection, as defined in Chapter 2, and they will directly write to the database.

You can execute the tests locally by running "npm test" to receive the following output:



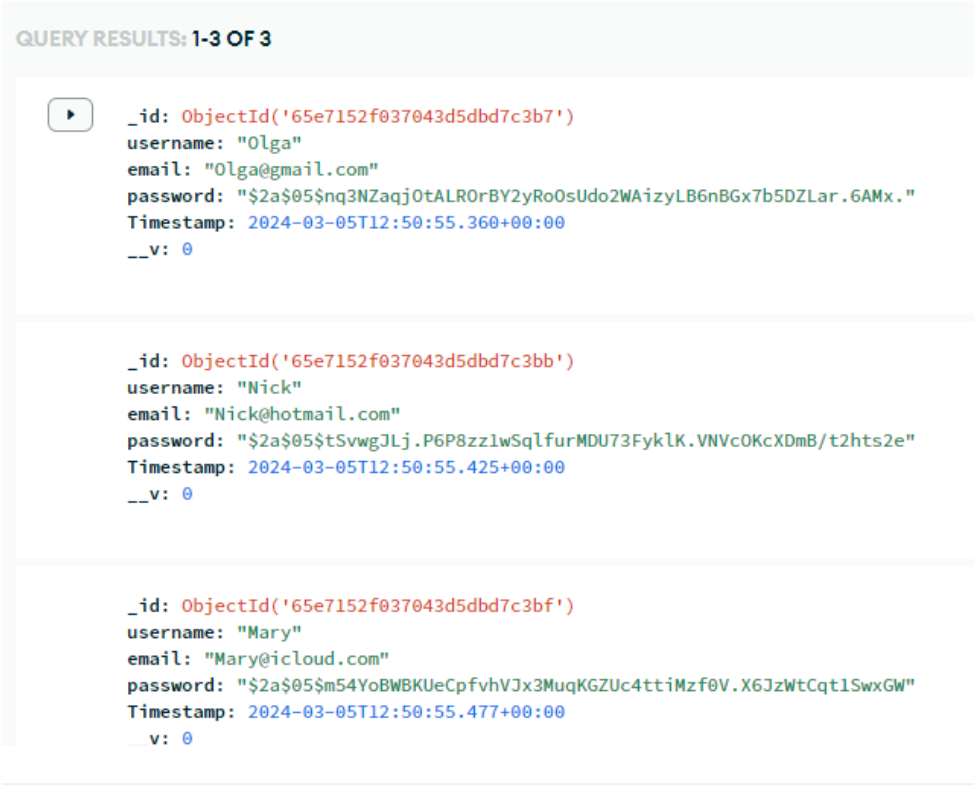
Test Case 1+2

Olga, Nick and Mary register in the application and access the API. Olga, Nick and Mary will use the oAuth v2 authorisation service to get their tokens.

```
describe('MiniWall Main Test', () => {

    // These are the fake credentials used by Olga, Nick and Mary
    const MockCredentials = [
        { username: "Olga", email: "Olga@gmail.com", password: "SuperSecretPassword"},
        { username: "Nick", email: "Nick@hotmail.com", password: "1234abcd" },
        { username: "Mary", email: "Mary@icloud.com", password: "qwerty1111" },
    ];

    //TC 1+2: Olga, Nick and Mary register in the application and access the API.
    it('Token Access Test', async () => {
        for(var i = 0; i < 3; i++)
        {
            // Register
            const response_register = await request(app)
            .post('/api/user/register/')
            .send(MockCredentials[i])

            // Successful response?
            expect(response_register.status).toBe(200);

            // Login
            const response_login = await request(app)
            .post('/api/user/login/')
            .send({
                "email": MockCredentials[i].email,
                "password": MockCredentials[i].password
            })

            // Successful response?
            expect(response_login.status).toBe(200);

            /// Verify token information
            const token = response_login.body['auth-token'];
            const decodedToken = jwt.decode(token);

            expect(decodedToken).toBeTruthy();
            expect(decodedToken._id).toBeTruthy();
            expect(decodedToken.iat).toBeTruthy();

            // Add token to the mock credentials array
            MockCredentials[i].Token = token
        }
    });
```

As seen above, we start by setting up mock credentials for our API users. The array containing these credentials is defined outside the initial test to ensure accessibility throughout subsequent tests. We validate successful responses (Status 200) for both registration and login processes. Subsequently, we confirm the validity of the token using the jsonwebtoken library. Finally, we store the token in the array for continued use throughout the test suite.

We can further validate the test by inspecting the mongoose database. Within, we'll find the three accounts:



Test Case 3

Olga calls the API (any endpoint) without using a token. This call should be unsuccessful as the user is unauthorised.

```
// TC 3: Olga calls the API (any endpoint) without using a token. This call should be unsuccessful as the user is unauthorised.
it('Token Access Test', async () => {

    // Access posts without a token
        const get_all_posts_without_token = await request(app)
        .get('/api/posts/') // Skip adding token
        .send({})

        // Verify correct error message
        expect(get_all_posts_without_token.body).toMatchObject({"message": "Access denied, missing token."});

        // Missing token results in 401 : "Unauthorized" (learn.microsoft.com/en-us/dotnet/api/system.net.httpstatuscode)
        expect(get_all_posts_without_token.status).toBe(401);
});
```

As expected, in this testing scenario, we receive the "401" error code along with the error output as defined in verifyToken.js.

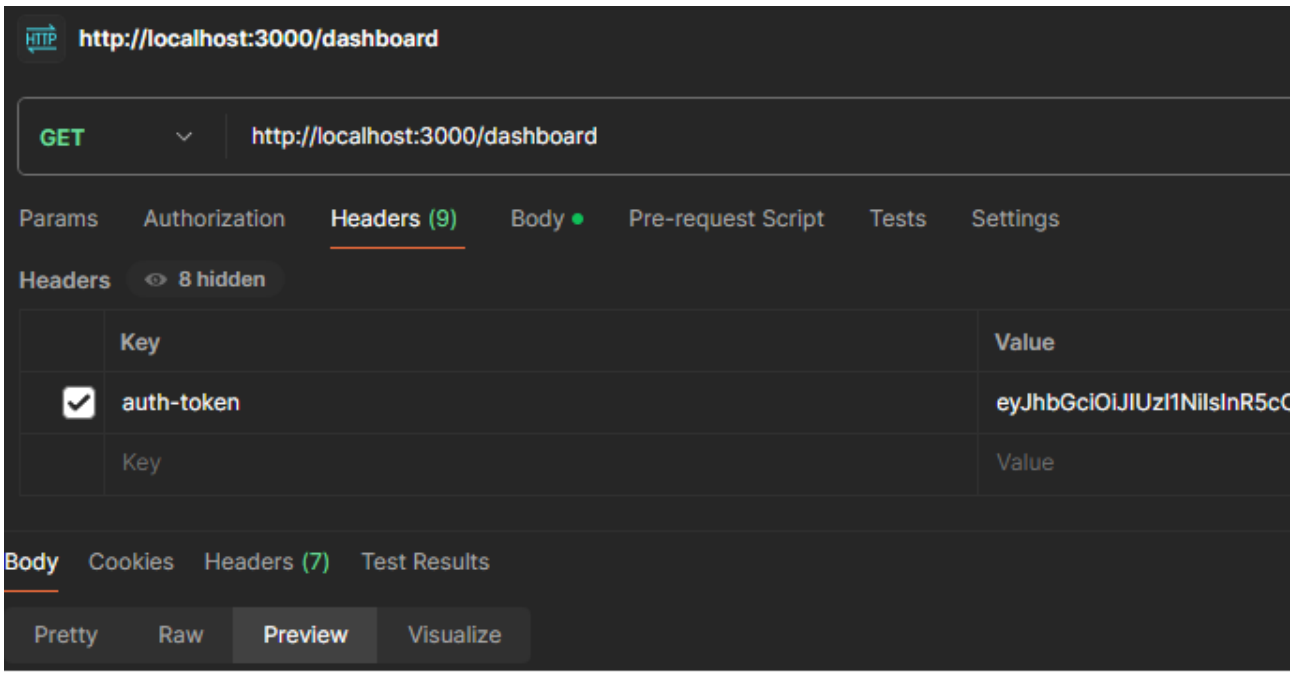
Test Case 4-6

Olga, Nick and Mary each post a text using their token.

```
// TC 4-6: Olga, Nick, Mary post texts using their tokens.
it('Posting Test', async () => {
    for(var i = 0; i < 3; i++)
    {
        // Olga, Nick, and Mary each make a post saying Hi and providing their e-mail in the description
        const making_a_post = await request(app)
        .post('/api/posts/new')
        .set('auth-token', MockCredentials[i].Token)
        .send({
           "Title": "Hello! I'm " + MockCredentials[i].username + ".",
           "Description": "You can e-mail me at " + MockCredentials[i].email,
        })

        // Check for successful status
        expect(making_a_post.status).toBe(200);

        // Verify if the output is correct
        expect(making_a_post.body.Title).toEqual("Hello! I'm " + MockCredentials[i].username + ".");
        expect(making_a_post.body.Description).toEqual("You can e-mail me at " + MockCredentials[i].email);
        expect(making_a_post.body).toMatchObject({
            Description: expect.any(String),
            OwnerName: MockCredentials[i].username,
            Timestamp: expect.any(String),
            Title: expect.any(String),
            Likes: 0,
            __v: expect.any(Number),
            _id: expect.any(String),
            threadID: "Empty"
        });

        // If Mary is currently posting, save the generated ID for further testing.
        if(MockCredentials[i].username == "Mary") {
            MarysPostID = making_a_post.body._id
        }

        // Wait 1.5s after posting.
        // Isn't nessessary, but makes the chronological sorting more obvious on the frontend.
        if(i != 2) {
            await new Promise(resolve => setTimeout(resolve, 1500));
        }
    }
});
  You, 3 weeks ago • Added remaining test cases
```

We iterate through the credentials of the three users and generate three posts, each containing their respective name and email addresses. After creating the post, the function returns the successfully created post object. We ensure the accuracy of the post by comparing all fields specified in the post request.

To validate the posts on the dashboard, we can log into Mary's account using Postman and ensure that all posts are displayed in accurately and in correct order.



Mary's Dashboard

Hello! I'm Mary.

You can e-mail me at Mary@icloud.com

0 likes posted by Mary at Tue Mar 05 2024 17:50:43 GMT+0100 (Central European Standard Time)

Hello! I'm Nick.

You can e-mail me at Nick@hotmail.com

0 likes posted by Nick at Tue Mar 05 2024 17:50:41 GMT+0100 (Central European Standard Time)

Hello! I'm Olga.

You can e-mail me at Olga@gmail.com

we can also inspect the data on mongoose:

Test Case 7

Nick and Olga browse available posts in reverse chronological order in the MiniWall; there should be three posts available.

```
// TC 7: Nick and Olga browse available posts in chronological order in the MiniWall; there should be three posts available.
it('Nick and Olga Browsing Test', async () => {

    // Nick Browses the available posts
    const nick_all_posts = await request(app)
        .get('/api/posts/')
        .set('auth-token', Credentials_Nick.Token)
        .send({})

    // Nick checks that there are exactly 3 objects in the array
    expect(nick_all_posts.body.length).toBe(3);

    // Nick checks if newer posts (lower time) are shown first
    for (let i = 1; i < nick_all_posts.body.length; i++) {
        const previous = new Date(nick_all_posts.body[i - 1].Timestamp).getTime();
        const current = new Date(nick_all_posts.body[i].Timestamp).getTime();
        expect(previous).toBeLessThanOrEqual(current);
    }

    // Olga browses the available posts
    const olga_all_posts = await request(app)
        .get('/api/posts/')
        .set('auth-token', Credentials_Olga.Token)
        .send({})

    // Olga checks that there are exactly 3 objects in the array
    expect(olga_all_posts.body.length).toBe(3);

    // Olga checks if newer posts (lower time) are shown first
    for (let i = 1; i < olga_all_posts.body.length; i++) {
        const previous = new Date(olga_all_posts.body[i - 1].Timestamp).getTime();
        const current = new Date(olga_all_posts.body[i].Timestamp).getTime();
        expect(previous).toBeLessThanOrEqual(current);
    }
});
```

Here, we repeat the same actions for Nick and Olga. Both verify the number of posts and examine the timestamps to check if they were posted in the correct order. We can also verify this using the frontend, as in Test Case 7.

Test Case 8

Nick and Olga comment Mary's post in a round-robin fashion (one after the other).

```
    You, 3 weeks ago • Added remaining test cases
// TC 8: Nick and Olga comment Mary's post in a round-robin fashion (one after the other).
it('Commenting Test', async () => {
    // Nick sends a comment
    const Response_Comment_Nick = await request(app)
        .patch('/api/posts/' + MarysPostID + '/comment/')
        .set('auth-token', Credentials_Nick.Token)
        .send({
            "text" : "Hi Mary, welcome to Miniwall!"
        })

    // Verify Status
    expect(Response_Comment_Nick.status).toBe(200);

    // Verify if Nick's comment was received
    let MarysPost = await Post.findById(MarysPostID)
    let thread = await Thread.findById(MarysPost.threadID);
    expect(thread.comments[0].text).toBe("Hi Mary, welcome to Miniwall!");

    // Olga sends a comment
    const Response_Comment_Olga = await request(app)
        .patch('/api/posts/' + MarysPostID + '/comment/')
        .set('auth-token', Credentials_Olga.Token)
        .send({
            "text" : "Hey Mary, I just joined Miniwall aswell!"
        })

    // Verify if Olga's comment was received
    expect(Response_Comment_Olga.status).toBe(200);
    thread = await Thread.findById(MarysPost.threadID); // Get Updated Thread
    expect(thread.comments[1].text).toBe("Hey Mary, I just joined Miniwall aswell!");
});
```

In this scenario, both Nick and Olga add a comment to Mary's post. By using the threadID from Mary's post, we access the comment thread, where we observe that both comments have been successfully added.

The mongoose server shows us that Mary's post is now connected to the thread "65e72cf907c2f4c8e189c026"

```
_id: ObjectId('65e72cf907c2f4c8e189c01c')
Title: "Hello! I'm Mary."
Description: "You can e-mail me at Mary@icloud.com"
OwnerName: "Mary"
▸ UserIDsLiked: Array (empty)
threadID: "65e72cf907c2f4c8e189c026"
Timestamp: 2024-03-05T14:32:25.855+00:00
__v: 0
```

Here we can see the Thread “65e72cf907c2f4c8e189c026” and the two comments within:

Test Case 9

Mary comments her post. This call should be unsuccessful; an owner cannot comment on owned posts.

```
// TC 9: Mary comments her post. This call should be unsuccessful; an owner cannot comment owned posts.
it('Marys fails to send a comment', async () => {

    // Mary sends a comment
    const Response_Comment_Mary = await request(app)
        .patch('/api/posts/' + MarysPostID + '/comment/')
        .set('auth-token', Credentials_Mary.Token)
        .send({
            "text" : "Thank you for the kind words Nick & Olga :)"
        })

    // Oh No!
    expect(Response_Comment_Mary.status).toBe(400);
    expect(Response_Comment_Mary.body).toStrictEqual({"message": "You cannot comment on your own post!"});
});
```

We can see here Mary tries to add a comment, but receives a error message.
There is no change to the comment thread as shown in test Case 8.

Test Case 10 + 11

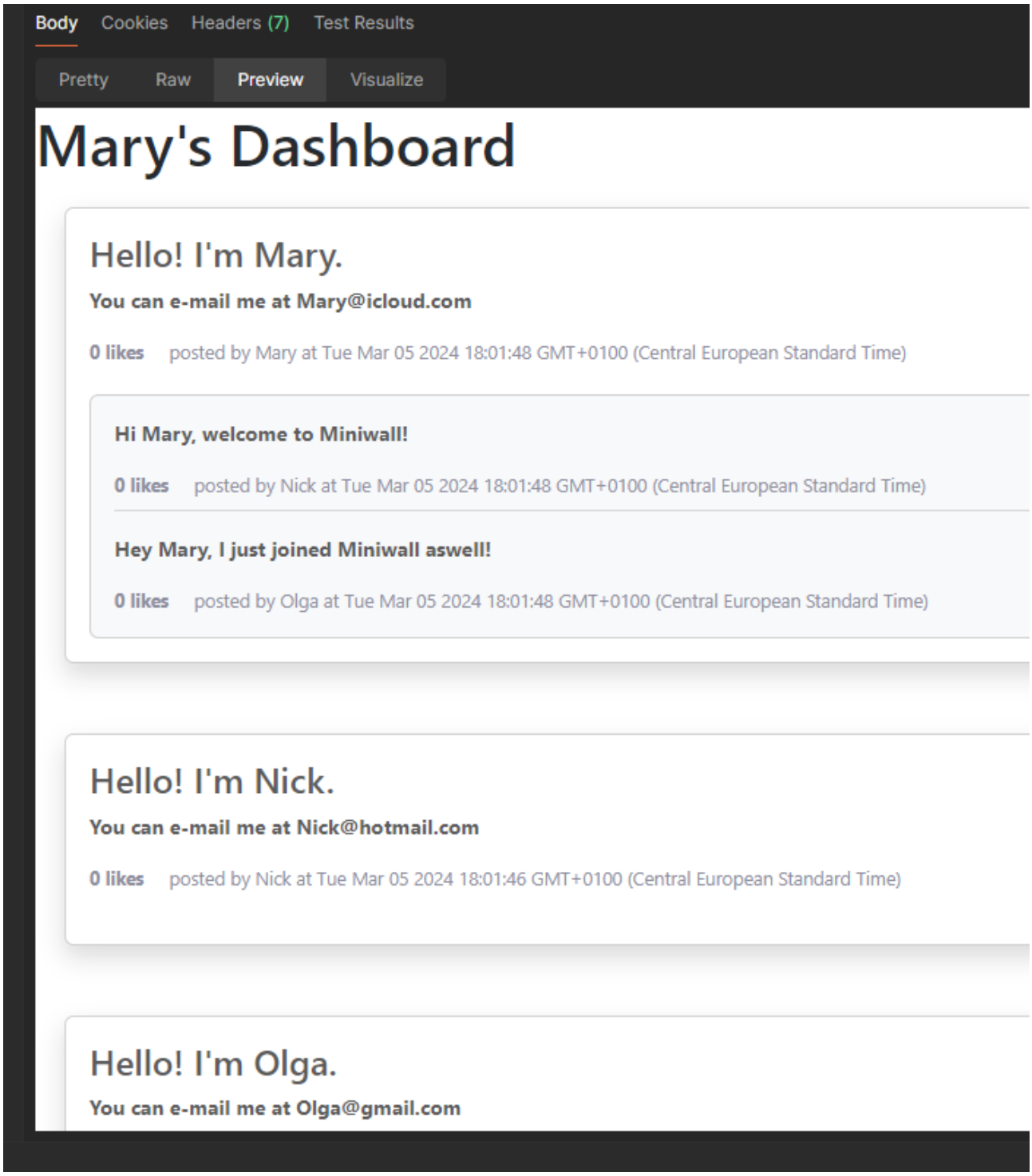
Mary can see posts in reverse chronological order (newest posts are on the top as there are no likes yet).

```
// TC 10:  Mary can see posts in reverse chronological order (newest posts are on the top as there are no likes yet).
it('Mary Browsing Test', async () => {

    // Mary browses the available posts
    Credentials_Mary = MockCredentials[2];
    const mary_all_posts = await request(app)
        .get('/api/posts/')
        .set('auth-token', Credentials_Mary.Token)
        .send({})

    // Mary checks that there are exactly 3 objects in the array
    expect(mary_all_posts.body.length).toBe(3);

    // Mary checks if newer posts (lower time) are shown first
    for (let i = 1; i < mary_all_posts.body.length; i++) {
        const previous = new Date(mary_all_posts.body[i - 1].Timestamp).getTime();
        const current = new Date(mary_all_posts.body[i].Timestamp).getTime();
        expect(previous).toBeLessThanOrEqual(current);
    }
});
```

Mary can see the comments for her post.

```
// TC 11: Mary can see the comments for her posts.
it('Mary reads the comments', async () => {

    // Make sure we have the ID of Mary's post
    Credentials_Mary = MockCredentials[2];
    const MarysPost = await Post.findOne().sort({Timestamp: -1})
    expect(MarysPost.Title).toBe("Hello! I'm Mary.");
    expect(MarysPost._id).toBeDefined();

    // Mary accessess her own post
    const mary_check_own_post = await request(app)
        .get('/api/posts/' + MarysPost._id)
        .set('auth-token', Credentials_Mary.Token)
        .send({})

    expect(mary_check_own_post.status).toBe(200);
    let thread = await Thread.findById(JSON.parse(mary_check_own_post.text).threadID);

    // Mary checks for Nick's comment
    expect(thread.comments[0].text).toBe("Hi Mary, welcome to Miniwall!");

    // Mary checks for Olgas's comment
    expect(thread.comments[1].text).toBe("Hey Mary, I just joined Miniwall aswell!");
});
```

We can also see the test results by logging into Mary's account and viewing the dashboard:



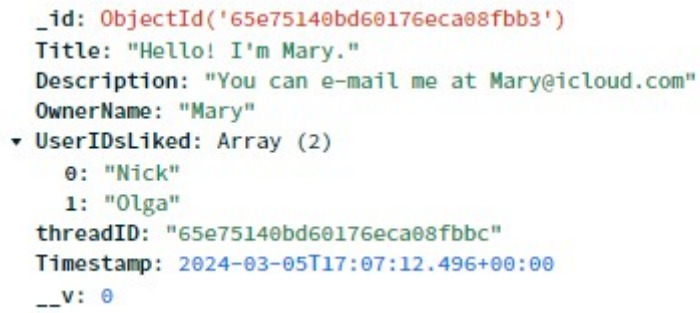
Test Case 12

Nick and Olga like Mary’s posts.

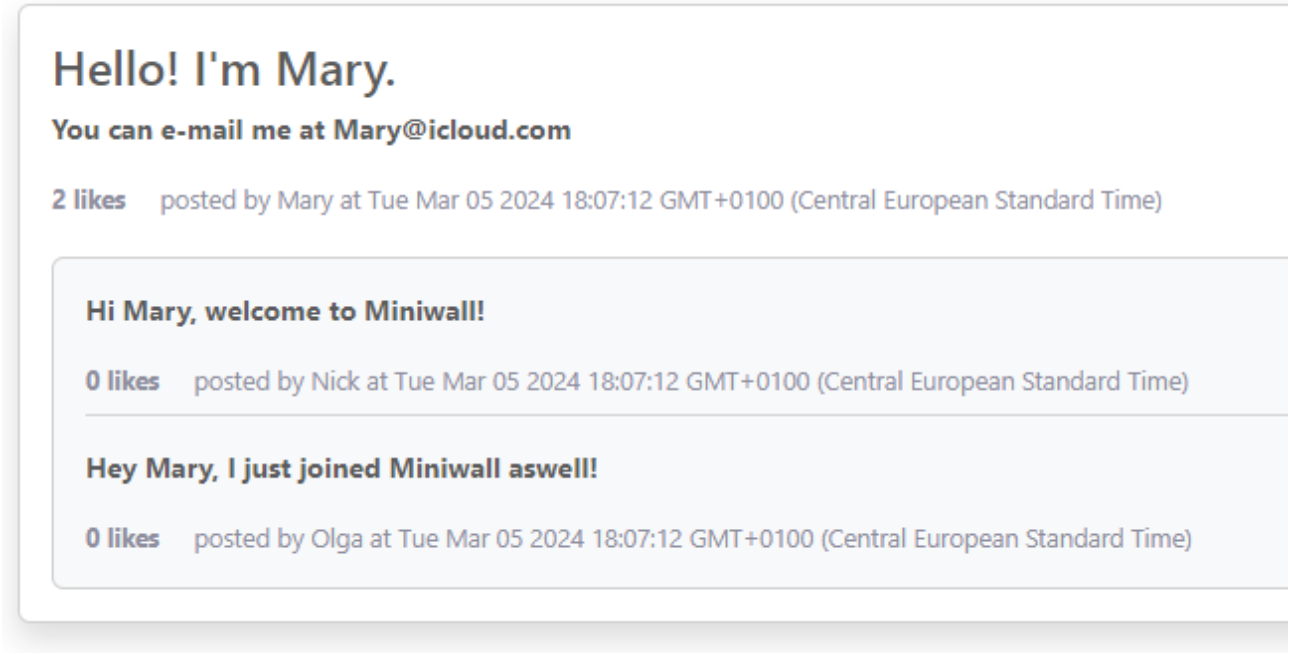
```
// TC 12: Nick and Olga like Mary’s posts.
it('Nick and Olga like Mary’s posts', async () => {

    // Nick likes the post
    const Nick_Likes_Marys_Post = await request(app)
        .patch('/api/posts/' + MarysPostID + "/like")
        .set('auth-token', Credentials_Nick.Token)
        .send({})

    // Verify Nick's like
    expect(Nick_Likes_Marys_Post.body.Likes).toBe(1);

    // Olga likes the post
    const Olga_Likes_Marys_Post = await request(app)
        .patch('/api/posts/' + MarysPostID + "/like")
        .set('auth-token', Credentials_Olga.Token)
        .send({})

    // Verify Olga's like
    expect(Olga_Likes_Marys_Post.body.Likes).toBe(2);
});
```

We send the like requests, which returns the post with the updated like count. We expect the likes to be 1 after Nick's like and 2 after Olga's like.

We can see who liked this post in the Database:



The like count is also visible in the frontend. for privacy reasons the API only returns the amount of likes, not who liked it.



Test Case 13

Mary likes her posts. This call should be unsuccessful; an owner cannot like their posts.

```
// TC 13: Mary likes her post. This call should be unsuccessful; an owner cannot like their posts.
it('Mary likes her post', async () => {

    // Mary Likes the post
    const Mary_Likes_Marys_Post = await request(app)
        .patch('/api/posts/' + MarysPostID + "/like")
        .set('auth-token', Credentials_Mary.Token)
        .send({})

    // Verify Nick's like
    expect(Mary_Likes_Marys_Post.body.message).toStrictEqual("You can't like this post - you are the owner!");
});
```

We Send a like using Marys credentials. The frontend and the server still look the same as in Test Case 12, no change was made to the like count. Mary receives the error printed above.

Test Case 14

Mary can see that there are two likes in her posts.

```
You, 3 weeks ago • Added remaining test cases
// TC 14: Mary can see that there are two likes in her posts.
it('Mary checks her likes', async () => {

    // Mary access her own post
    const mary_check_own_post = await request(app)
        .get('/api/posts/' + MarysPostID)
        .set('auth-token', Credentials_Mary.Token)
        .send({})

    // Mary checks if Nick & Olga have liked her post
    expect(mary_check_own_post.body.Likes).toBe(2);
});
```

The likes are visible in the screenshot of case 12, here we verify them again using the API.

Test Case 15

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

```
// TC 15:  Nick can see the list of posts, since Mary's post has two likes it is shown at the top.
it('Nick checks list of posts', async () => {

    // Nick browses the available posts
    const nick_all_posts = await request(app)
        .get('/api/posts/')
        .set('auth-token', Credentials_Nick.Token)
        .send({})

    // Nick checks that there are exactly 3 posts in the array
    expect(nick_all_posts.body.length).toBe(3);

    // check order of posts
    for (let i = 1; i < nick_all_posts.body.length; i++) {

        if (nick_all_posts.body[i - 1].Likes == nick_all_posts.body[i].Likes) {

            // If the like amount is identical, check if the newest is shown first
            const previous = new Date(nick_all_posts.body[i - 1].Timestamp).getTime();
            const current = new Date(nick_all_posts.body[i].Timestamp).getTime();
            expect(previous).toBeLessThanOrEqual(current); // Previous should have less time

        } else {

            // If the like amount is identical, verify the highest like count is first
            const previous = nick_all_posts.body[i - 1].Likes;
            const current = nick_all_posts.body[i].Likes;
            expect(previous).toBeGreaterThanOrEqual(current); // Previous should have more likes
        }
    }
});
```

We iterate over all visible posts to verify the correct order.

8. Deployment

Finally, the MiniWall project was deployed into a GCP VM Docker Container using GitHub and Google Cloud.

```
kai_benjamin_b@cloud-vm-2:~$ git clone git@github.com:University-of-London/csm020-coursework-KaiBowers99.git
Cloning into 'csm020-coursework-KaiBowers99'...
Warning: Permanently added the ECDSA host key for IP address '140.82.114.3' to the list of known hosts.
Enter passphrase for key '/home/kai_benjamin_b/.ssh/id_ed25519':
remote: Enumerating objects: 152, done.
remote: Counting objects: 100% (152/152), done.
remote: Compressing objects: 100% (80/80), done.
remote: Total 152 (delta 74), reused 126 (delta 48), pack-reused 0
Receiving objects: 100% (152/152), 125.00 KiB | 4.31 MiB/s, done.
Resolving deltas: 100% (74/74), done.
```

To do this, a Dockerfile was created in the Google Cloud using ssh. A dockerignore file was created on the server to exclude node_modules.

```
kai_benjamin_b@cloud-vm-2:~$ pico Dockerfile
kai_benjamin_b@cloud-vm-2:~$ cat Dockerfile
FROM alpine
RUN apk add --update nodejs npm
COPY . /src
WORKDIR /src
EXPOSE 3000
ENTRYPOINT ["node", "./app.js"]
```

```
kai_benjamin_b@cloud-vm-2:~/csm020-coursework-KaiBowers99$ docker image build -t mini-wall-run .
[+] Building 0.5s (9/9) FINISHED                                                                                docker:default
 => [internal] load build definition from Dockerfile                                                                      0.0s
 => => transferring dockerfile: 150B                                                                                      0.0s
 => [internal] load .dockerignore                                                                                         0.0s
 => => transferring context: 2B                                                                                           0.0s
 => [internal] load metadata for docker.io/library/alpine:latest                                                          0.3s
 => [1/4] FROM docker.io/library/alpine@sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8e1ad6b           0.0s
 => [internal] load build context                                                                                         0.0s
 => => transferring context: 2.66kB                                                                                       0.0s
 => CACHED [2/4] RUN apk add --update nodejs npm                                                                          0.0s
 => CACHED [3/4] COPY . /src                                                                                              0.0s
 => CACHED [4/4] WORKDIR /src                                                                                             0.0s
 => exporting to image                                                                                                    0.0s
 => => exporting layers                                                                                                   0.0s
 => => writing image sha256:555a742836ec83818c061b28290737887aa0e14b57f3ee56034132f1bf5f8cf3                              0.0s
 => => naming to docker.io/library/mini-wall-run                                                                          0.0s
```

```
kai_benjamin_b@cloud-vm-2:~/csm020-coursework-KaiBowers99$ docker container run -d --name mini-wall-run-1 --publish 8080:3000 mini-wall-run
0c782381669971ab3b3339788a4c4057f38f57867eeda336366170958a2001b9
```

As a final conclusion of the project, we build and execute the Docker image to allow accessibility to MiniWall from anywhere in the world.

9. Quality

To ensure code quality, the “Prettier” plug-in available on the Visual Studio marketplace was used in the last three commits for formatting.