Miniwall Technical report

Name and Surname: Davide Biagio Ferri **Student number:** 13808681

Repository: https://github.com/DavideFerri/MiniWall

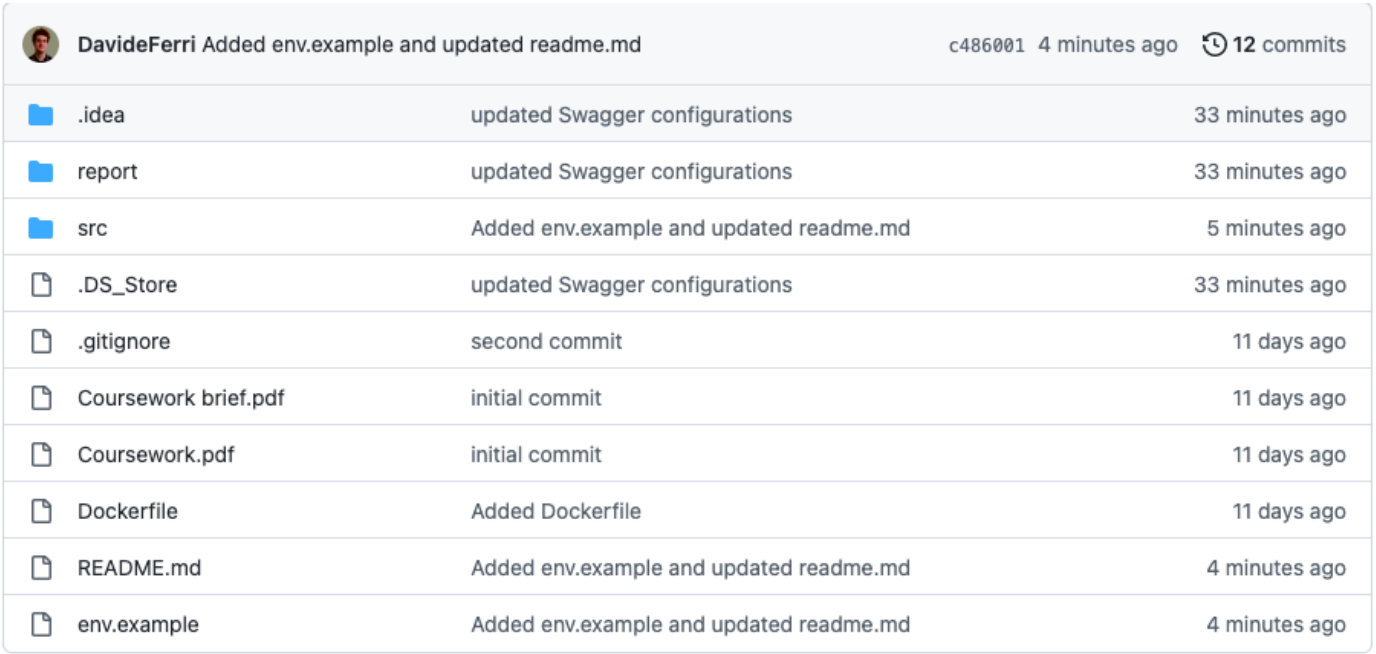
I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

Index

- Project structure and dependancies
- API description
- Database Design
- Deployment using docker
- Tests
- References

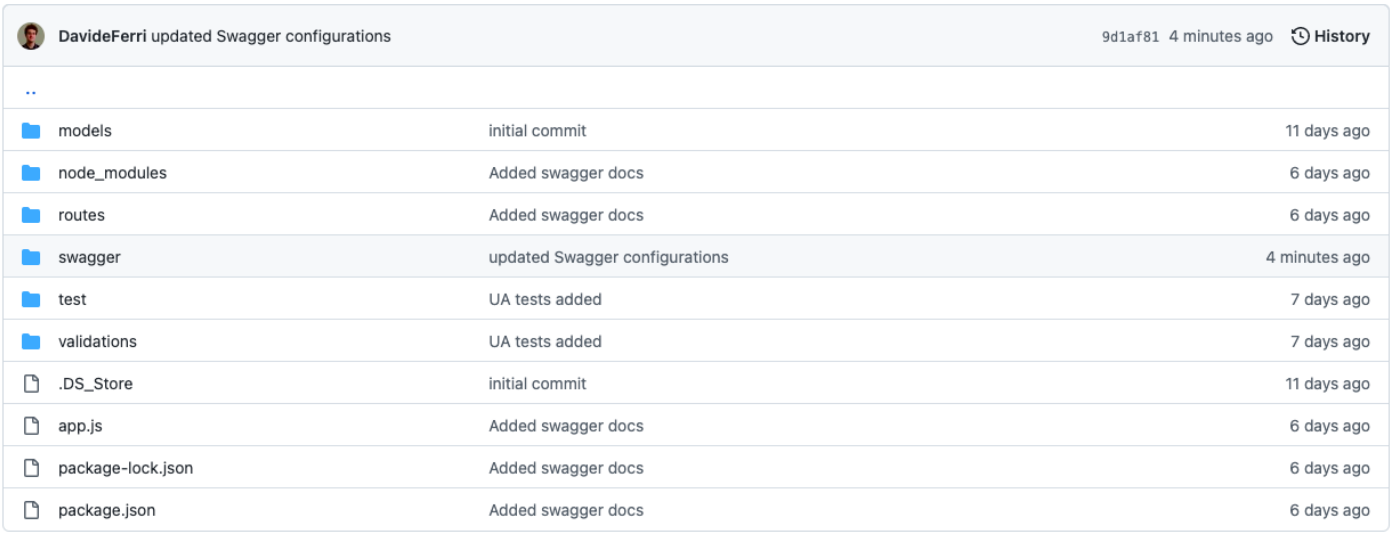
Project structure and dependencies

The following picture provides an overview of the repository structure:



DavideFerri Added env.example and updated readme.md	c486001 4 minutes ago	**12** commits
.idea	updated Swagger configurations	33 minutes ago
report	updated Swagger configurations	33 minutes ago
src	Added env.example and updated readme.md	5 minutes ago
.DS_Store	updated Swagger configurations	33 minutes ago
.gitignore	second commit	11 days ago
Coursework brief.pdf	initial commit	11 days ago
Coursework.pdf	initial commit	11 days ago
Dockerfile	Added Dockerfile	11 days ago
README.md	Added env.example and updated readme.md	4 minutes ago
env.example	Added env.example and updated readme.md	4 minutes ago

Please have a look at README.md to see how the repo works and how to access the API docs. We keep a .env file in local to store the database connection string and token secrets (see env.example). The actual code is stored in the src folder:



Let us discuss this part of the repository more in details:

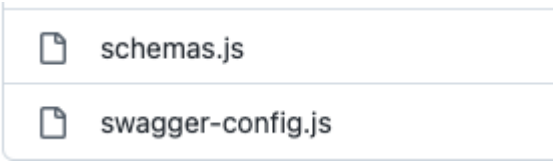
- App.js is the main application file - the server can be started by running npm start on the terminal.
- package.json specifies the project basic information, the main commands (npm start and npm test) as well as the project dependencies. The complete set of dependencies is as follows:

```
"dependencies": {
  "bcryptjs": "^2.4.3",
  "body-parser": "^1.20.1",
  "dotenv": "^16.0.3",
  "express": "^4.18.2",
  "jest": "^29.3.1",
  "joi": "^17.7.0",
  "jsonwebtoken": "^8.5.1",
  "mongoose": "^6.6.7",
  "mongoose-to-swagger": "^1.4.0",
  "nodemon": "^2.0.20",
  "supertest": "^6.3.2",
  "swagger-jsdoc": "^6.2.5",
  "swagger-ui-express": "^4.6.0"
}
```

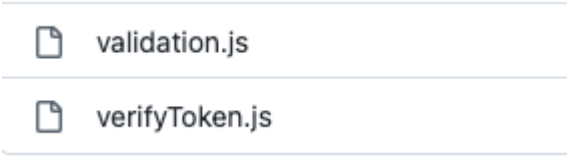
- models/ contains all mongoose models.

- routes/ contains the code for the different API resources and requests

- swagger contains the models and configurations needed for setting up a simple client with Swagger, which we use to build a dynamic documentation for the API (to access it run npm start in local, and then server_path_and_port/api-docs)

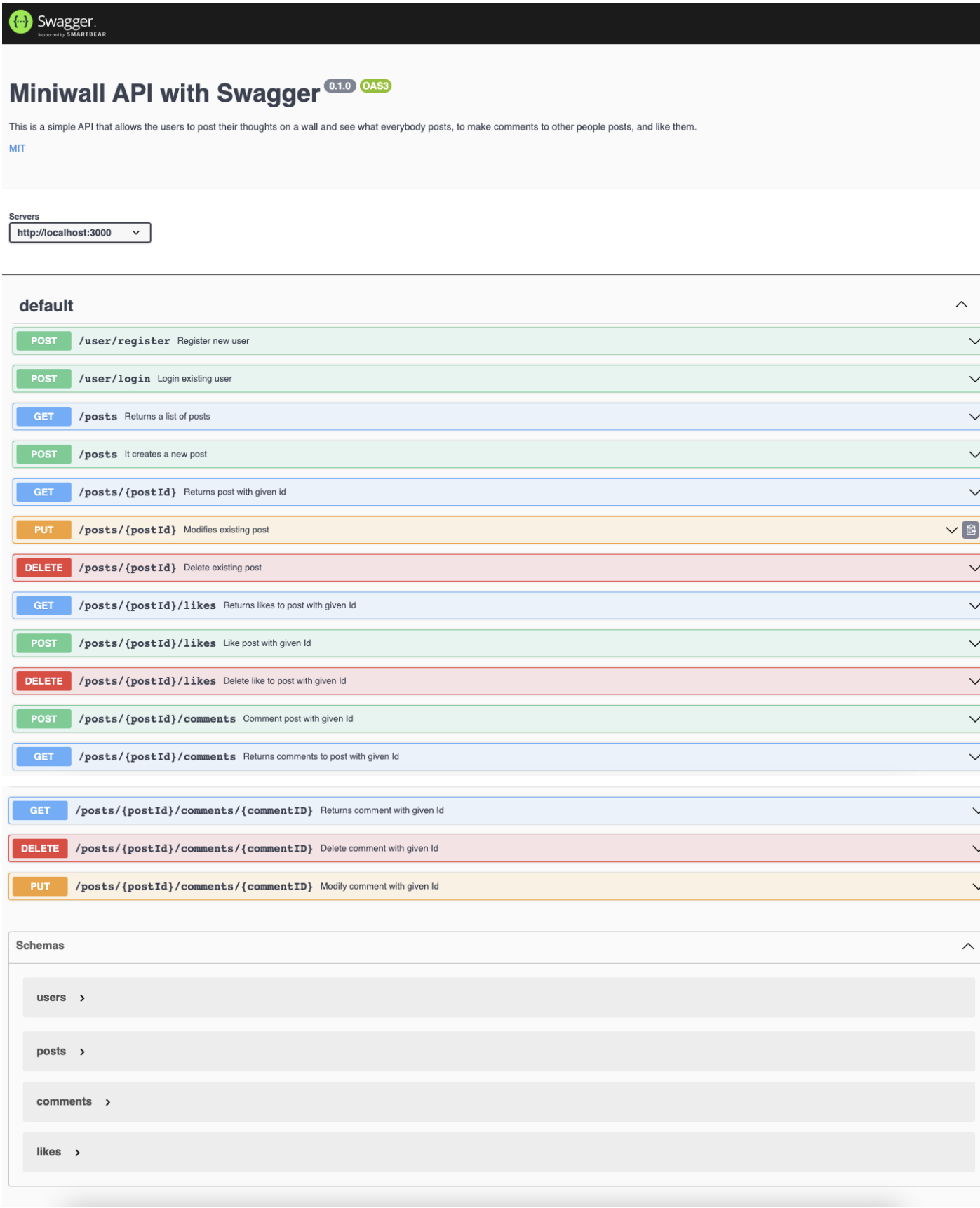


- test/ contains the UAT for the project. To run the tests use npm test
- validations/ contains the code we use to validate the user inputs to the API. In particular, validation.js provides validation functions for user registration, login, post and comments creation, while verifyToken.js makes the use of external modules to validate user tokens



API description

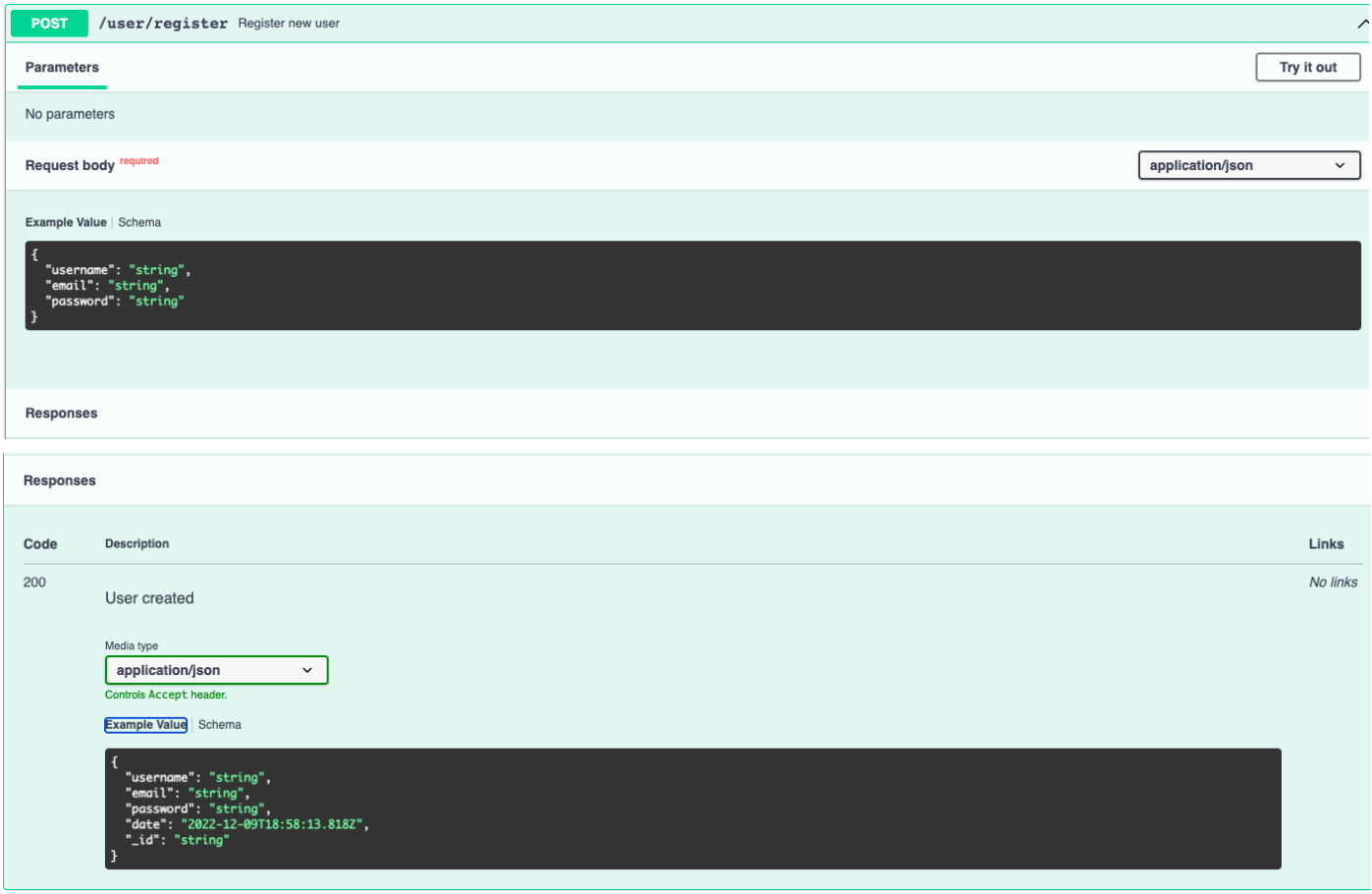
I have built an API client with Swagger to document my Miniwall API. The following picture provides an overview of the service:



I will now explain how each endpoint works and what results is supposed to achieve for our user.

Let us start with analysing how the user can register on our Miniwall application and starts using it.

Registration and Login



The POST register endpoint requires the user to provide a username, an email and a password - it then confirms that the operation was successful as well as the details of the new user (hashed password is returned). I have defined a User model which captures the information that should be saved on the user.

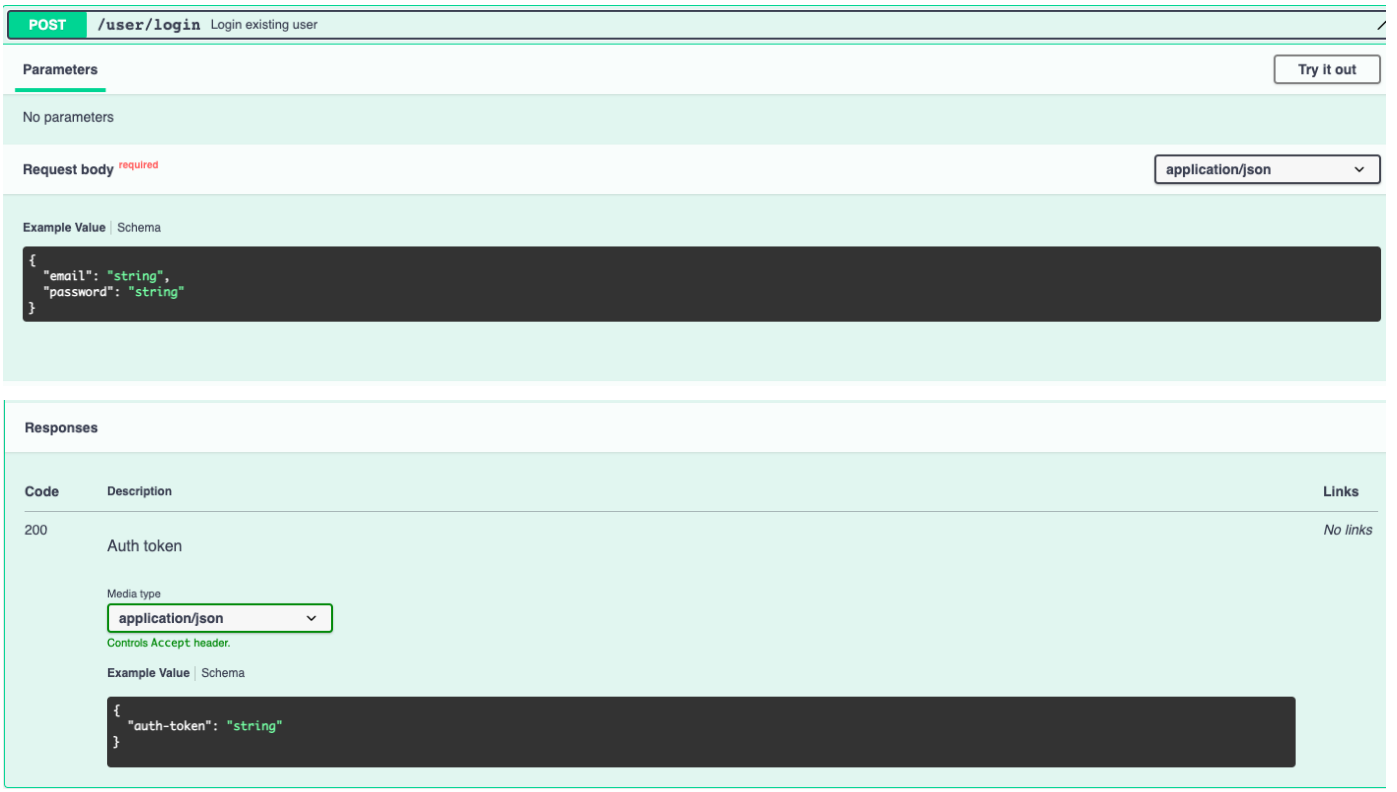
```
users {
    username*           string
    email*              string
    password*           string
    date                string($date-time)
    _id                 string
}
```

The data is saved in a “users” collection. An example of a user record saved in database is offered by the following:

```
_id: ObjectId('638b772e254807116c7338ed')
username: "nick"
email: "nick@gmail.com"
password: "$2a$05$j5A4qH4B3DRA7GHyiXekDeK7NJ17e5YhpaXFjX.YIR0uM8sEBOW02"
date: 2022-12-03T16:19:58.534+00:00
__v: 0
```

$2a$05$j5A4qH4B3DRA7GHyiXekDeK7NJ17e5
YhpaXFjX.YIR0uM8sEBOW02

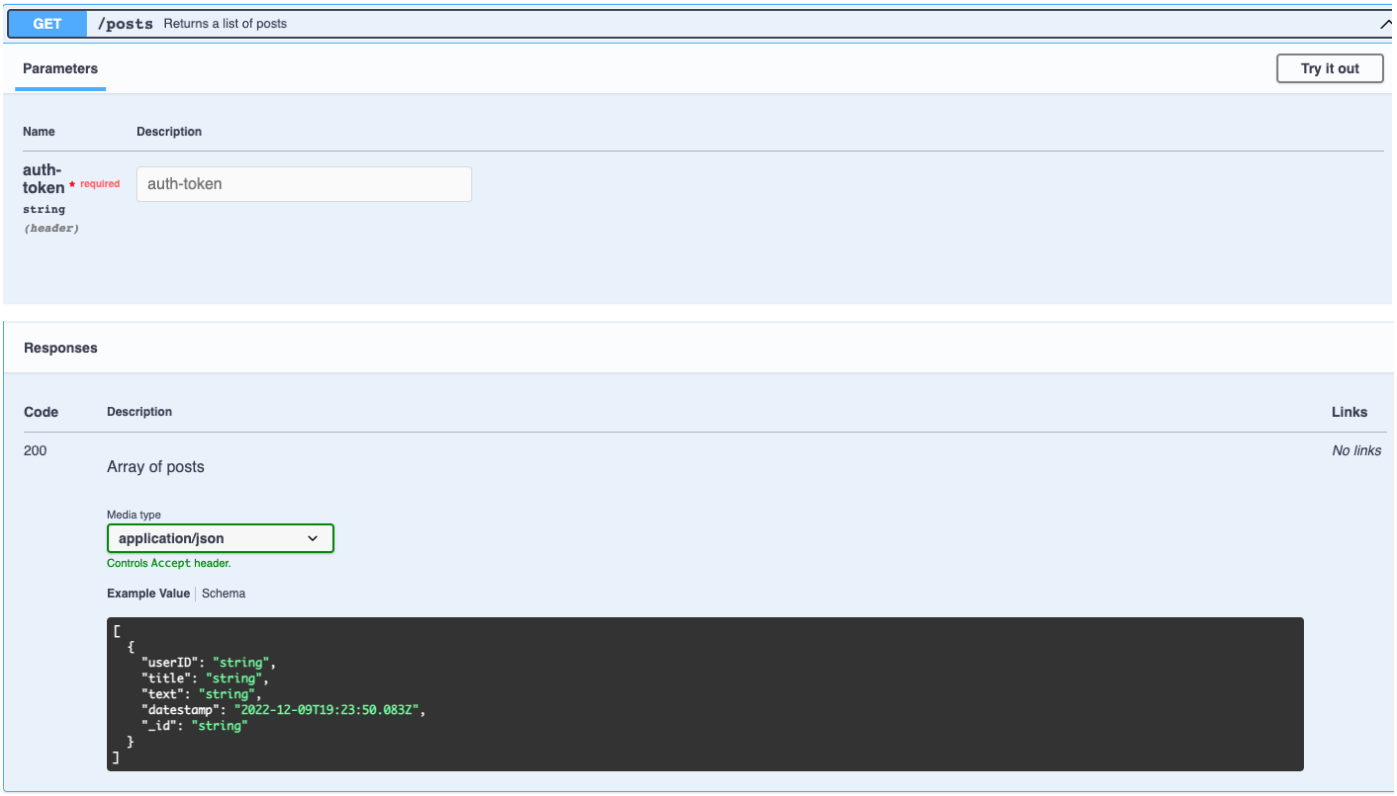
Once the user has registered, they can use their login details to access the application. This is done via the following:



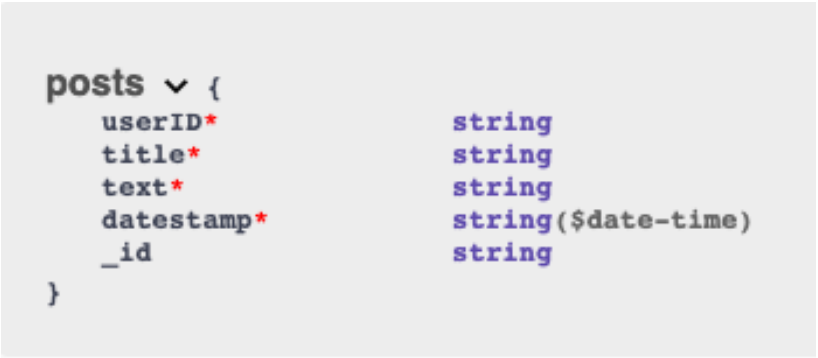
Our user needs to specify their email and password to get an authorization token, which can then be used as a header-parameter in all other API endpoints to gain the necessary authorization to access the content.

Create, modify, see and delete posts

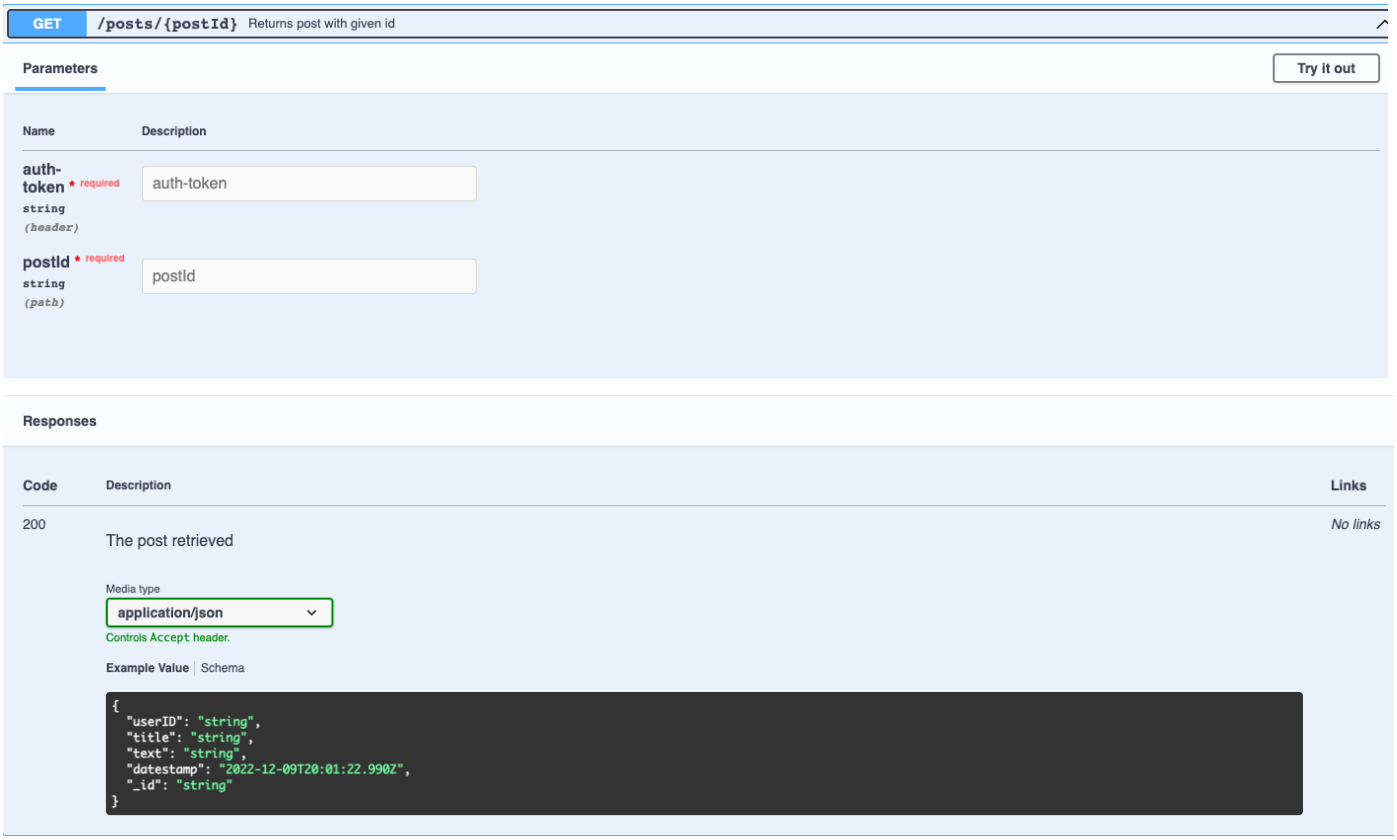
A logged-in user can see all posts on the wall, ordered first by number of likes and then chronologically. To see all posts the user needs to issue a GET request to the API:



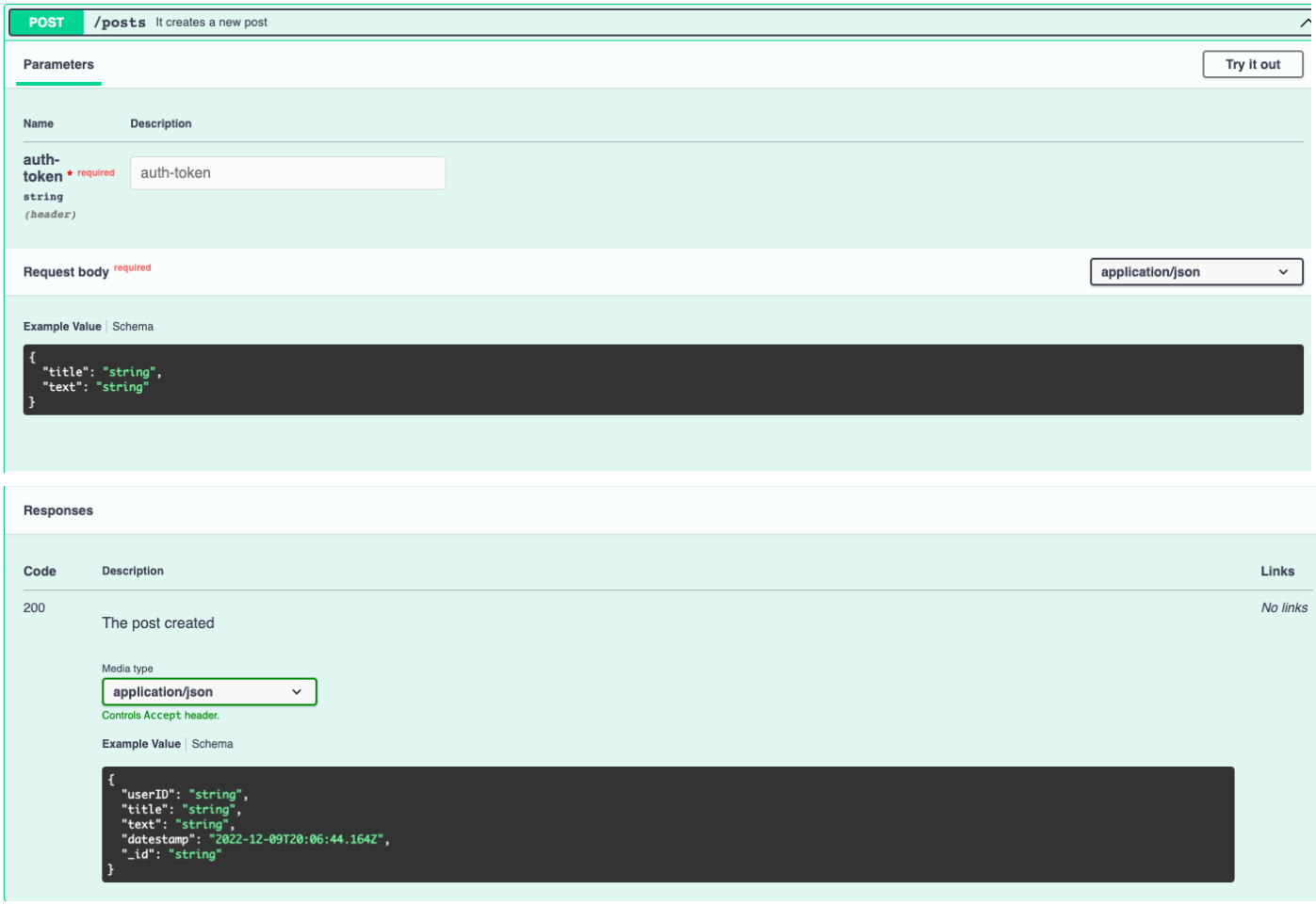
Posts need to follow a specific format, and are then saved in our database in the “posts” collection.



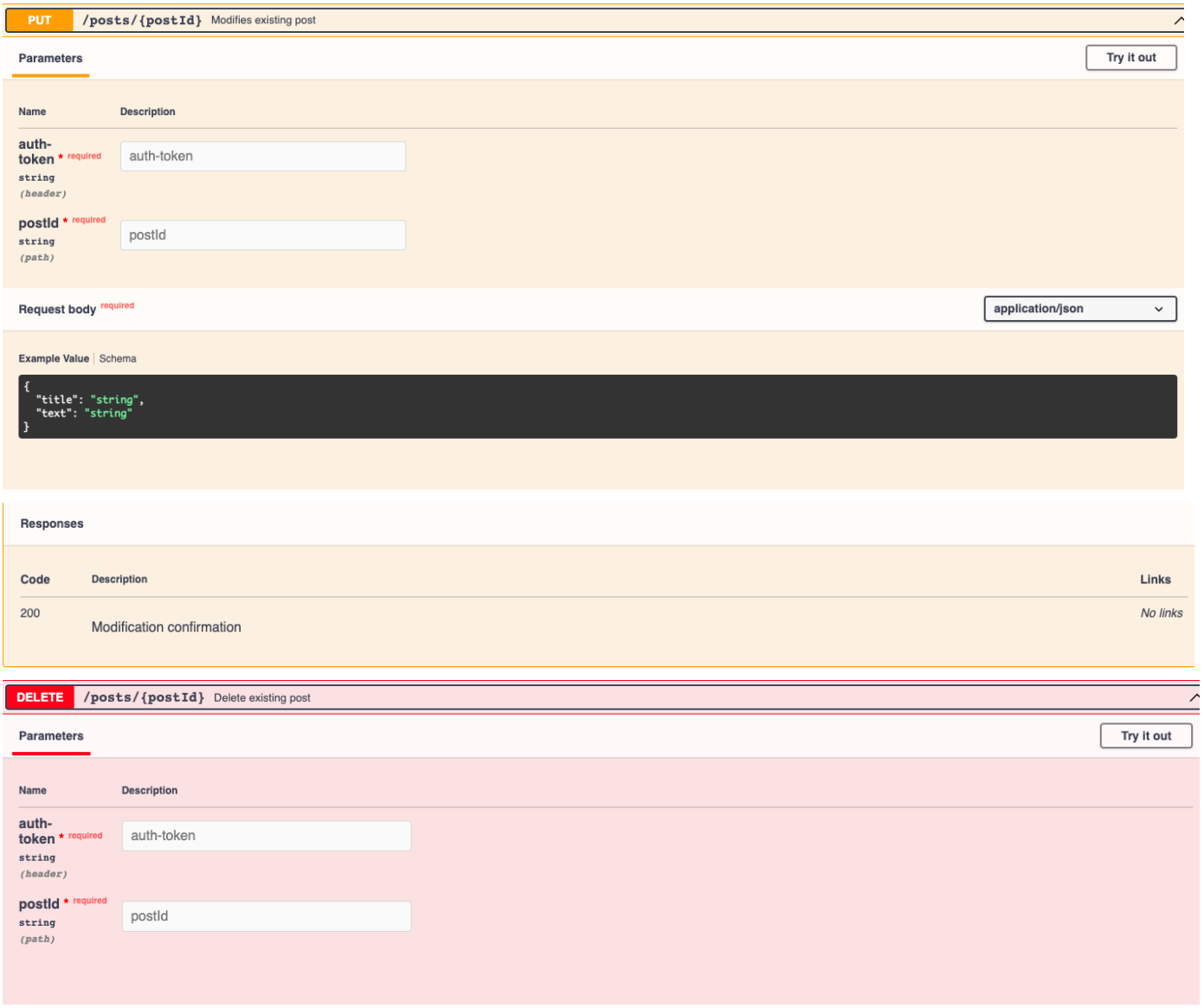
The user can also see a specific post by providing the post ID, as the following shows:



The Miniwall application lets the user interact with the wall in a number of different ways: first of all, it lets users write their posts, modify and delete them. To write a post, the user needs to provide a post title and a corpus of text

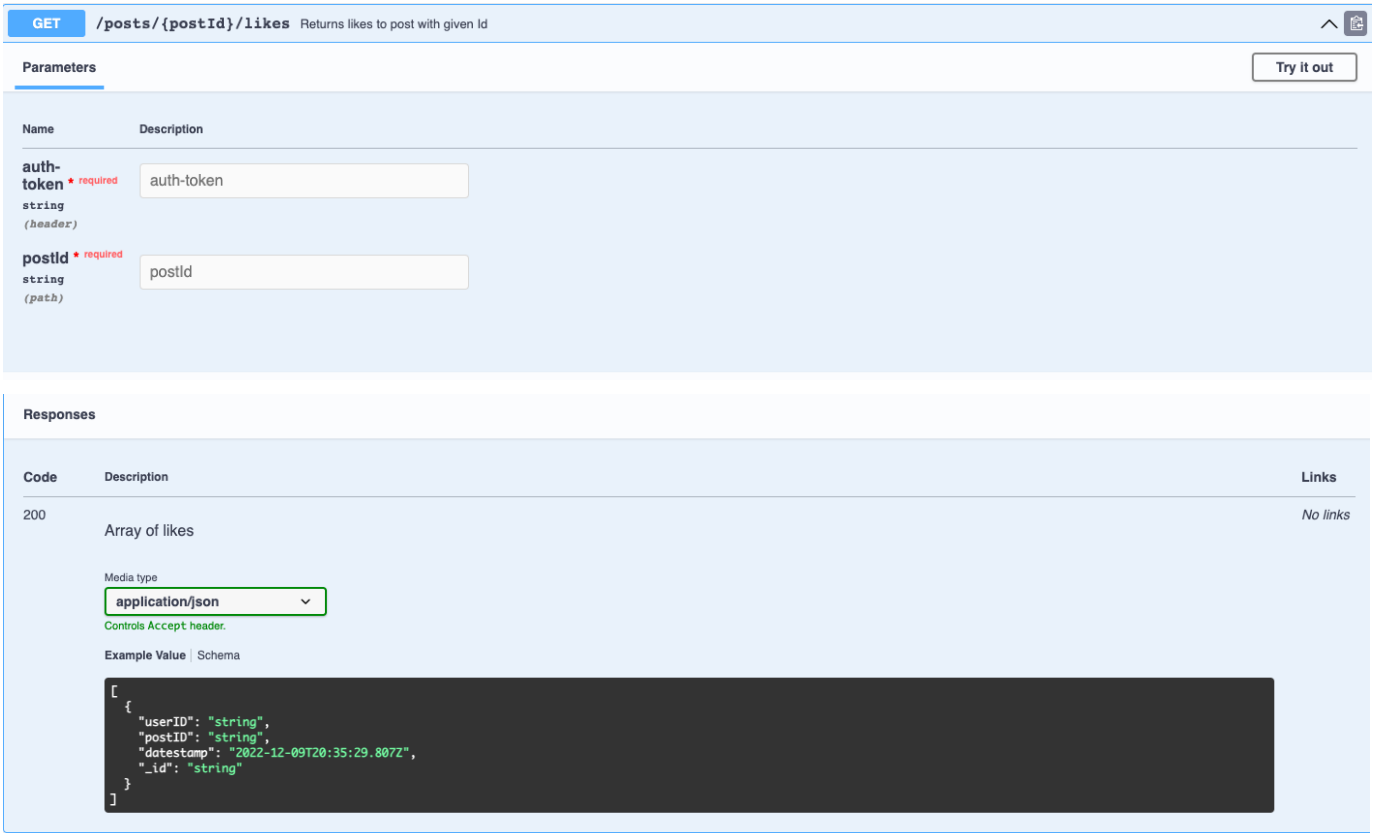


As a result, the user gets the confirmation of success and the actual post saved in the database. The PUT AND DELETE calls allow the user to respectively modify and delete one of their posts: please not that it is not possible to modify and delete other users' posts.

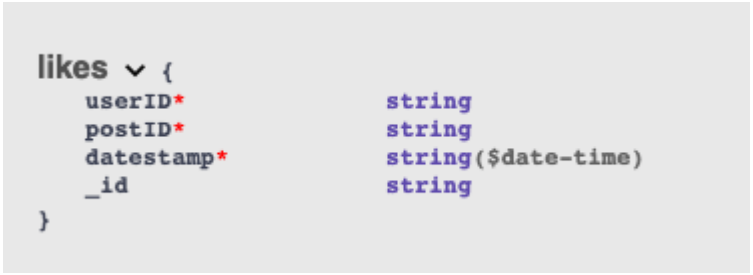


Like posts

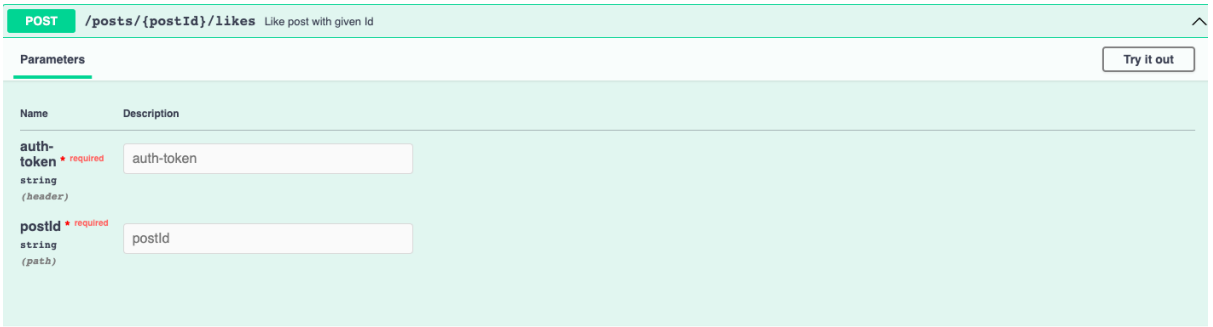
Every Miniwall API user can like other users' posts - the liking operation is reversible, that is it is always possible to delete a like. But first, let's see how a user can see all the likes to a post.

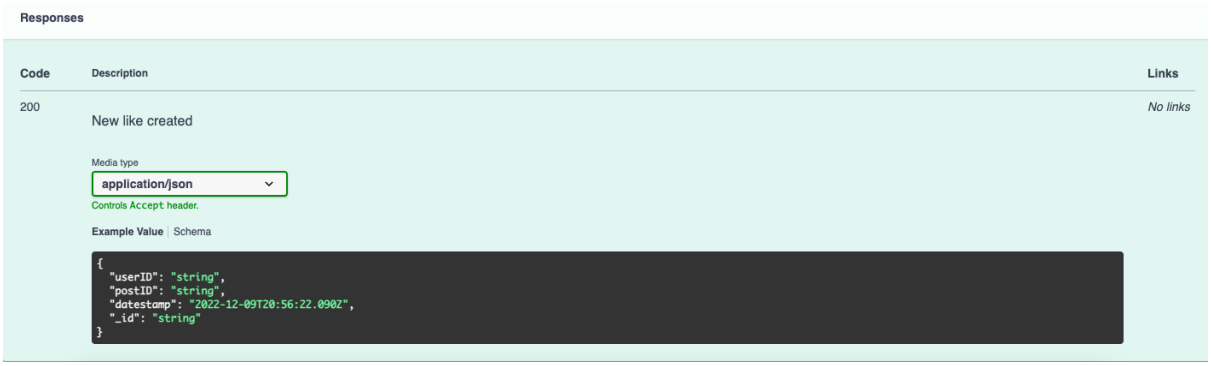


It suffices to specify its postID (together with an authorization token) to see all likes received by a post. In particular, the API calls returns a list of likes, where the likes follow this scheme:



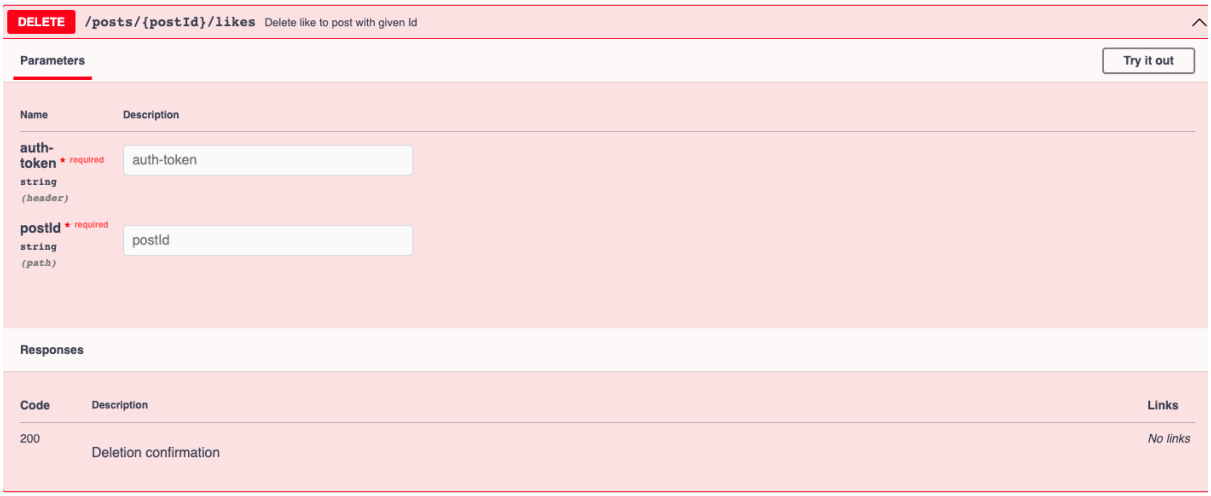
The user can issue a POST request to like another user's post. It is not possible to like one's own posts.





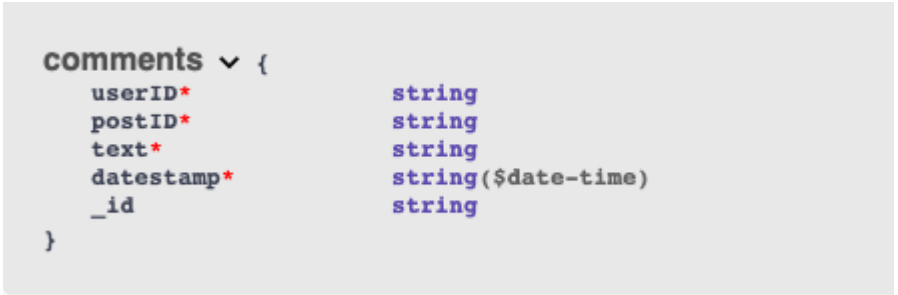
The API call returns a confirmation of success and the like created. Likes are saved in the “likes” collection in the database.

The DELETE operation on the likes endpoint will allow the use to delete a previously created like to a post

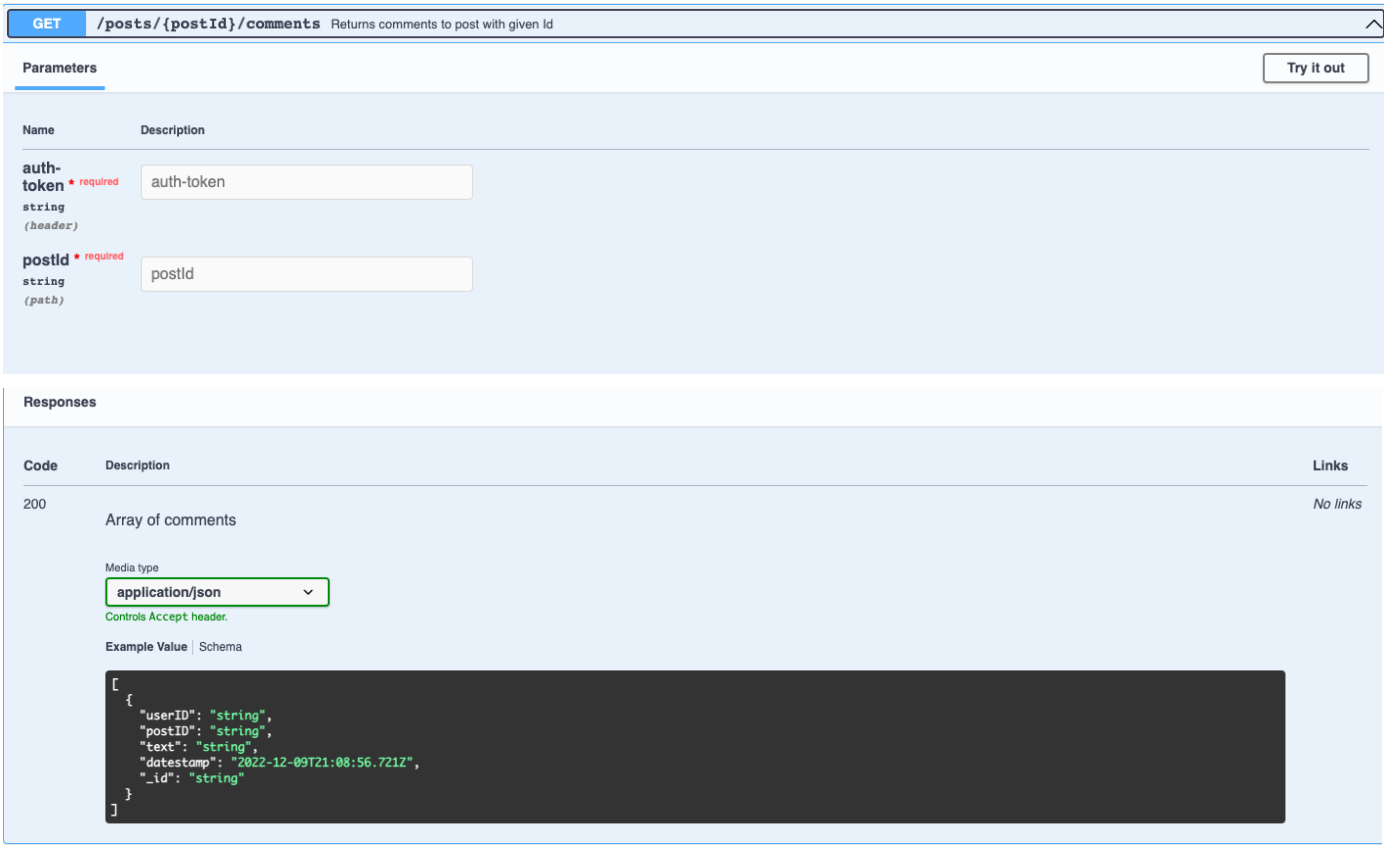


Comment posts

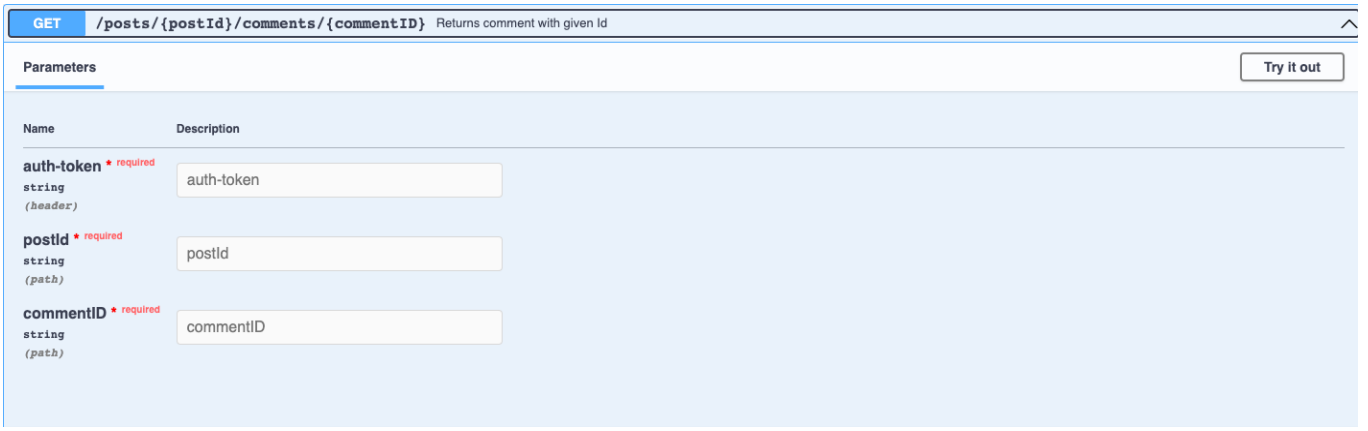
Finally, users can comment on other users’ posts. Please note that in this simple application it is not possible to make comments on other comments or simply to comment one’s own posts. A comment is nothing but some text referring to a comment. The full set of data stored for each comment is described as follows:

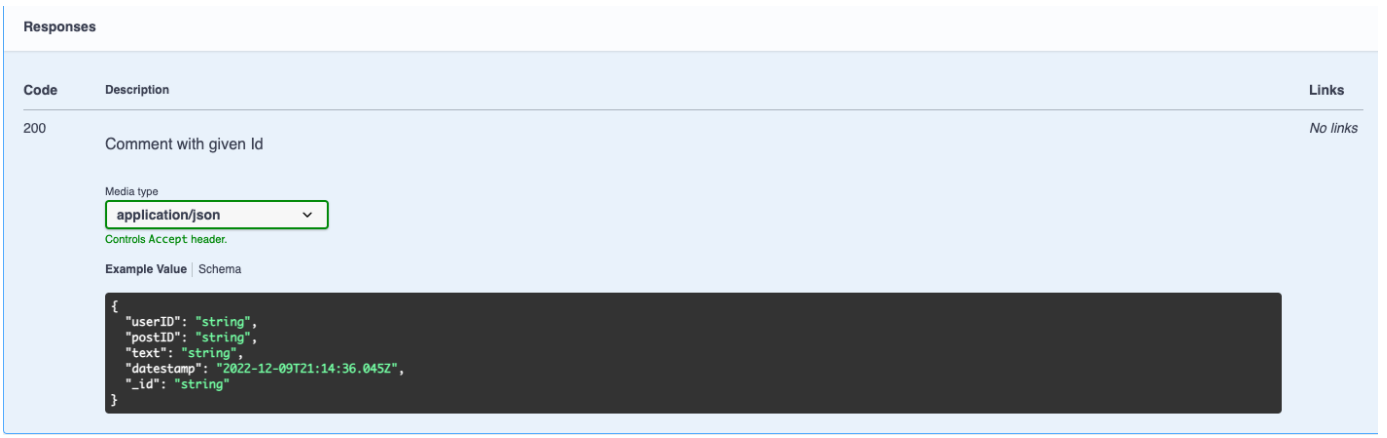


All comments are stored in the database in the “comments” collection. First, a user can see all comments to a given post by means of a GET request:

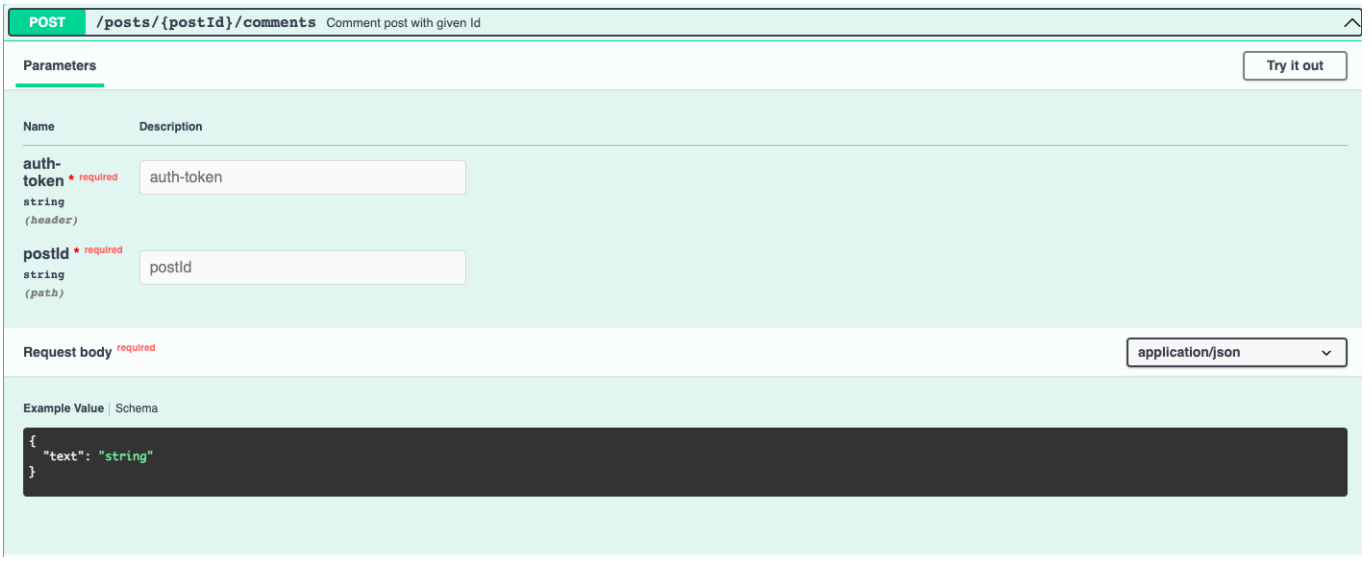


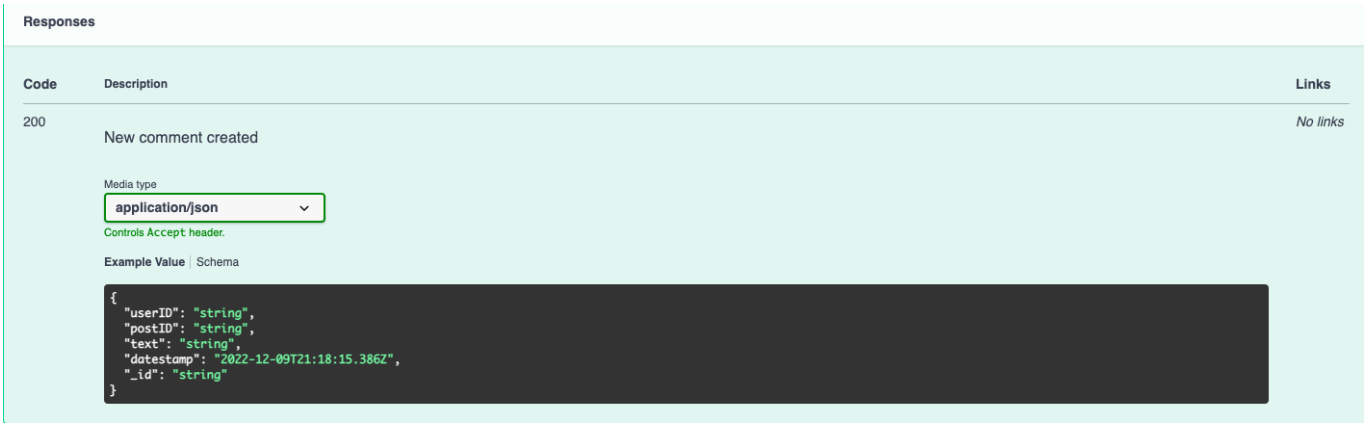
Again, the user just needs provide a valid auth token and post ID to receive back a list of likes to that post. One could also provide a comment ID to focus on one particular comment only:



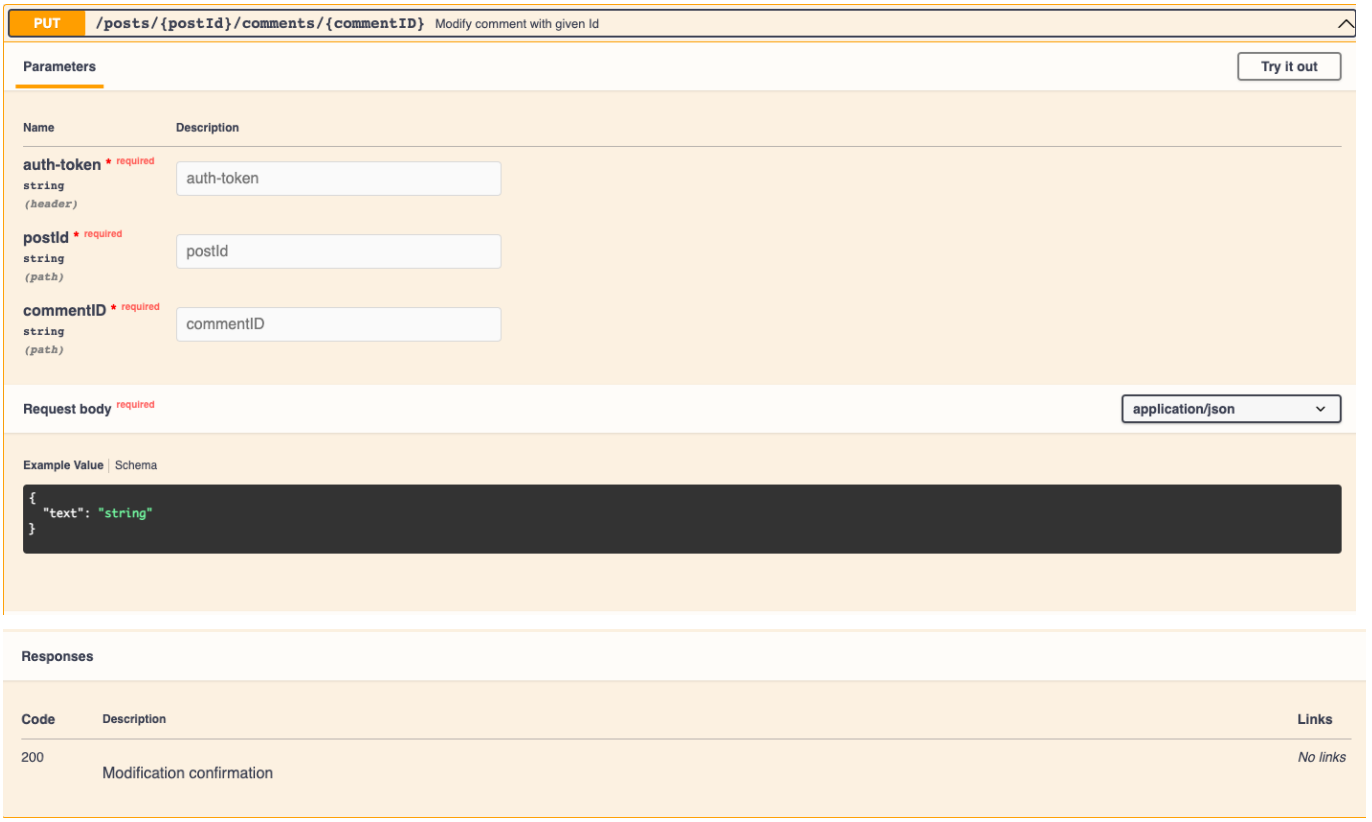


To comment on a post it just suffices to specify some text and the post ID; The API endpoint returns a confirmation of success as well as the actual comment created.





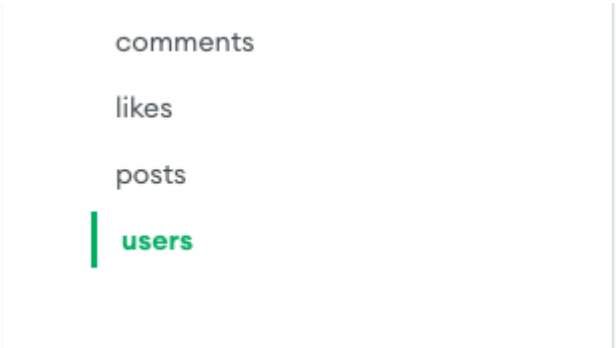
Miniwall users always have the opportunity to modify or delete their comments. Obviously, no user can modify or delete another user's comments. The PUT endpoint allows the user to edit a comment previously created:



If the operation is successful, a confirmation of success is sent to the client. Similarly, users can delete their own posts:

Database Design

The following picture offers an overview of all existing collections in our database:



The choice of collections reflects the conceptual structure of the application. Let us now discuss some of the choices we have made:

- For the users we store their email, username, password and date of registration. The simple functionalities required in the Miniwall application do not seem to require anything else; however, one could think of adding some user-specific settings in more sophisticated versions of the application.

```
_id: ObjectId('638b772e254807116c7338ed')
username: "nick"
email: "nick@gmail.com"
password: "$2a$05$j5A4qH4B3DRA7GHyiXekDeK7NJ17e5YhpaXFjX.YIR0uM8sEBOW02"
date: 2022-12-03T16:19:58.534+00:00
__v: 0
```

- For the posts we store the userID, postID, title, text and date of creation. Posts could also show images / report links - in that case, we would need to enrich the model in the future

```
_id: ObjectId('638b772f254807116c7338f5')
userID: "638b772e254807116c7338eb"
title: "Just woke up happy"
text: "Hello, how are all my friends today? I feel happy"
datestamp: 2022-12-03T16:19:59.485+00:00
__v: 0
```

- Likes are very basic for now and we store the userID, postID, likeID and date of creation. In future versions of the application we could develop a wider range of reactions to a post, so that the model would need to specify how the user reacted to the post (or build one model for each reaction, which would likely be an overkill)

```
_id: ObjectId('638b77312548071l6c733908')
userID: "638b772e254807116c7338eb"
postID: "638b7730254807116c7338f9"
datestamp: 2022-12-03T16:20:01.367+00:00
__v: 0
```

- For comments we store the userID, postID and commentID together with a text and date of creation. In future versions of the application we might want to give users the opportunity to make comments on comments - in that case, the postID could generalize and refer either to a post or to another comment.

```
_id: ObjectId('638b77302548071l6c7338fe')
userID: "638b772e254807116c7338eb"
postID: "638b77302548071l6c7338f9"
text: "Hi Mary, I feel happy today!"
datestamp: 2022-12-03T16:20:00.542+00:00
__v: 0
```

- To store comments and likes we have chosen a relational model over a tree-structure for no particular reason other than that of keeping all documents simple to read and analyse. This is particularly valuable when the user wants to inspect posts. Clearly, as comments and likes refer to one and only one post, they could have well been stored in the post document directly. However, if in the future we were to allow the users to comment on other people's comments as well, each post document would have to manage a significant complexity: by storing posts and likes in separate collections we gain some flexibility for future enhancements.

Deployment using Docker

The repository contains the information necessary to build a docker image in the Dockerfile

6 lines (6 sloc) | 125 Bytes

```
1  FROM alpine
2  RUN apk add --update nodejs npm
3  COPY . /workdir
4  WORKDIR /workdir
5  EXPOSE 3000
6  ENTRYPOINT ["node", "./src/app.js"]
```

The GCP VM used for deployment has the following details

Basic information

Name	miniwall
Instance ID	7724081454726773128
Description	None
Type	Instance
Status	Running
Creation time	Nov 29, 2022, 6:54:48 pm UTC+01:00
Zone	northamerica-northeast2-a
Instance template	None
In use by	None
Reservations	Automatically choose
Labels	None
Tags	–
Deletion protection	Disabled
Confidential VM service	Disabled
Preserved state size	0 GB

Machine configuration

Machine type	e2-medium
CPU platform	Intel Broadwell
Architecture	x86/64
vCPUs to core ratio	–
Custom visible cores	–
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

Network interfaces

Subnetwork	Primary internal IP address	Alias IP ranges	Stack type	External IP address	Network tier	IP forwardin
default	10.188.0.2		IPv4	34.130.11.127 (Ephemeral)	Premium	Off

Storage

Boot disk

Name	Image	Interface type	Size (GB)	Device name	Type	Architecture	Encryption	Mode	Wh
miniwall	ubuntu-1804-bionic-v20221125	SCSI	10	miniwall	Balanced persistent disk	x86/64	Google-managed	Boot, read/write	Del

After cloning the MiniWall repository in the GCP VM by “git clone https://github.com/DavideFerri/MiniWall.git”
the Docker image can be created by running “docker image build -t miniwall-image:1 .”, which uses the Dockerfile above to build a Docker image.

```
docker-user@miniwall:~/MiniWall$ docker image build -t miniwall-image:1 .
Sending build context to Docker daemon  73.15MB
Step 1/6 : FROM alpine
 ---> 49176f190c7e
Step 2/6 : RUN apk add --update nodejs npm
 ---> Using cache
 ---> bae33e864d61
Step 3/6 : COPY . /workdir
 ---> 8b208a005298
Step 4/6 : WORKDIR /workdir
 ---> Running in cd06caelaedd
Removing intermediate container cd06caelaedd
 ---> bf7f4f184464
Step 5/6 : EXPOSE 3000
 ---> Running in e17b05cec316
Removing intermediate container e17b05cec316
 ---> b5be8f59e237
Step 6/6 : ENTRYPOINT ["node", "src/app.js"]
 ---> Running in 71b4053a71d2
Removing intermediate container 71b4053a71d2
 ---> 5de7410722df
Successfully built 5de7410722df
Successfully tagged miniwall-image:1
```

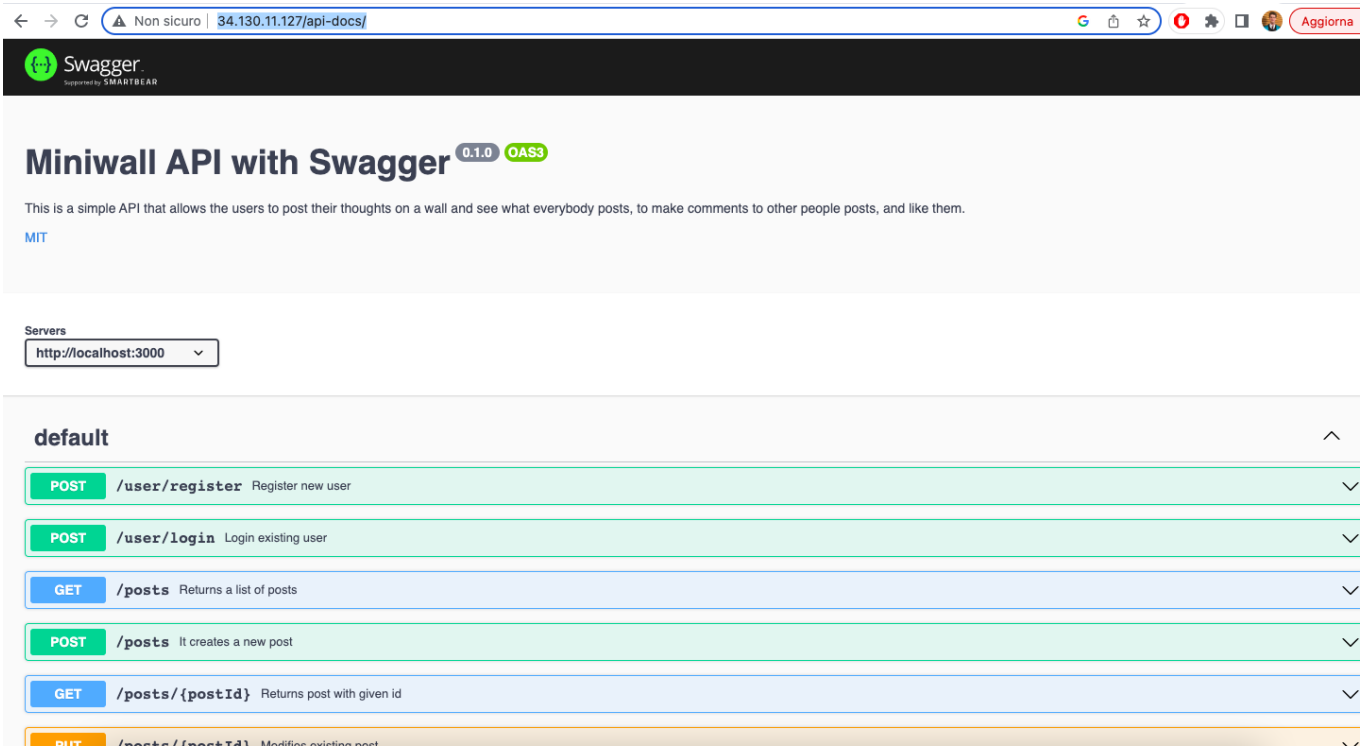
The list of Docker images can be seen by running “docker images”

```
docker-user@miniwall:~/MiniWall$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED              SIZE
miniwall-image   1         5de7410722df   About a minute ago   132MB
<none>           <none>    85bcfc421d6a   10 days ago          88.1MB
alpine           latest    49176f190c7e   2 weeks ago          7.05MB
```

Finally one can start the Docker container by running the following command: “docker container run -d --name miniwall-container --publish 80:3000 miniwall-image:1”

```
docker-user@miniwall:~/MiniWall$ docker container run -d --name miniwall-container --publish 80:3000 miniwall-i
mage:1
421d45157b65cc56f3c087ab940ba286247c2ffc926cdc902514b9a204e39bf5
```

One can easily check that the operation was successful (if the GCP server is up and running) by visiting the following address: http://34.130.11.127/api-docs/ , as the following screenshot shows:



Tests

We use Jest to write the tests for this project. The test can be run in local by “npm test”. There are 15 test cases, which correspond to those proposed in the coursework document. Here are the screenshots of the test cases:

```
test("TC1",async ()=>{
    // register all users
    await Promise.all(users.map( async user=>{
        await request('http://localhost:3000').post('/user/register').send({"username": user.username,
                                                                            "email": user.email,
                                                                            "password": user.password}).expect(200
    }))
})

test("TC2", async ()=>{
    // login users
    await Promise.all(users.map(async user=>{
        const res = await request('http://localhost:3000').post('/user/login').send({
                                                                                "email": user.email,
                                                                                "password": user.password
                                                                            })
        expect(res.body['auth-token']).toBeTruthy();
        // store token for each user
        user.token = res.body['auth-token'];
    }))
})

test("TC3",async ()=>{
    // call API without token
    const res = await request('http://localhost:3000').get('/posts')
    expect(res.statusCode).toEqual(401);
})

test("TC4", async ()=>{

    // Olga creates post
    const olga = users[0]
    const res = await request('http://localhost:3000').post('/posts').set("auth-token", olga.token)
        .send({title: 'Just woke up happy',
                text: "Hello, how are all my friends today? I feel happy"})
    olga.postID = res.body._id
    expect(res.statusCode).toEqual(200);
 })
```

```
test("TC5", async ()=>{

   // Nick creates post
   const nick = users[1]
   const res = await request('http://localhost:3000').post('/posts').set("auth-token", nick.token)
       .send({
           title: 'Just woke up sad',
           text: "Hello, how are all my friends today? I feel sad"
       })
   nick.postID = res.body._id
   expect(res.statusCode).toEqual(200);
})

test("TC6", async ()=>{

   // Mary creates post
   const mary = users[2]
   const res = await request('http://localhost:3000').post('/posts').set("auth-token", mary.token).send({
           title: 'Just woke up mini',
           text: "Hello, how are all my friends today? I feel mini"
       })
   mary.postID = res.body._id
   expect(res.statusCode).toEqual(200);
})
```

```
test("TC7", async ()=>{

    // Olga sees all posts in chronological order
    const olga = users[0]
    const res = await request('http://localhost:3000').get('/posts').set("auth-token", olga.token)
    expect(res.statusCode).toEqual(200);
    expect(res.body.length).toEqual(3)
    expect(new Date(res.body[0].datestamp) > new Date(res.body[1].datestamp)).toBeTruthy()
    expect(new Date(res.body[0].datestamp) > new Date(res.body[2].datestamp)).toBeTruthy()

    // Nick sees all posts in chronological order
    const nick = users[1]
    const res2 = await request('http://localhost:3000').get('/posts').set("auth-token", nick.token)
    expect(res2.statusCode).toEqual(200);
    expect(res2.body.length).toEqual(3)
    expect(new Date(res2.body[0].datestamp) > new Date(res2.body[1].datestamp)).toBeTruthy()
    expect(new Date(res2.body[0].datestamp) > new Date(res2.body[2].datestamp)).toBeTruthy()
})

test("TC8", async ()=>{

    const olga = users[0]
    const nick = users[1]
    const mary = users[2]

    // Olga comments Mary's post
    const res = await request('http://localhost:3000').post(`/posts/${mary.postID}/comments`)
        .send({"text": 'Hi Mary, I feel happy today!'}).set("auth-token", olga.token)
    expect(res.statusCode).toEqual(200);

    // Nick comments Mary's post
    const res2 = await request('http://localhost:3000').post(`/posts/${mary.postID}/comments`)
        .send({"text": 'Hi Mary, I feel sad today!'}).set("auth-token", nick.token)
    expect(res2.statusCode).toEqual(200);
})
```

```
test("TC9",async ()=>{

    const mary = users[2]
    // Mary cannot comment her own posts
    const res = await request('http://localhost:3000').post(`/posts/${mary.postID}/comments`)
        .send({"text": 'Hi Mary, I feel mini today!'}).set("auth-token", mary.token)
    expect(res.statusCode).toEqual(400);
})

test("TC10",async ()=>{

    const mary = users[2]
    const res = await request('http://localhost:3000').get('/posts').set("auth-token", mary.token)
    // Mary can see the posts in chronological order
    expect(res.statusCode).toEqual(200);
    expect(res.body.length).toEqual(3);
    expect(new Date(res.body[0].datestamp) > new Date(res.body[1].datestamp)).toBeTruthy();
    expect(new Date(res.body[1].datestamp) > new Date(res.body[2].datestamp)).toBeTruthy();
})

test("TC11",async ()=>{

    const mary = users[2]
    // Mary can see the comments to her post
    const res = await request('http://localhost:3000').get(`/posts/${mary.postID}/comments`).set("auth-token", mary.token)
    expect(res.statusCode).toEqual(200);
    expect(res.body.length).toEqual(2);
})

test("TC12", async ()=>{
    const olga = users[0]
    const nick = users[1]
    const mary = users[2]

    // Olga likes Mary's post
    const res = await request('http://localhost:3000').post(`/posts/${mary.postID}/likes`).set("auth-token", olga.token)
    expect(res.statusCode).toEqual(200);
```

References

https://swagger.io/
https://jestjs.io/
Cloud computing Lab tutorials 1-5
https://www.docker.com/