TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

WeShare

Data Transmissions Project

2023

Team members:

Oprea Sergiu Daniel

Pleșca Evelyn Iulia

Tăslăuan Alexandra

Cuprins

Introduction

Overview

Today, especially since Covid-19, the trend is for people to go online and spend as much time there. Even though that might not be the best solution in some cases, in other cases is the only solution: when we are far from home, far from family and friends and we want to keep in touch with them.

We have created a social media platform designed to connect people, foster collaboration, and inspire meaningful interactions.

Goals

The main goal of our application was to create a digital space that promotes positive engagements, enables authentic connections, and empowers users to make a difference in their communities and beyond.

Functions

One of the features we decided it was crucial working with was the database: we wanted to store the users, posts, comments, and other such activities somewhere where we can access them.

Second of, we conceptualized this application to be as easy as possible to use and to have a great user experience, having the following features:

- Login, Logout and Registration pages: we want to keep your experience tailored to your interests by having a safe account.
- Change profile and cover pictures: let other people see you by changing these representative images of yourself whenever you want to.
- Post something: show the world what are you thinking and feeling.
- Like other people's posts: let people know how much you appreciate seeing their posts.
- Comment on posts: don't be scared to comment on posts and let people know how great it is seeing them.

Application Design

Database Design

The *social* database has six tables, each having an ID number which serves as a primary key and the foreign keys are proof of the relationships between the tables.

The tables used in our application are:

- comments
- likes
- posts
- relationships
- stories
- users

The ***comments*** table contains all the information regarding a comment, such as description, creation date, the ID of the user who left the comment and the ID of the post the comment is on.

The ***likes*** table contains the ID of the user that likes the song and the ID of the liked post.

The ***posts*** table contains the description of the post, the attached image, the creation date and the ID of the user who created the post.

The ***relationships*** table contains the ID of the following user and that of the followed user.

The ***stories*** table contains the attached image and the ID of the user who posted the story.

The ***users*** table contains data about the user, including username, email address, password, name, cover picture, profile picture, city and website.

The detailed design of the table columns and the relationships with other tables can be seen in detail in **Table 1.**

Table 1

Table	Columns	Properties	Relationships
Comments	id [PK] desc createdAt userId postId	int varchar(200) datetime int int	N-to-N with *users* N-to-N with *posts*
Likes	id userId postId	int int int	N-to-N with *posts* N-to-N with *users*
Posts	id desc img userId createdAt	int varchar(200) varchar(200) int datetime	N-to-N with *users*
Relationships	id followerUserId	int int	1-toN with *users*

	followedUserId	int	
Stories	id img userId	int varchar(200) int	N-to-N with *users*

Users	id
	username
	email
	password
	name
	coverPic
	profilePic
	city
	website
		int
		varchar(45)
		varchar(45)
		varchar(200)
		varchar(45)
		varchar(300)
		varchar(300)
		varchar(45)
		varchar(45)
			N-to-N with *comments*
			N-to-N with *likes*
			N-to-N with *posts*
			N-to-N with *relationships*

The relational diagram of the database can be seen in **Fig. 1**:

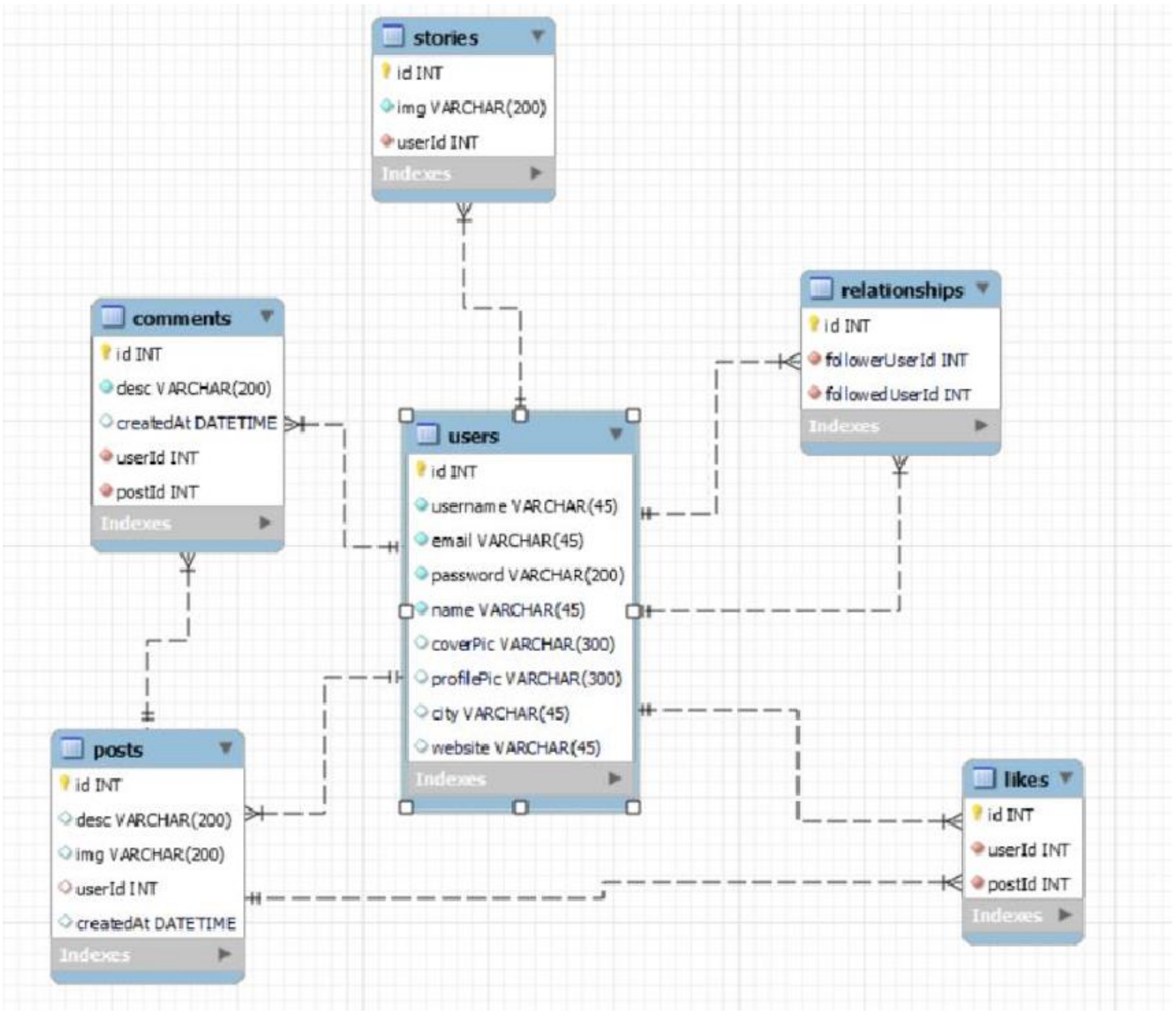


Figure 1

GUI Design

On the starting application, the first page appearing on the WeShare application is the Login page (*Figure 2*) consisting of a form that requires adding your username and password. Then, you can log in pressing the “Login” button. Alternatively, in case you don’t already have an account, you can press the registration button that will redirect you to the Register page (*Figure 4*).

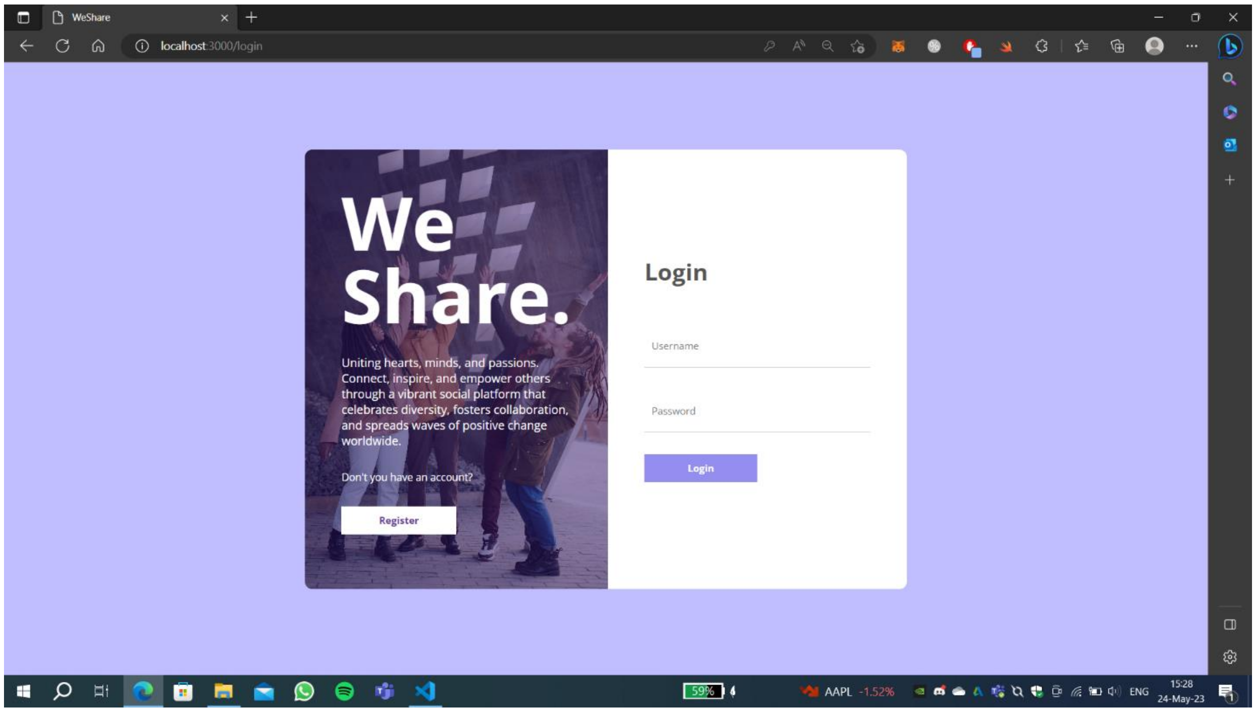
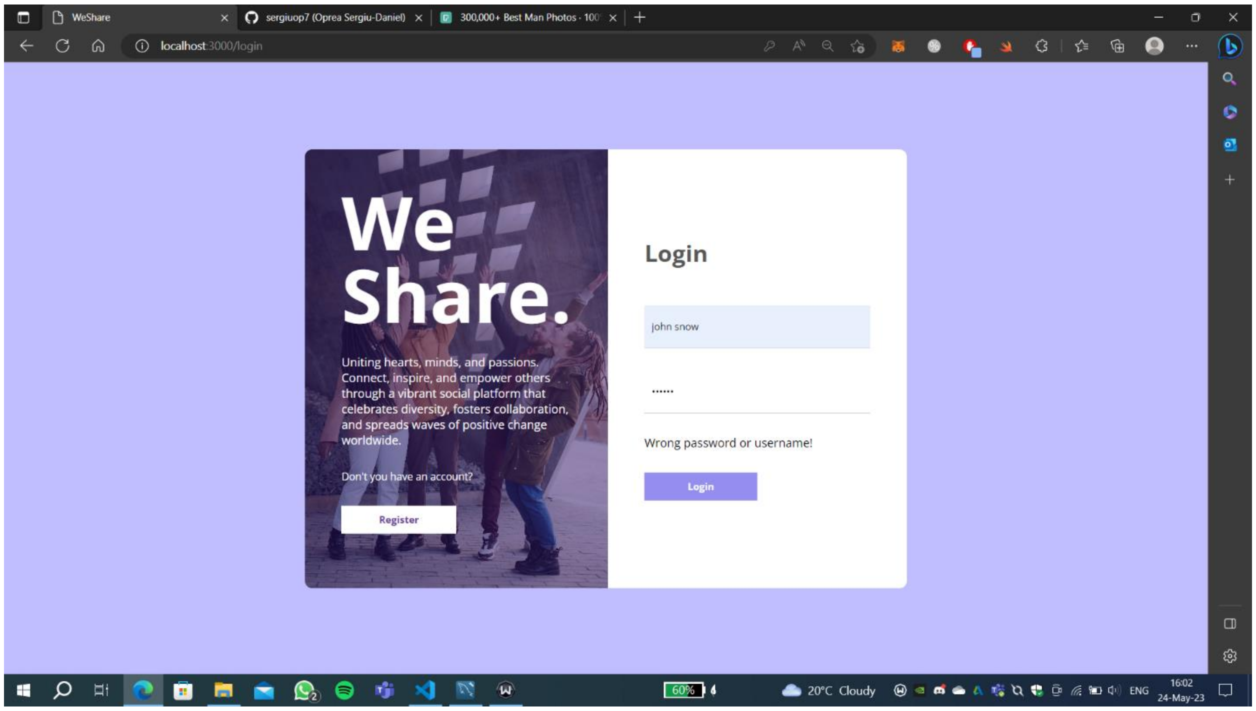

WeShare
localhost:3000/login
We Share.
Uniting hearts, minds, and passions. Connect, inspire, and empower others through a vibrant social platform that celebrates diversity, fosters collaboration, and spreads waves of positive change worldwide.
Don't you have an account?
Register
Login
Username
Password
Login

Figure 2 – Login page

The Register page requires additional information compared to the login page to create a new account.

Besides the username and password, you must also enter your email and name.

When we are trying to login with an account but the information entered is incorrect, we will get a “Wrong password or username!” error message (*Figure 3*).



Uniting hearts, minds, and passions. Connect, inspire, and empower others through a vibrant social platform that celebrates diversity, fosters collaboration, and spreads waves of positive change worldwide.

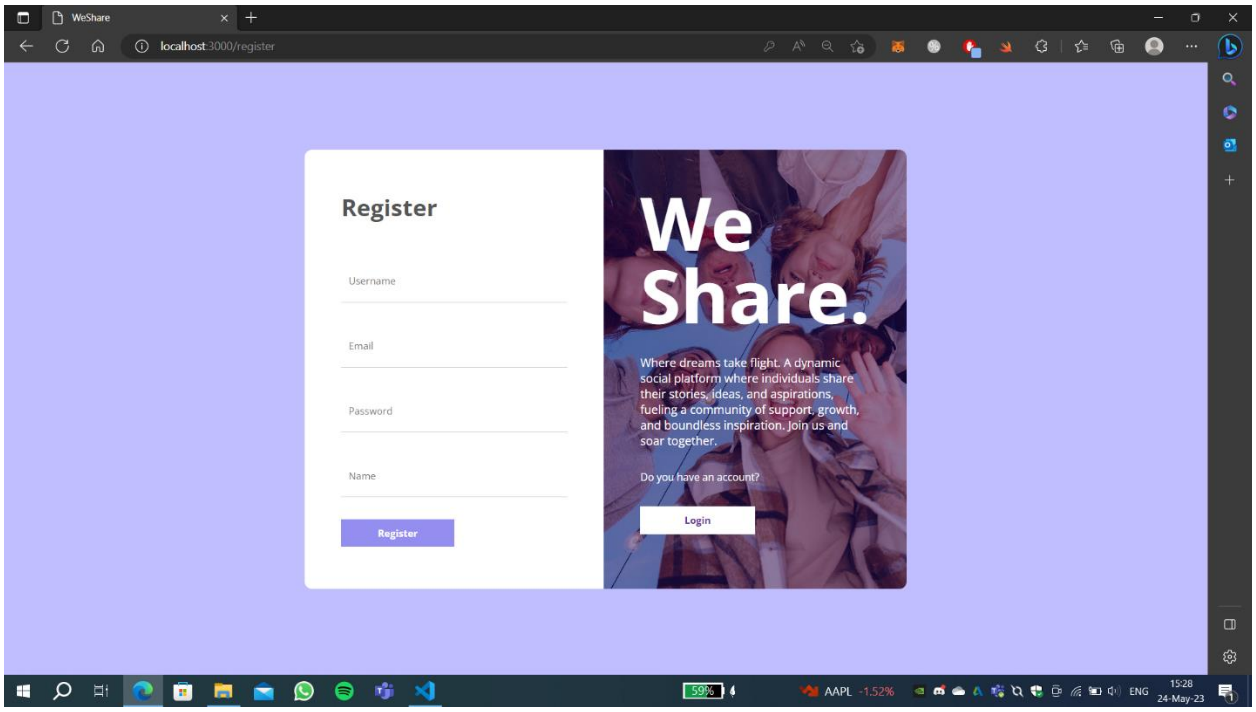
Don't you have an account?

john snow

••••••

Wrong password or username!

Figure 3 - Wrong password or username when creating trying to login with an account



Where dreams take flight. A dynamic social platform where individuals share their stories, ideas, and aspirations, fueling a community of support, growth, and boundless inspiration. Join us and soar together.

Do you have an account?

Figure 4 – Register page

If we want to register an already existing user, we will get an error above the "Register" button saying "User already exists!" saying that we need to enter other information for the user (*Figure 5*).

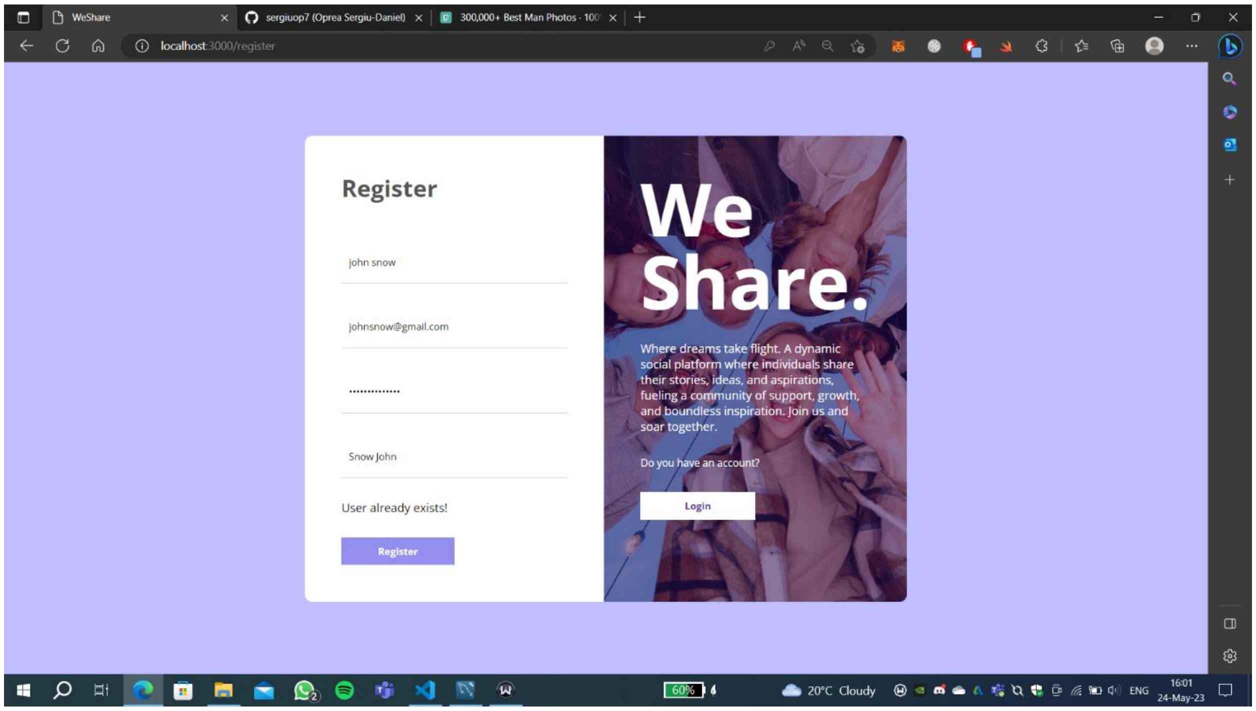

Register
john snow
johnsnow@gmail.com
Snow John
User already exists!
Register
We Share.
Where dreams take flight. A dynamic social platform where individuals share their stories, ideas, and aspirations, fueling a community of support, growth, and boundless inspiration. Join us and soar together.
Do you have an account?
Login

Figure 5 – User already exists error when trying to register a new account

After creating the account and logging in with it, you are redirected to the **Home** page.

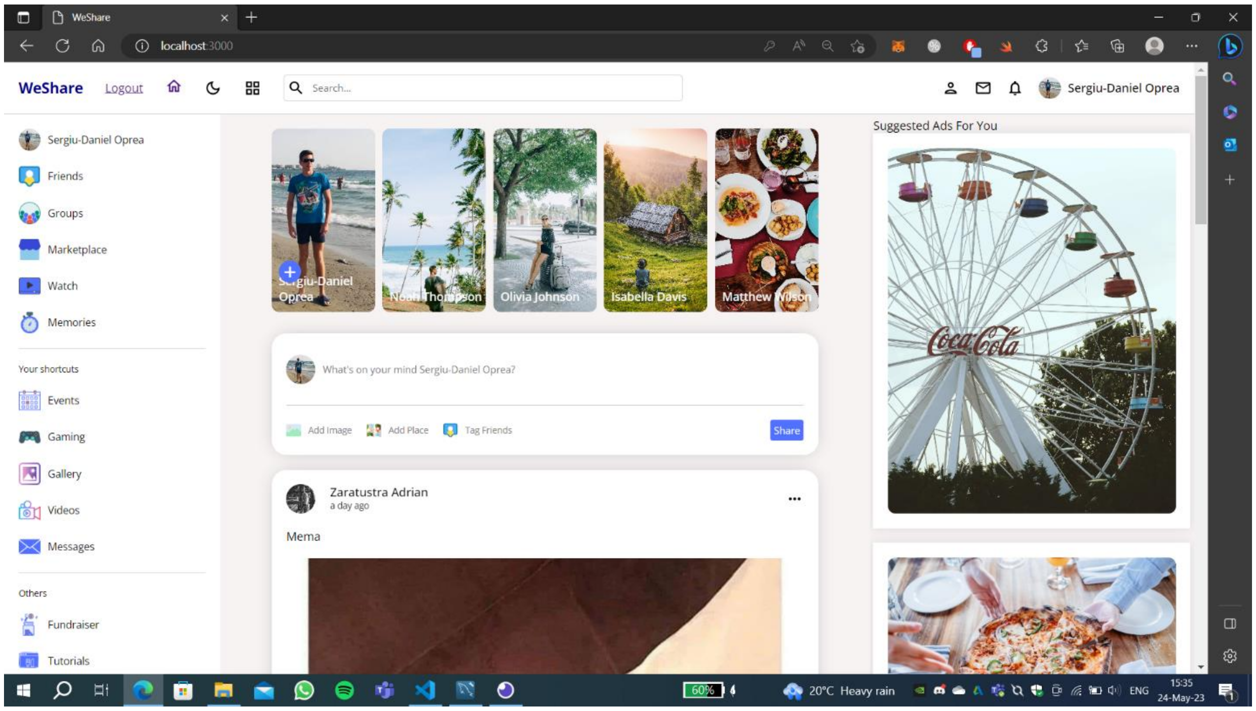
On the upper part of the page, you can see a bar that contains a few buttons. First, we have the logo of our application WeShare, followed by a "Logout" button that lets you log out of the application and redirects you to the login page. Then, we have the "Home" button, that redirects you to the feed of the application. Next, we have a moon-shaped button that enables you to switch between dark and white theme (*Figure 6*). The other button is decorative, and we will give it a functionality in a future release. Next, on the upper part of the page we have a search bar that allows the user to search other users. On the right side of the page, we have a button with the name of the user that allows the user to see their own page.

On the left side of the page, we have a sidebar that has multiple buttons, such as Friends, Groups, Marketplace etc. that will have functionalities in a future release.

On the right side of the page, we hardcoded some sponsored posts that showcase the capitalistic side of all contemporary applications.

Finally, on the center of the page, we have a few stories. Although the other ones (beside ours) are hardcoded on the view of the page, we can add our own story by pressing the "plus" button.

On the center of the page there also is a box where the user can add a new post and write a message or add images, add a location, or tag users. They will automatically appear in the feed after posting them with the time set as "A few seconds ago" (*Figure 6*).



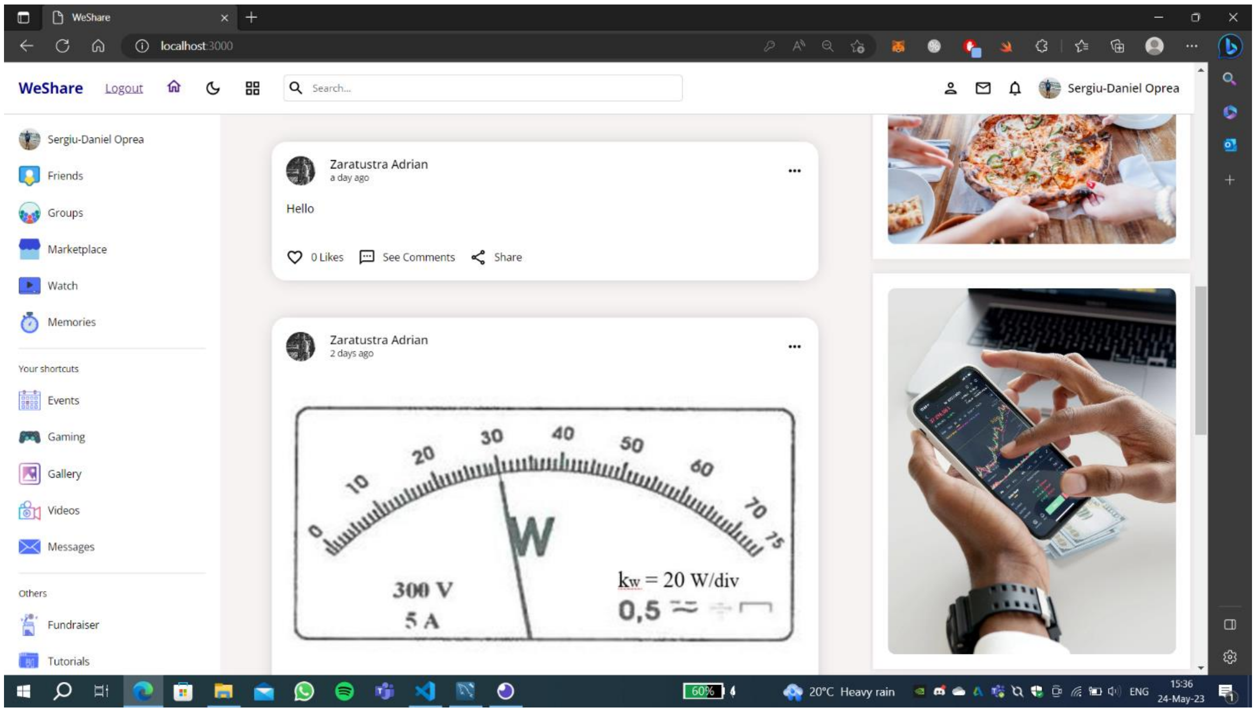
What's on your mind Sergiu-Daniel Oprea?

Zaratustra Adrian
a day ago

Mema

Figure 6 – Feed page

Below the box, we can see in the feed all the posts of the user or their friends in chronological order (*Figure 7*).

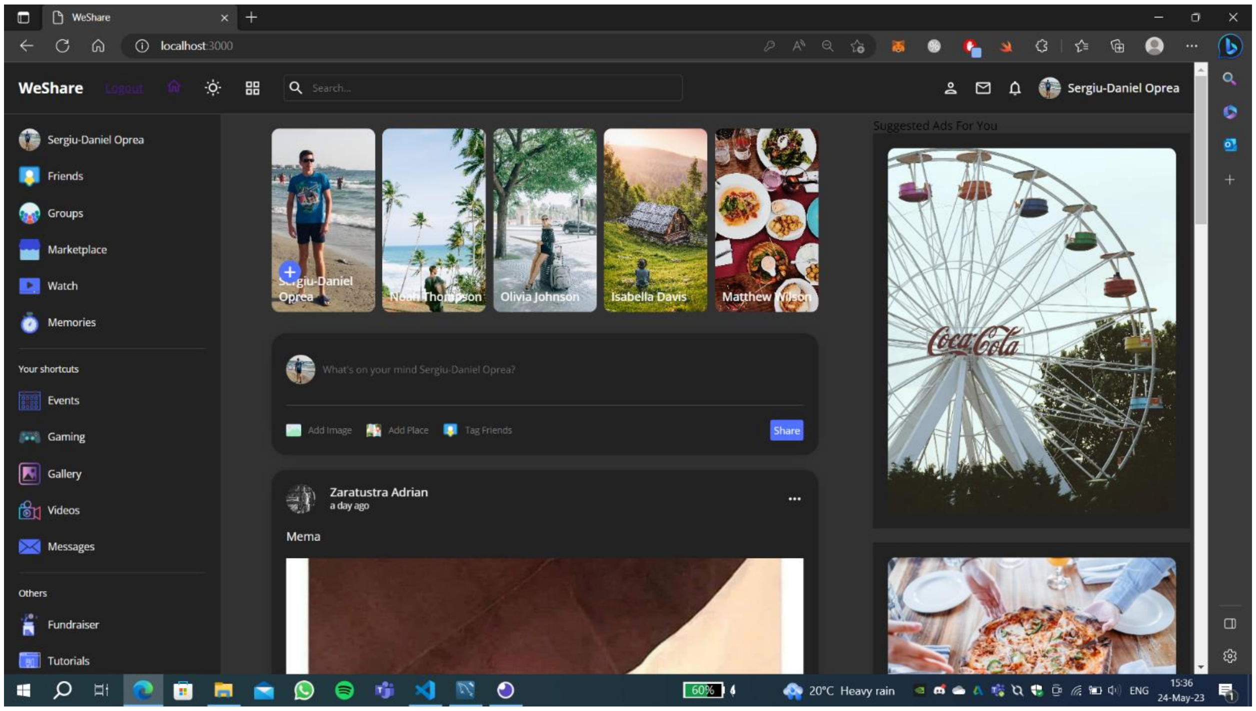


Zaratustra Adrian
a day ago

Hello

Zaratustra Adrian
2 days ago

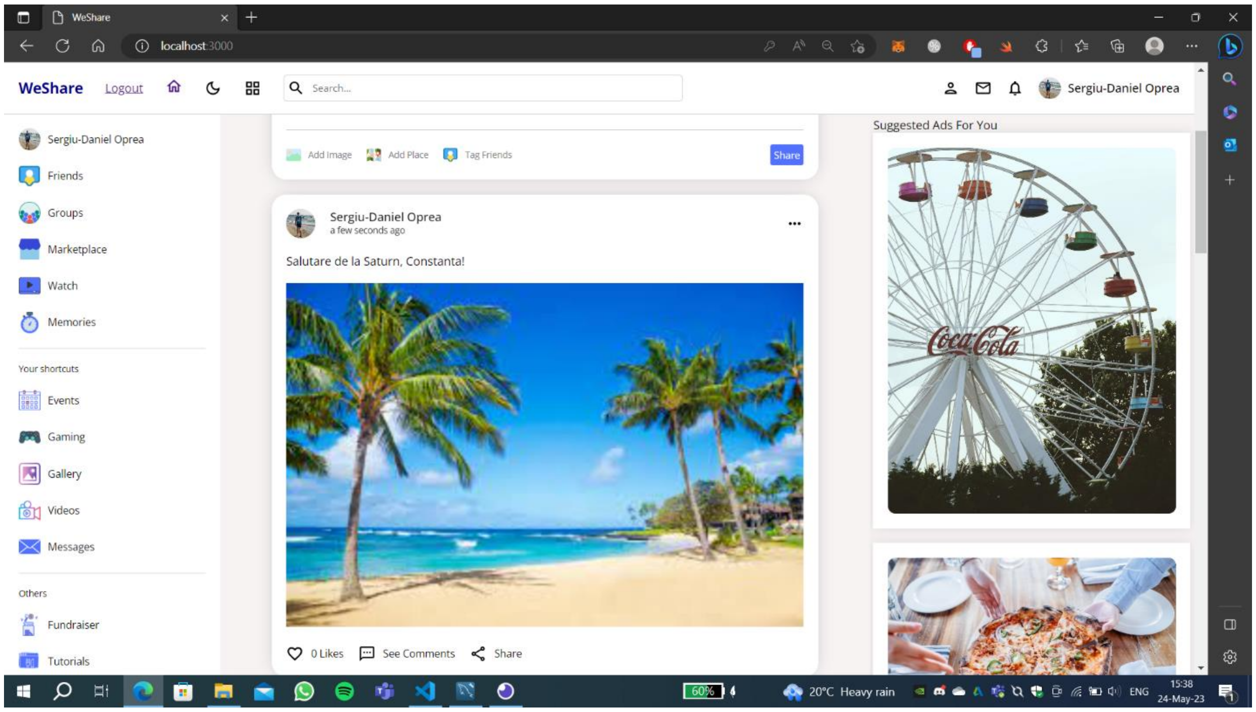
Figure 7 – Feed page, friends posts



Zaratustra Adrian
a day ago

Mema

Figure 8 – Dark theme page

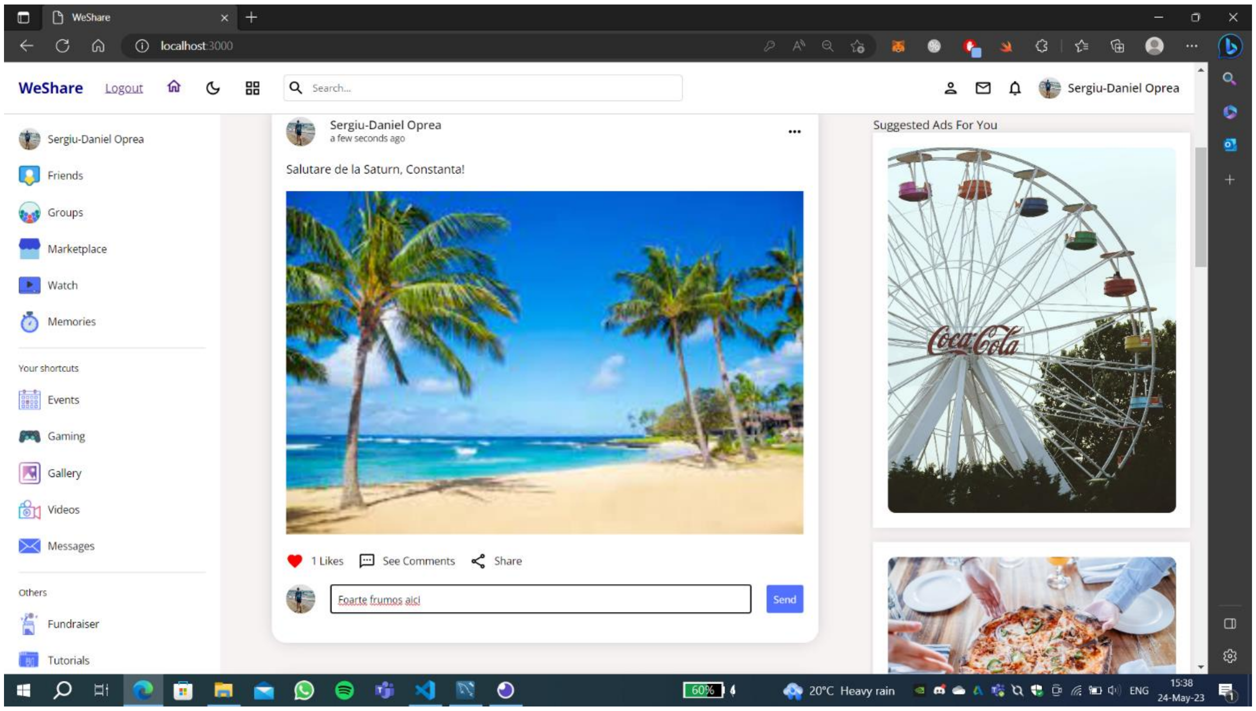


Sergiu-Daniel Oprea
a few seconds ago

Salutare de la Saturn, Constanta!

Figure 9 – Own post in feed, seconds after posting

When seeing any posts, the user can like them, comment on them, or share the post by pressing on the buttons below the post (*Figure 8*).

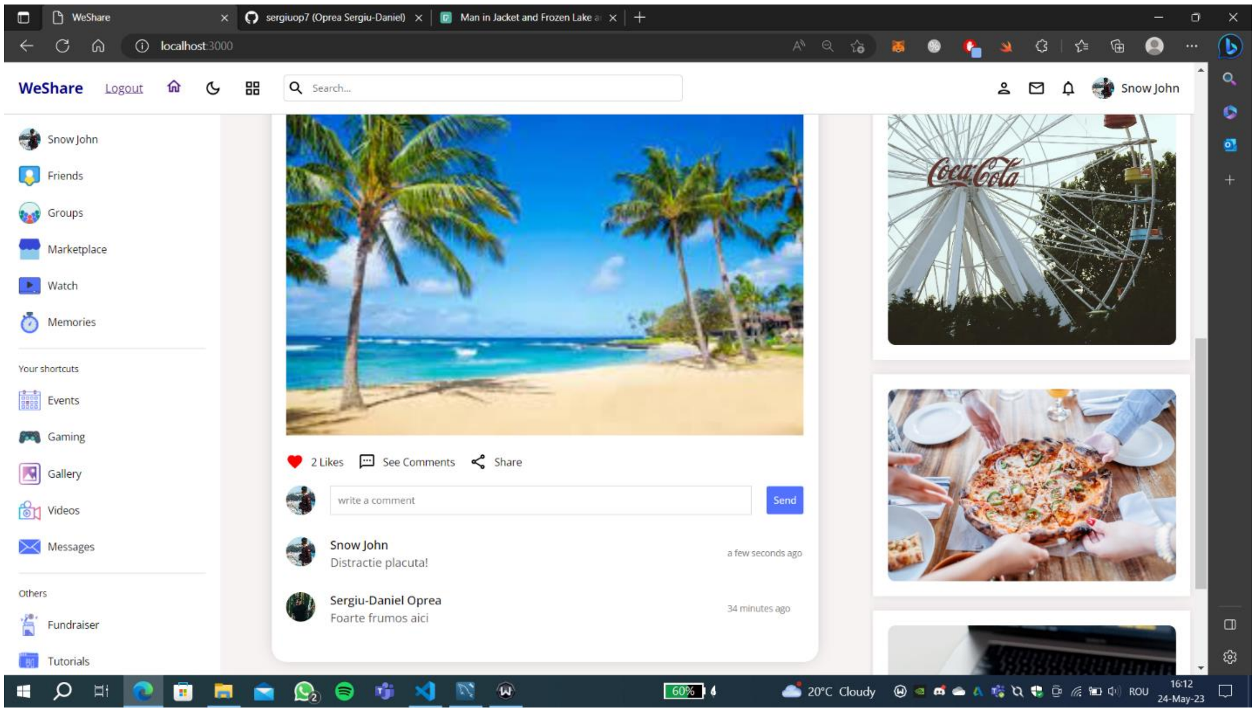


Sergiu-Daniel Oprea
a few seconds ago
Salutare de la Saturn, Constanta!

Foarte frumos aici

Figure 10 – Writing a comment on a post

After writing a comment and pressing on the "Send" button at the right of the textbox in which we input the message, the user and all their friends can see the comment on that specific post. Similarly to the posts, at the right side of the comment, the users can see how long ago it was posted (*Figure 9*).



Snow John
Distractie placuta!
a few seconds ago

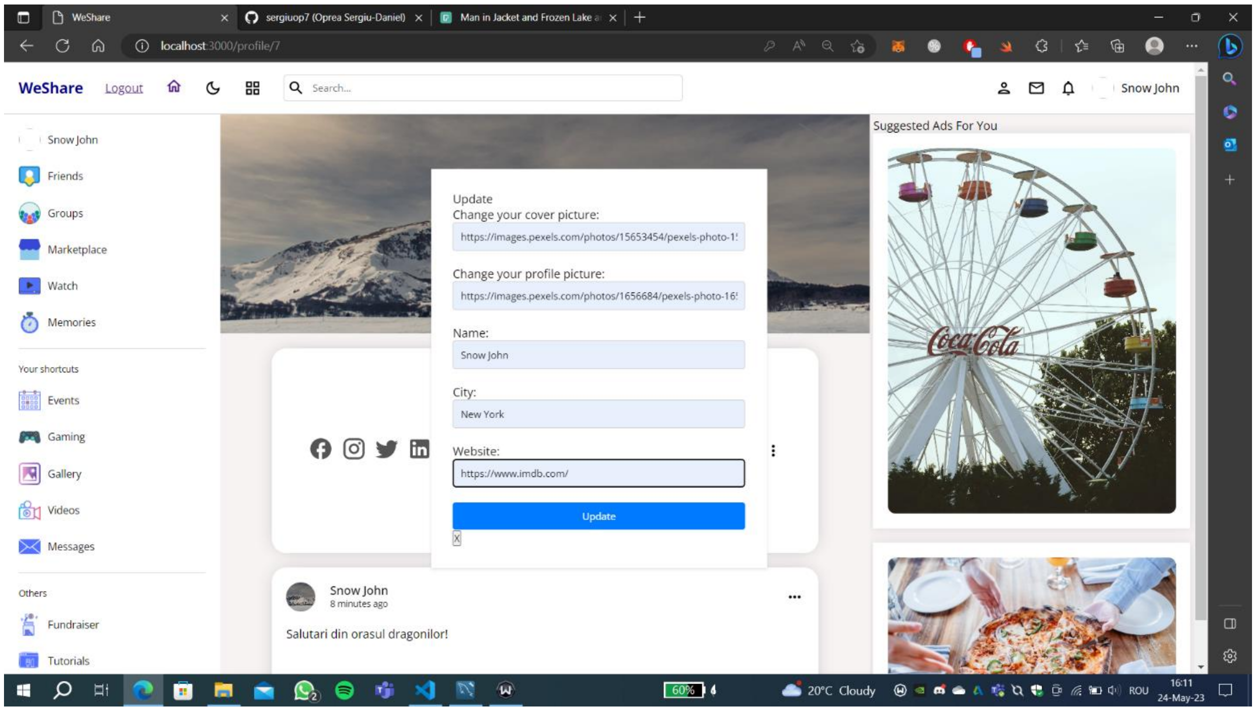
Sergiu-Daniel Oprea
Foarte frumos aici
34 minutes ago

Figure 11 – Seeing the comments

When you enter the **Profile** page (*Figure 10*), you will have the same side bar as the Home page, but the center of the page will have different functionalities. Here, the users can see the profile and cover pictures of their account, together with the location and a website that we linked there. We can also add links that redirect the users at the press of a button to other social media pages, such as Facebook, Instagram, Twitter, LinkedIn and Pinterest, as well as their email address.

In the box containing the user data, there is an "update" button that allows the user to update their information by inputting the new data into the form (*Figure 12*). The form requests new cover and profile pictures, name, city and website and the user can choose which information he wants to update. The user will then have a new updated information (*Figure 13*).

Below the box containing the user information, we can see all the posts of the user.



Update

Change your cover picture:
https://images.pexels.com/photos/15653454/pexels-photo-1!

Change your profile picture:
https://images.pexels.com/photos/1656684/pexels-photo-16!

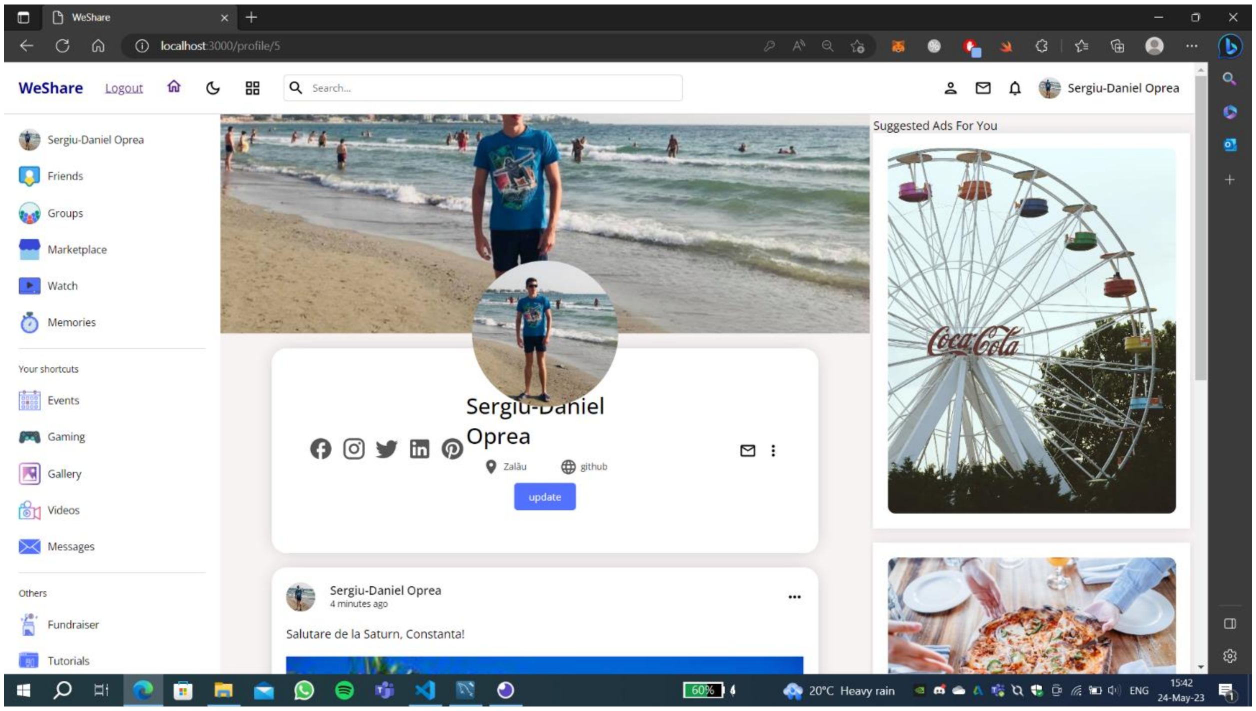
Name:
Snow John

City:
New York

Website:
https://www.imdb.com/

Snow John
8 minutes ago
Salutari din orasul dragonilor!

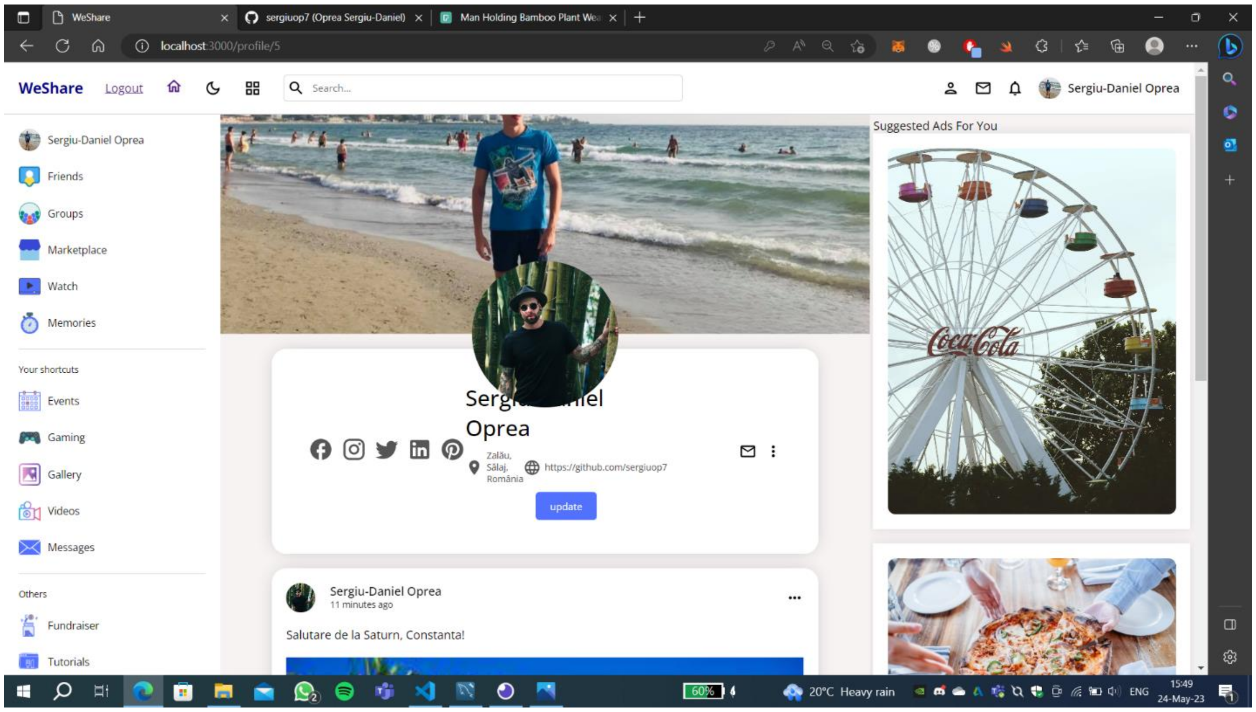
Figure 12 – Updating information form



Sergiu-Daniel Oprea

Salutare de la Saturn, Constanta!

Figure 13 – Profile page

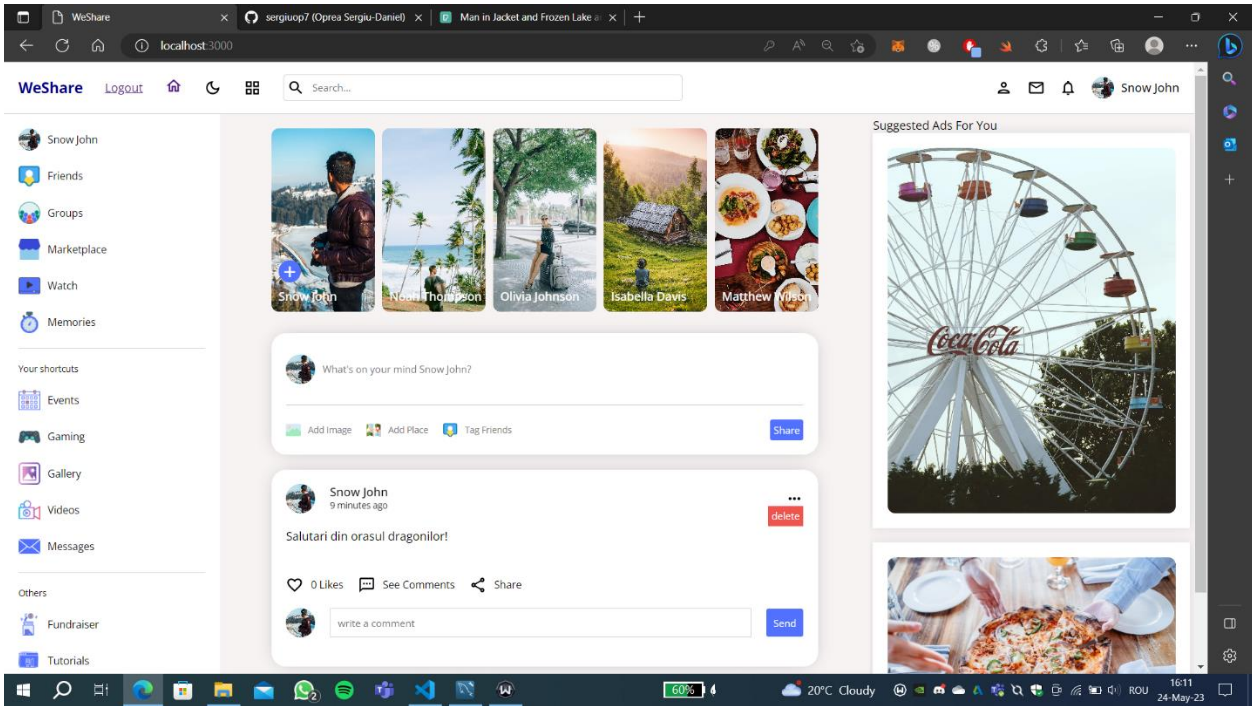


Sergiu-Daniel Oprea

Salutare de la Saturn, Constanta!

Figure 14 – Profile page after updating profile picture, location and website

At the Home page, the user can delete comment or post he posted on the application by pressing the *Delete* button.



Salutari din orasul dragonilor!

Figure 15 – Delete button displayed on the post

After pressing the button, the post will simply be deleted and not displayed anymore on the feed.

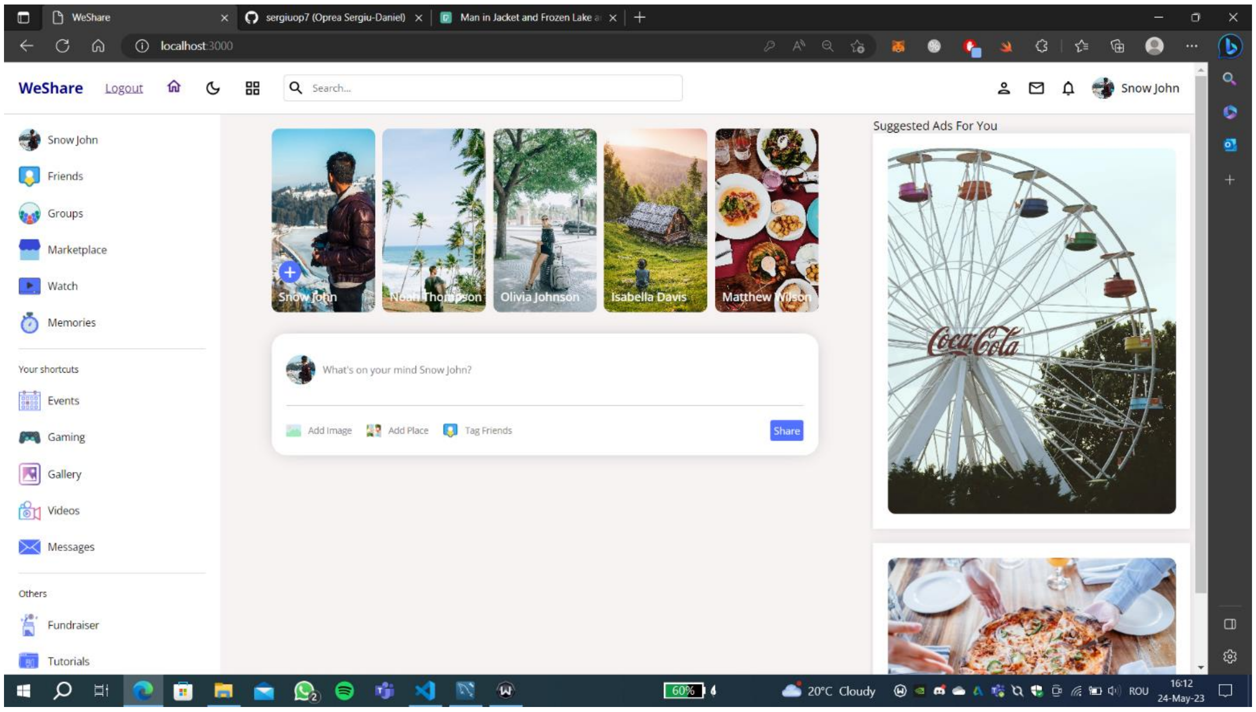


Figure 16 – Feed after deleting the post

Application Implementation

Database Implementation

The database was implemented in MySQL Workbench. Every text table column has an appropriate number of characters allowed, to adapt to all possible scenarios, specifically the password field that is encrypted to ensure the security of the users.

The SQL implementation of every particular table respects the design made in *Figure 1* and can be found in *Figure 17* to *Figure 22*.

Comments

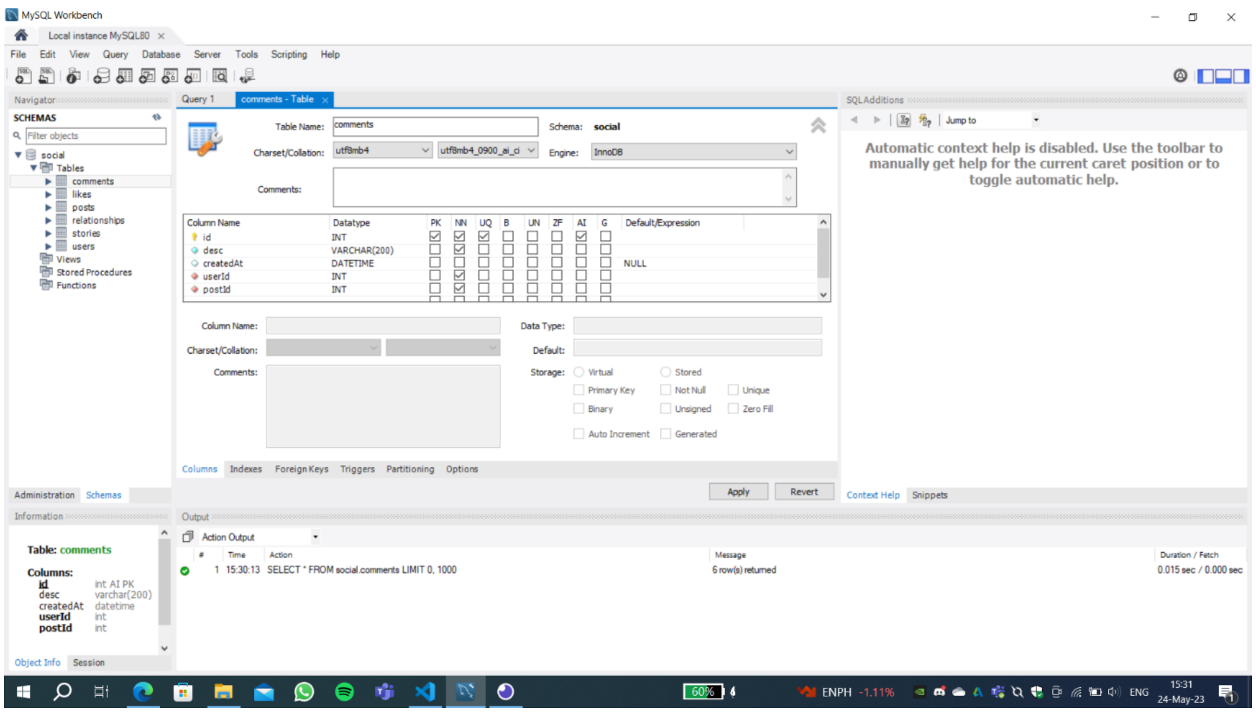


Table Name: comments
Schema: social
Charset/Collation: utf8mb4 utf8mb4_0900_ai_ci
Engine: InnoDB
Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
desc	VARCHAR(200)		✓							
createdAt	DATETIME									NULL
userId	INT		✓							
postId	INT		✓							

Column Name:
Data Type:
Charset/Collation:
Default:
Comments:
Storage: Virtual Stored
Primary Key Not Null Unique
Binary Unsigned Zero Fill
Auto Increment Generated

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Information

Table: comments

Columns:

id	int AI PK
desc	varchar(200)
createdAt	datetime
userId	int
postId	int

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	15:30:13	SELECT * FROM social.comments LIMIT 0, 1000	6 row(s) returned	0.015 sec / 0.000 sec

Figure 17 – Comments table

Likes

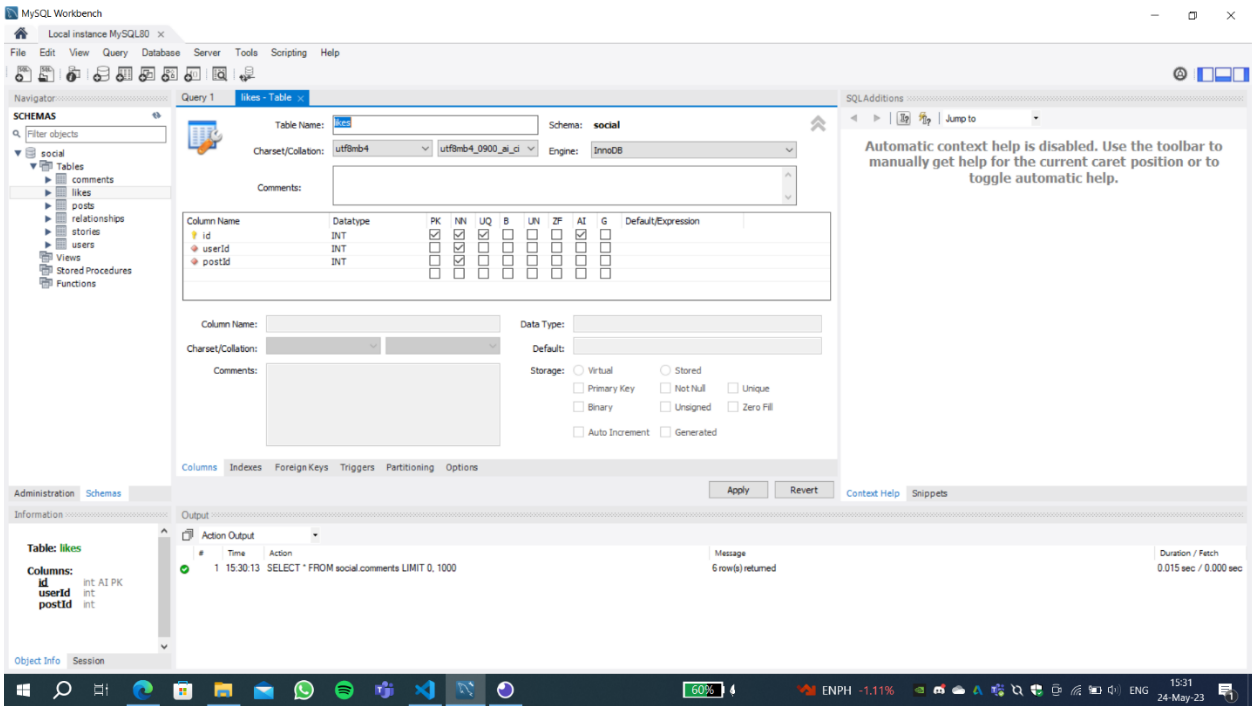


Table Name: likes
Schema: social
Charset/Collation: utf8mb4 | utf8mb4_0900_ai_ci
Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
userId	INT		✓							
postId	INT		✓							

Table: likes

Columns:
id int AI PK
userId int
postId int

```
#  Time      Action                                        Message               Duration / Fetch
1  15:30:13  SELECT * FROM social.comments LIMIT 0, 1000   6 row(s) returned     0.015 sec / 0.000 sec
```

Figure 18 – Likes table

Posts

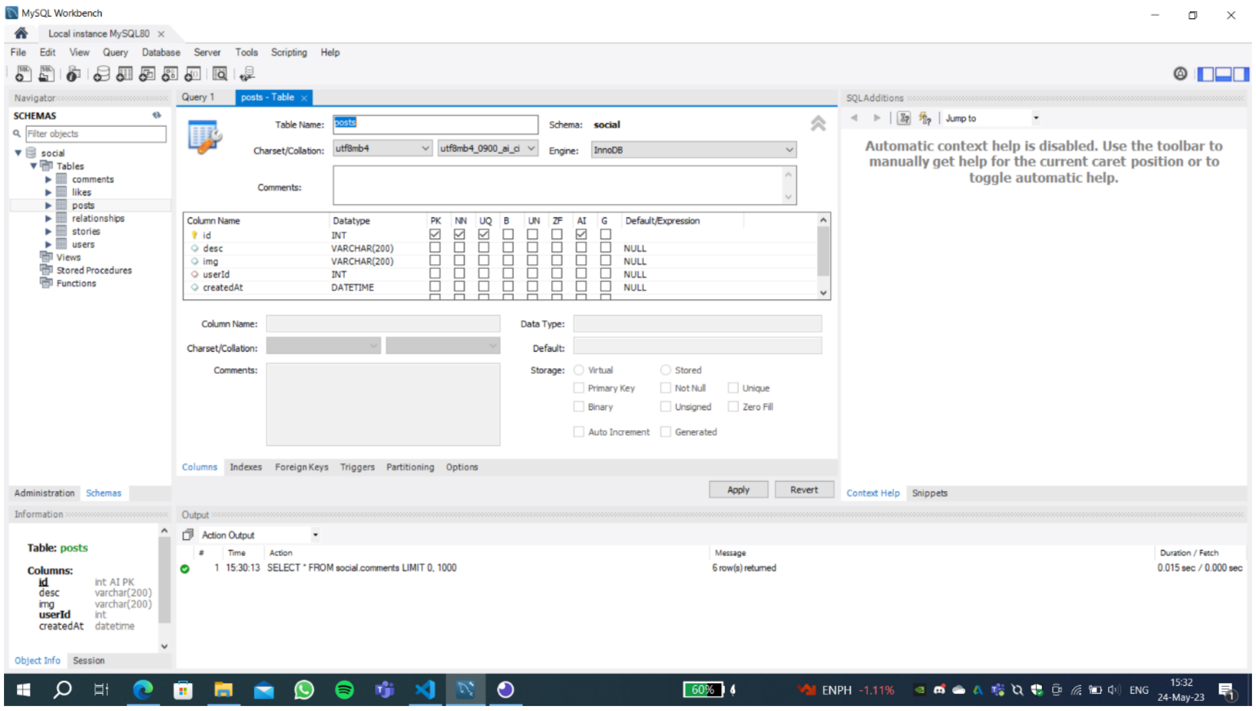


Table Name: posts
Schema: social
Charset/Collation: utf8mb4 | utf8mb4_0900_ai_ci
Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
desc	VARCHAR(200)									NULL
img	VARCHAR(200)									NULL
userId	INT									NULL
createdAt	DATETIME									NULL

Table: posts

Columns:
id int AI PK
desc varchar(200)
img varchar(200)
userId int
createdAt datetime

```
#  Time      Action                                        Message               Duration / Fetch
1  15:30:13  SELECT * FROM social.comments LIMIT 0, 1000   6 row(s) returned     0.015 sec / 0.000 sec
```

Figure 19 – Posts table

Relationships

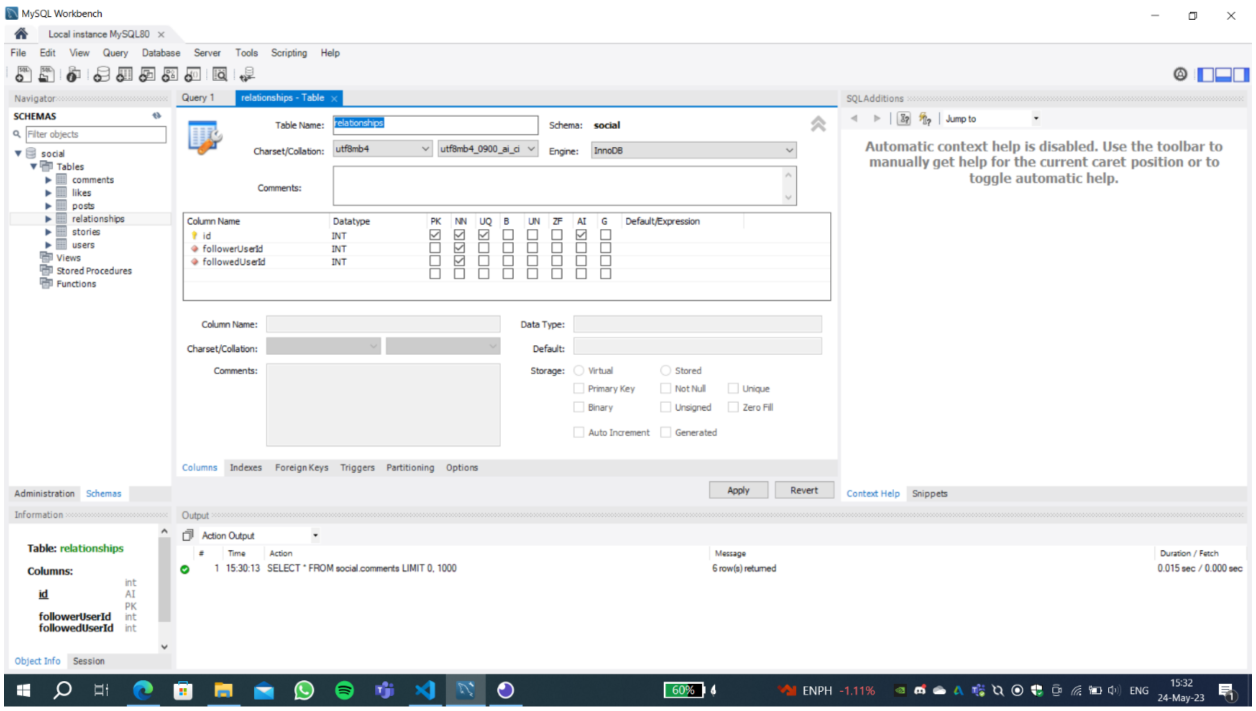


Table Name: relationships   Schema: social
Charset/Collation: utf8mb4   utf8mb4_0900_ai_ci   Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
followerUserId	INT		✓							
followedUserId	INT		✓							

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

```
#  Time      Action                                         Message               Duration / Fetch
1  15:30:13  SELECT * FROM social.comments LIMIT 0, 1000    6 row(s) returned     0.015 sec / 0.000 sec
```

Table: relationships
Columns:

```
id              int
                AI
                PK
followerUserId  int
followedUserId  int
```

Figure 20 – Relationships table

Stories

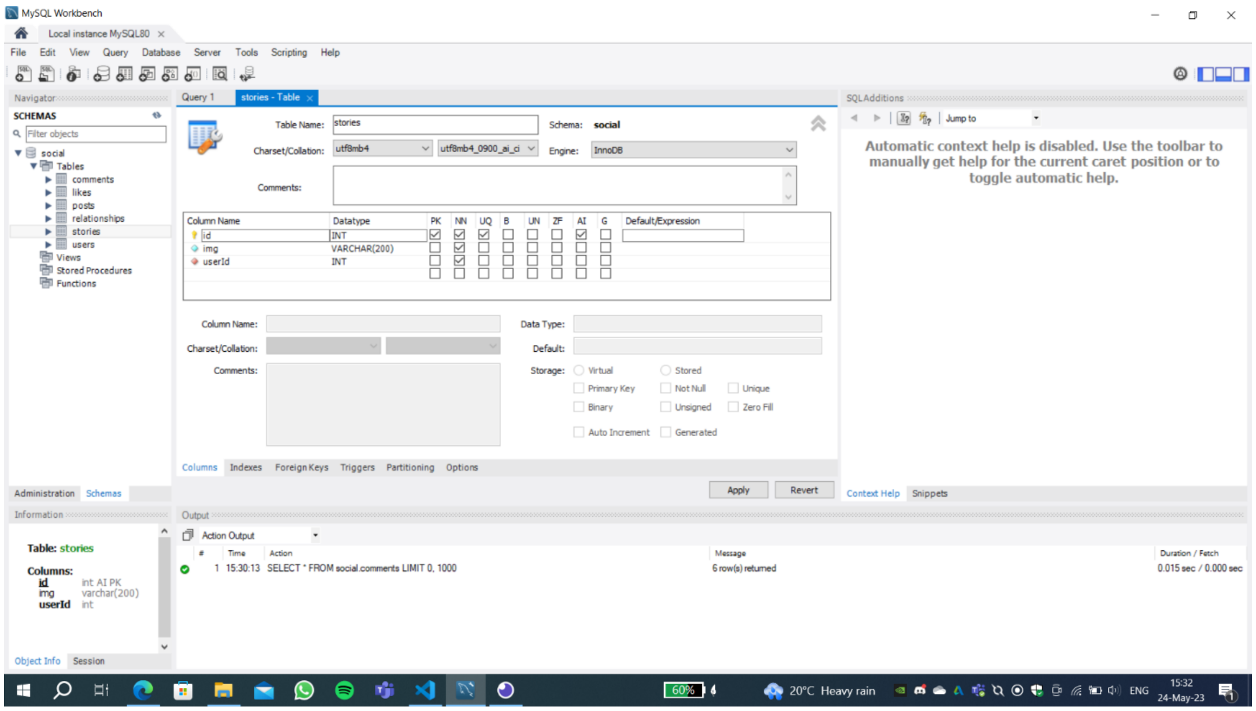


Table Name: stories   Schema: social
Charset/Collation: utf8mb4   utf8mb4_0900_ai_ci   Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
img	VARCHAR(200)		✓							
userId	INT		✓							

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

```
#  Time      Action                                         Message               Duration / Fetch
1  15:30:13  SELECT * FROM social.comments LIMIT 0, 1000    6 row(s) returned     0.015 sec / 0.000 sec
```

Table: stories
Columns:

```
id      int AI PK
img     varchar(200)
userId  int
```

Figure 10 – Stories table

Users

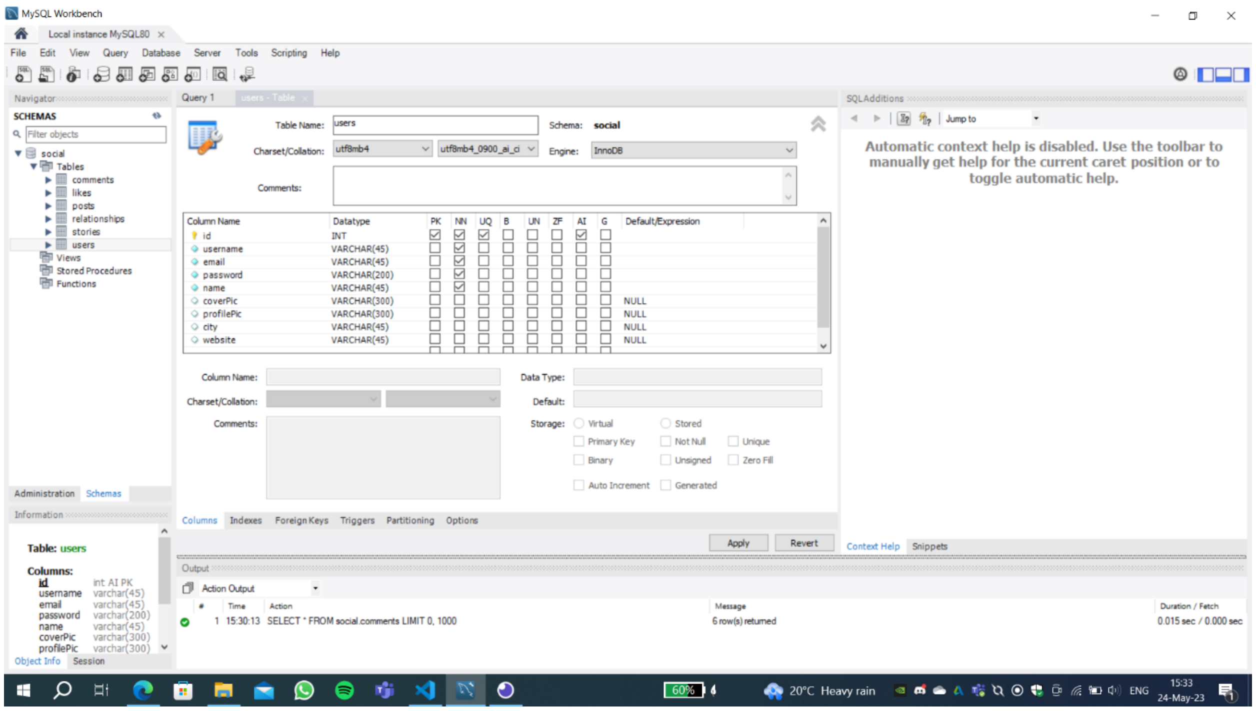


Table Name: users Schema: social
Charset/Collation: utf8mb4 utf8mb4_0900_ai_ci Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓	✓				✓		
username	VARCHAR(45)		✓							
email	VARCHAR(45)		✓							
password	VARCHAR(200)		✓							
name	VARCHAR(45)		✓							
coverPic	VARCHAR(300)									NULL
profilePic	VARCHAR(300)									NULL
city	VARCHAR(45)									NULL
website	VARCHAR(45)									NULL

Table: users

Columns:

```
id          int AI PK
username    varchar(45)
email       varchar(45)
password    varchar(200)
name        varchar(45)
coverPic    varchar(300)
profilePic  varchar(300)
```

#	Time	Action	Message	Duration / Fetch
1	15:30:13	SELECT * FROM social.comments LIMIT 0, 1000	6 row(s) returned	0.015 sec / 0.000 sec

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Figure 11 – Users table

Implementation Details

We divided the code into two parts: *controllers* and *route*. We included the functions used for our website in *controllers* and added routes for different functionalities of our application in the *route* folder.

We will first explain the *controllers* part of our application.

The first particularity of the code is the registration class that we created, where we hashed the password for increased security.

Auth.js

```
export const register = (req,res)=>{

    //CHECK USER IF EXISTS

    const q = "SELECT * FROM users WHERE username = ?"

    db.query(q,[req.body.username], (err,data)=>{
        if(err) return res.status(500).json(err);
        if(data.length) return res.status(409).json("User already exists!");
            //CREATE A NEW USER
      //Hash the password
      const salt = bcrypt.genSaltSync(10);
      const hashedPassword = bcrypt.hashSync(req.body.password, salt)

      const q = "INSERT INTO users (`username`,`email`,`password`,`name`)
VALUE (?)"

      const values =
[req.body.username,req.body.email,hashedPassword,req.body.name]

      db.query(q,[values], (err,data)=>{
        if(err) return res.status(500).json(err);
        return res.status(200).json("User has been created.");
      })
    })

}
```

Then, we implemented the CRUD operations in for the comments, so that any logged in user can post, delete, and update the comments at any post.

Comments.js

```
export const getComments = (req, res) => {
  const q = `SELECT c.*, u.id AS userId, name, profilePic FROM comments AS c
JOIN users AS u ON (u.id = c.userId)
    WHERE c.postId = ? ORDER BY c.createdAt DESC
    `;

  db.query(q, [req.query.postId], (err, data) => {
    if (err) return res.status(500).json(err);

    return res.status(200).json(data);

  });

};


export const addComment = (req, res) => {

  const token = req.cookies.accessToken;

  if (!token) return res.status(401).json("Not logged in!");


  jwt.verify(token, "secretkey", (err, userInfo) => {

    if (err) return res.status(403).json("Token is not valid!");


    const q = "INSERT INTO comments(`desc`, `createdAt`, `userId`, `postId`)
VALUES (?)";

    const values = [

      req.body.desc,

      moment(Date.now()).format("YYYY-MM-DD HH:mm:ss"),

      userInfo.id,

      req.body.postId

    ];


    db.query(q, [values], (err, data) => {

      if (err) return res.status(500).json(err);

      return res.status(200).json("Comment has been created.");

    });

  });

};


export const deleteComment = (req, res) => {

  const token = req.cookies.access_token;
  if (!token) return res.status(401).json("Not authenticated!");


  jwt.verify(token, "jwtkey", (err, userInfo) => {
    if (err) return res.status(403).json("Token is not valid!");


    const commentId = req.params.id;
    const q = "DELETE FROM comments WHERE `id` = ? AND `userId` = ?";


    db.query(q, [commentId, userInfo.id], (err, data) => {
      if (err) return res.status(500).json(err);
      if (data.affectedRows > 0) return res.json("Comment has been
deleted!");
      return res.status(403).json("You can delete only your comment!");
    });
  });
};
```

We also implemented CRUD operations for the Likes.

Likes.js

```
export const getLikes = (req,res)=>{
    const q = "SELECT userId FROM likes WHERE postId = ?";


    db.query(q, [req.query.postId], (err, data) => {
      if (err) return res.status(500).json(err);
      return res.status(200).json(data.map(like=>like.userId));
    });
}


export const addLike = (req, res) => {
  const token = req.cookies.accessToken;
  if (!token) return res.status(401).json("Not logged in!");

  jwt.verify(token, "secretkey", (err, userInfo) => {
    if (err) return res.status(403).json("Token is not valid!");

    const q = "INSERT INTO likes (`userId`,`postId`) VALUES (?)";
    const values = [
      userInfo.id,
      req.body.postId
    ];

    db.query(q, [values], (err, data) => {
      if (err) return res.status(500).json(err);
      return res.status(200).json("Post has been liked.");
    });
  });
};

export const deleteLike = (req, res) => {

  const token = req.cookies.accessToken;
  if (!token) return res.status(401).json("Not logged in!");

  jwt.verify(token, "secretkey", (err, userInfo) => {
    if (err) return res.status(403).json("Token is not valid!");

    const q = "DELETE FROM likes WHERE `userId` = ? AND `postId` = ?";

    db.query(q, [userInfo.id, req.query.postId], (err, data) => {
      if (err) return res.status(500).json(err);
      return res.status(200).json("Post has been disliked.");
    });
  });
};
```

Post.js

```
export const getPosts = (req,res)=>{

    const userId = req.query.userId;
    const token = req.cookies.accessToken;
    if(!token) return res.status(401).json("Not logged in!")

    jwt.verify(token, "secretkey", (err, userInfo)=>{
        if(err) return res.status(403).json("Token is not valid")

    console.log(userId);

    const q =
     userId !== "undefined"
        ? 'SELECT p.*, u.id AS userId, name, profilePic FROM posts AS p JOIN
users AS u ON (u.id = p.userId) WHERE p.userId = ? ORDER BY p.createdAt DESC'
        : `SELECT p.*, u.id AS userId, name, profilePic FROM posts AS p JOIN
users AS u ON (u.id = p.userId)
    LEFT JOIN relationships AS r ON (p.userId = r.followedUserId) WHERE
r.followerUserId= ? OR p.userId =?
    ORDER BY p.createdAt DESC`;

    const values =
     userId !== "undefined" ? [userId] : [userInfo.id, userInfo.id]
    db.query(q, values, (err,data) => {
        if (err) return res.status(500).json(err);
        return res.status(200).json(data);
    });
    })
}

export const addPost = (req,res)=>{
    const token = req.cookies.accessToken;
    if(!token) return res.status(401).json("Not logged in!")

    jwt.verify(token, "secretkey", (err, userInfo)=>{
        if(err) return res.status(403).json("Token is not valid")

    const q = "INSERT INTO posts (`desc`,`img`,`createdAt`,`userId`) VALUE
(?)";

    const values = [
        req.body.desc,
        req.body.img,
        moment(Date.now()).format("YYYY-MM-DD HH:mm:ss"),
        userInfo.id
    ]

    db.query(q, [values], (err,data) => {
        if (err) return res.status(500).json(err);
        return res.status(200).json("Post has been created");
    });
    })
};
```

```
export const deletePost = (req, res) => {

    const token = req.cookies.accessToken;

    if (!token) return res.status(401).json("Not logged in!");


    jwt.verify(token, "secretkey", (err, userInfo) => {

      if (err) return res.status(403).json("Token is not valid!");


      const q =

        "DELETE FROM posts WHERE `id`=? AND `userId` = ?";


      db.query(q, [req.params.id, userInfo.id], (err, data) => {

        if (err) return res.status(500).json(err);

        if(data.affectedRows>0) return res.status(200).json("Post has been
deleted.");

        return res.status(403).json("You can delete only your post")

      });

    });

  };
```

Relationship.js

```
export const getRelationships = (req,res)=>{

    const q = "SELECT followerUserId FROM relationships WHERE followedUserId
= ?";


    db.query(q, [req.query.followedUserId], (err, data) => {

      if (err) return res.status(500).json(err);

      return
res.status(200).json(data.map(relationship=>relationship.followerUserId));

    });

}
```

```
export const addRelationship = (req, res) => {
  const token = req.cookies.accessToken;
  if (!token) return res.status(401).json("Not logged in!");

  jwt.verify(token, "secretkey", (err, userInfo) => {
    if (err) return res.status(403).json("Token is not valid!");

    const q = "INSERT INTO relationships (`followerUserId`,`followedUserId`)
VALUES (?)";
    const values = [
      userInfo.id,
      req.body.userId
    ];

    db.query(q, [values], (err, data) => {
      if (err) return res.status(500).json(err);
      return res.status(200).json("Following");
    });
  });
};

export const deleteRelationship = (req, res) => {

  const token = req.cookies.accessToken;
  if (!token) return res.status(401).json("Not logged in!");

  jwt.verify(token, "secretkey", (err, userInfo) => {
    if (err) return res.status(403).json("Token is not valid!");
    const q = "DELETE FROM relationships WHERE `followerUserId` = ? AND
`followedUserId` = ?";

    db.query(q, [userInfo.id, req.query.userId], (err, data) => {
      if (err) return res.status(500).json(err);
      return res.status(200).json("Unfollow");
    });
  });
};
```

User.js

```
export const getUser = (req, res) => {
    const userId = req.params.userId;
    const q = "SELECT * FROM users WHERE id=?";

    db.query(q, [userId], (err, data) => {
      if (err) return res.status(500).json(err);
      const { password, ...info } = data[0];
      return res.json(info);
    });
  };

  export const updateUser = (req, res) => {
    const token = req.cookies.accessToken;
    if (!token) return res.status(401).json("Not authenticated!");

    jwt.verify(token, "secretkey", (err, userInfo) => {
      if (err) return res.status(403).json("Token is not valid!");

      const q =
        "UPDATE users SET
`name`=?,`city`=?,`website`=?,`profilePic`=?,`coverPic`=? WHERE id=? ";

      db.query(
        q,
        [
          req.body.name,
          req.body.city,
          req.body.website,
          req.body.coverPic,
          req.body.profilePic,
          userInfo.id,
        ],
        (err, data) => {
          if (err) res.status(500).json(err);
          if (data.affectedRows > 0) return res.json("Updated!");
          return res.status(403).json("You can update only your post!");
        }
      );
    });
  };
```

Then, we have the *route* part of our application.

Auth.js

```
import Express from "express";
import { login,register,logout } from "../controllers/auth.js";

const router = Express.Router()

router.post("/login", login)
router.post("/register", register)
router.post("/logout", logout)

export default router
```

Comments.js

```
import express from "express";
import {
  getComments,
  addComment,
  deleteComment,
} from "../controllers/comment.js";

const router = express.Router();

router.get("/", getComments);
router.post("/", addComment);
router.delete("/:id", deleteComment);

export default router;
```

Likes.js

```
import express from "express";
import { getLikes, addLike, deleteLike } from "../controllers/like.js";

const router = express.Router()

router.get("/", getLikes)
router.post("/", addLike)
router.delete("/", deleteLike)
```

```
export default router
```

Posts.js

```
import Express from "express";
import { getPosts, addPost, deletePost } from "../controllers/post.js";

const router = Express.Router()

router.get("/", getPosts)
router.post("/", addPost)
router.delete("/:id", deletePost)

export default router
```

Relationships.js

```
import express from "express";
import { getRelationships, addRelationship, deleteRelationship } from
"../controllers/relationship.js";

const router = express.Router()

router.get("/", getRelationships)
router.post("/", addRelationship)
router.delete("/", deleteRelationship)



export default router
```

Users.js

```
import Express from "express";
import { getUser, updateUser } from "../controllers/user.js";

const router = Express.Router()
```

```
router.get("/find/:userId", getUser)
router.put("/", updateUser)

export default router
```

We have connected everything in the **index.html** file:

Index.html

```
import express from "express";
const app = express();
import authRoutes from "./routes/auth.js";
import userRoutes from "./routes/users.js";
import postRoutes from "./routes/posts.js";
import commentRoutes from "./routes/comments.js";
import likeRoutes from "./routes/likes.js";
import relationshipRoutes from "./routes/relationships.js";
import cors from "cors";
import multer from "multer";
import cookieParser from "cookie-parser";

//middlewares
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Credentials", true);
  next();
});
app.use(express.json());
app.use(
  cors({
    origin: "http://localhost:3000",
  })
);
app.use(cookieParser());

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "../client/public/upload");
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + file.originalname);
  },
});

const upload = multer({ storage: storage });

app.post("/api/upload", upload.single("file"), (req, res) => {
  const file = req.file;
  res.status(200).json(file.filename);
});

app.use("/api/auth", authRoutes);
app.use("/api/users", userRoutes);
app.use("/api/posts", postRoutes);
app.use("/api/comments", commentRoutes);
app.use("/api/likes", likeRoutes);
app.use("/api/relationships", relationshipRoutes);

app.listen(8800, () => {
  console.log("API working!");
});
```

Conclusion

In conclusion, the WeShare application was developed following each step in the design process. From the idea of an application that can bring together people and help them keep in touch, to the testing of it, the entire team pitched in ideas and helped with solutions to parts that posed difficulties.

Following a certain pattern, we learnt how a real-life application is developed from the stage of a concept to the actual application used in everyday life. Keeping an organized structure, it was easier to make modifications to parts that did not follow certain criteria, posing as a great exercise for when we will be part of a team that implements real applications that will be put at the disposition of users.