

WEB APPLICATION DEVELOPMENT

FEDDIT FORUM

PRESNTATION OF GROUP DQ



TABLE OF CONTENT

I. INTRODUCTION

II. PROJECT OBJECTIVES

III. PROJECT ARCHITECTURE

IV. CORE FEATURES

V. KEY FUNCTIONALITIES

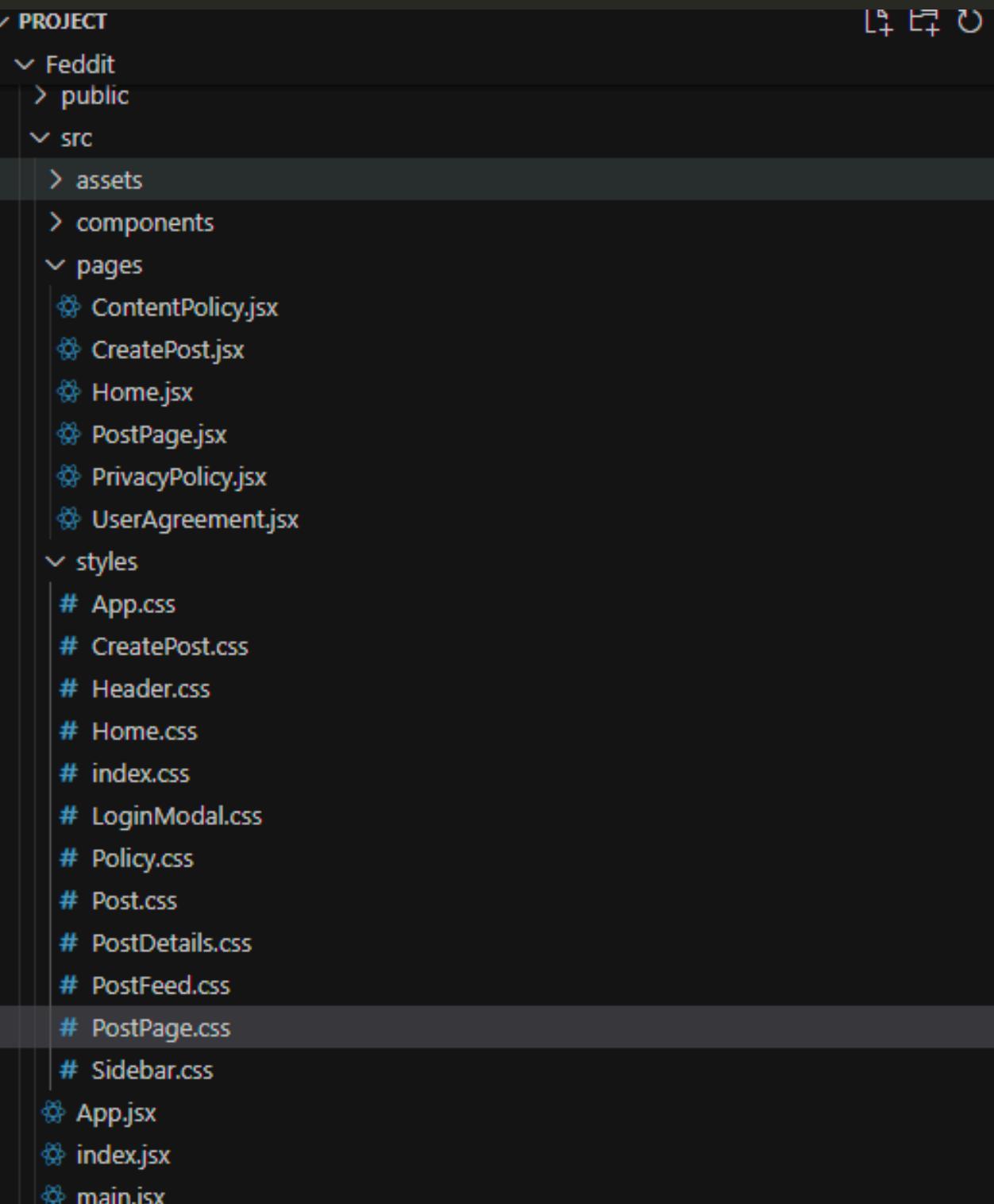
VI. CHALLENGES & TESTING

VII. FUTURE ENHANCEMENTS

VIII. CONCLUSION

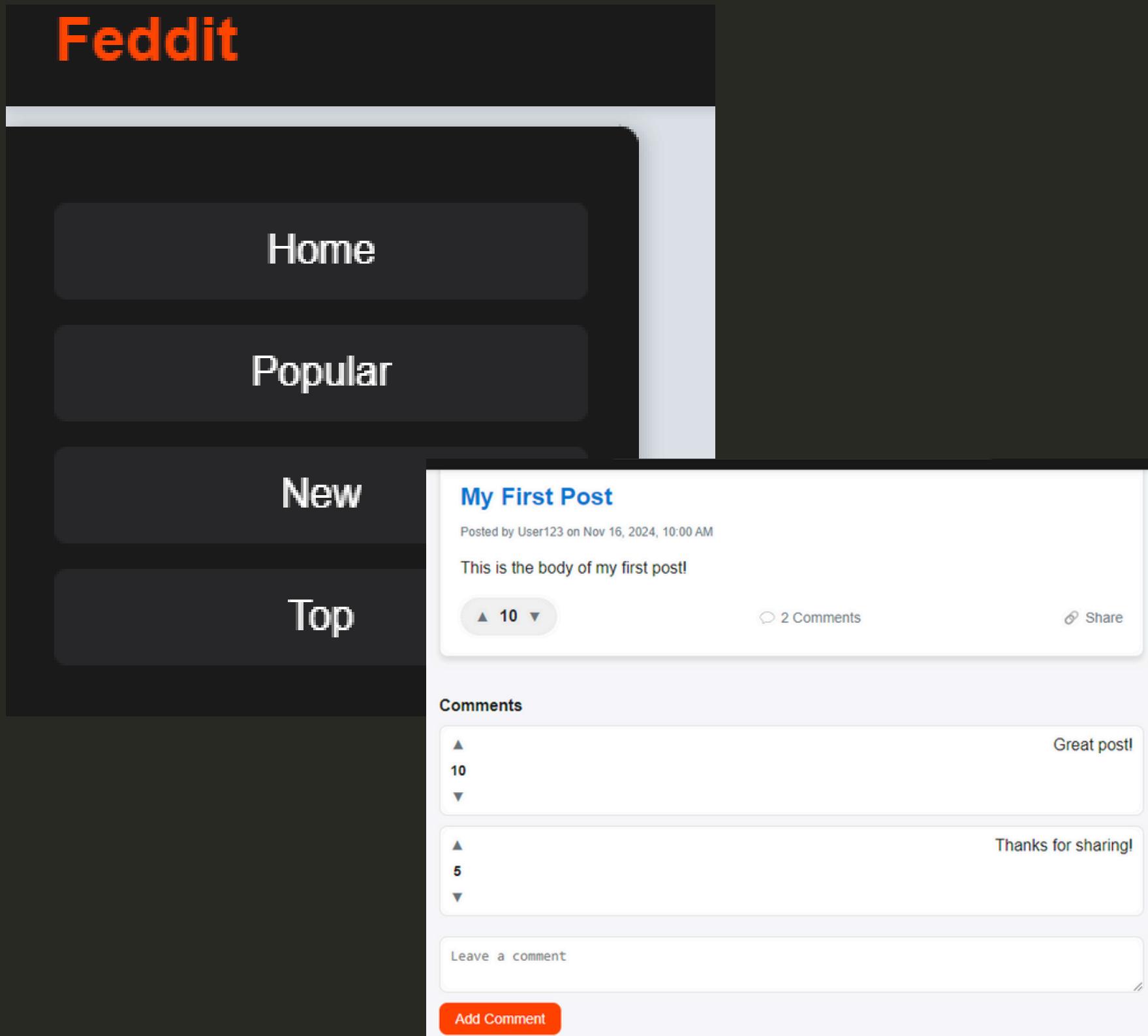
PROJECT ARCHITECTURE

- Frontend Architecture:
 - Header: Navigation and search.
 - Sidebar: Category navigation.
 - PostFeed: Dynamic post feed.
 - PostPage: Detailed post view with comments.



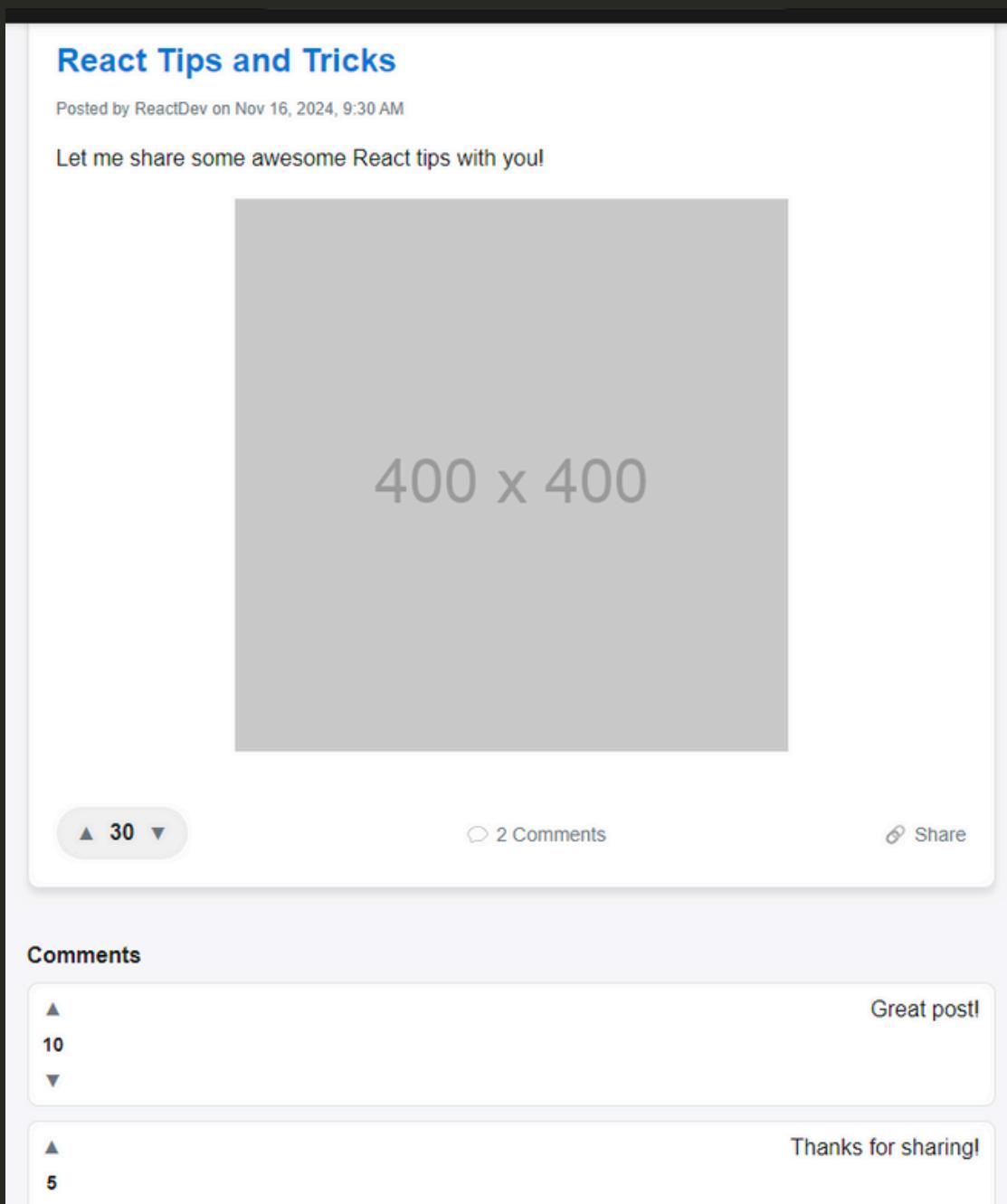
CORE FEATURES

- Home Page: Sorted post feeds (Popular, New, Top).
- Post Page:
 - Title, votes, comments, and share button.
 - Comment section with dynamic upvotes.
- Sidebar: Quick navigation across categories.



KEY FUNCTIONALITIES

- Upvote/Downvote Logic: Dynamic updates with React state.
- Commenting System: Add comments dynamically with nested structure.
- Share Button: URL display and clipboard copy functionality.



BACKEND: SERVER.JS

- Establish connection to MongoDB
- Create local server to run the web application.

```
feddit-backend > src > server.js > ...
...
1  const express = require('express');
2  const mongoose = require('mongoose');
3  const cors = require('cors');
4  const dotenv = require('dotenv');
5  const routes = require('./routes');

...
6
7
8  dotenv.config();
9
10 const app = express();
11 const PORT = process.env.PORT || 5001;
12
13 // Middleware
14 app.use(cors());
15 app.use(express.json());
16 routes(app);
17
18 // Connect to MongoDB
19 mongoose.connect('mongodb+srv://vuvandovuvan:Blackiscool123@cluster0.rcltc.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0')
20   .then(() => console.log('MongoDB connected'))
21   .catch(err => console.log(err));
22
23 // Start the server
24 app.listen(PORT, () => {
25   console.log(`Server is running on http://localhost:${PORT}`);
26 });
27
28 console.log('MongoDB URI:', process.env.MONGODB_URI);
29
30 //Test server
31 app.get('/',(req, res) => {
32   return res.send('Server is running');
33 })
```

BACKEND: COMMENTS MODEL

- Defines fields of a Comment Object.
- One user can make multiple comments in one post.
- Author and post fields refer to User and Post Models respectively.

```
feddit-backend > src > models > Comment.js > ...
1 // models/Comment.js
2 const mongoose = require('mongoose');
3
4 const commentSchema = new mongoose.Schema({
5   body: { type: String, required: true }, // Body of the comment
6   author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }, // Reference to the User model
7   post: { type: mongoose.Schema.Types.ObjectId, ref: 'Post', required: true }, // Reference to the Post model
8   timestamp: { type: Date, default: Date.now }, // Timestamp for when the comment was created
9   upvotes: { type: Number, default: 0 }, // Number of upvotes for the comment
10 });
11
12 module.exports = mongoose.model('Comment', commentSchema);
```

BACKEND: POST MODEL

- Defines fields of a Post Object.
- One post belongs to one user and can have multiple comments.
- Author and comments field refer to User and Comment Models.

```
feddit-backend > src > models > Post.js > ...
1 // models/Post.js
2 const mongoose = require('mongoose');
3
4 const postSchema = new mongoose.Schema({
5   title: { type: String, required: true },
6   body: { type: String, required: false },
7   media: { type: String, required: false },
8   author: { type: mongoose.Schema.Types.ObjectId, ref: 'User ', required: true }, // Reference to the User model
9   timestamp: { type: Date, default: Date.now },
10  views: { type: Number, default: 0 },
11  upvotes: { type: Number, default: 0 },
12  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment' }] // Reference to the Comment model
13 });
14
15 module.exports = mongoose.model('Post', postSchema);
```

BACKEND: USER MODEL

- Defines fields of a User Object.
- One user can have multiple posts.
- Post field refers to Post model.

```
feddit-backend > src > models > User.js > ...
1 // models/User.js
2 const mongoose = require('mongoose');
3
4 const UserSchema = new mongoose.Schema({
5   username: { type: String, required: true, unique: true },
6   password: { type: String, required: true },
7   email: { type: String, required: true, unique: true },
8   firstName: { type: String, required: true },
9   lastName: { type: String, required: true },
10  profilePicture: { type: String, default: '' },
11  dateOfBirth: { type: Date, required: false },
12  createdAt: { type: Date, default: Date.now },
13  updatedAt: { type: Date, default: Date.now },
14  posts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Post' }] // Reference to the Post model
15 });
16
17 module.exports = mongoose.model('User', UserSchema);
```

BACKEND: AUTHENTICATION

- POST /register: allows new user to register.
- Extracts username and password from request body.
- hashes the password using bcrypt.
- Saves to database and returns status code.
- POST /login: Checks username and compare password.
- Compares provided password with stored hased password.
- Returns JWT to client if successsful.

```
feddit-backend > src > routes > auth.js > ...
● 1  const express = require('express');
2  const bcrypt = require('bcryptjs');
3  const jwt = require('jsonwebtoken');
4  const User = require('../models/User');
5  const nodemailer = require('nodemailer'); // For sending emails
6  const crypto = require('crypto'); // For generating tokens
7
8  const router = express.Router();
9
10 // Register
11 router.post('/register', async (req, res) => {
12   const { username, password } = req.body;
13   const hashedPassword = await bcrypt.hash(password, 10);
14   const newUser = new User({ username, password: hashedPassword });
15   await newUser.save();
16   res.status(201).json({ message: 'User registered' });
17 });
18
19 // Login
20 router.post('/login', async (req, res) => {
21   const { username, password } = req.body;
22   const user = await User.findOne({ username });
23   if (!user) return res.status(400).json({ message: 'User not found' });
24
25   const isMatch = await bcrypt.compare(password, user.password);
26   if (!isMatch) return res.status(400).json({ message: 'Invalid credentials' });
27
28   const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });
29   res.json({ token, username });
30 });
31
32 // Forgot Password
33 router.post('/forgot-password', async (req, res) => {
34   const { email } = req.body;
35   const user = await User.findOne({ email });
36   if (!user) return res.status(400).json({ message: 'User not found' });
37
38   // Generate a reset token
39   const resetToken = crypto.randomBytes(32).toString('hex');
40   user.resetToken = resetToken;
41   user.resetTokenExpiration = Date.now() + 3600000; // 1 hour
42   await user.save();
```

BACKEND: AUTHENTICATION

- POST /forgot-password: allows user to request password reset.
- Checks user's email and generate reset token.
- Sends email with link to reset using nodemailer.
- POST /reset-password: Resets password using token.
- Checks validity and expiration of token.
- Hashes new password and update in the database.
- Removes reset token to prevent reuse.

```
feddit-backend > src > routes > auth.js > ...
33  router.post('/forgot-password', async (req, res) => {
34    // Send email with reset Link (using nodemailer)
35    const transporter = nodemailer.createTransport({
36      service: 'Gmail',
37      auth: {
38        user: process.env.EMAIL_USER,
39        pass: process.env.EMAIL_PASS,
40      },
41    });
42
43    const mailOptions = {
44      to: email,
45      subject: 'Password Reset',
46      text: `You requested a password reset. Click the link to reset your password:  
http://localhost:5000/reset-password/${resetToken}`,
47    };
48
49    transporter.sendMail(mailOptions, (error, info) => {
50      if (error) {
51        return res.status(500).json({ message: 'Error sending email' });
52      }
53      res.json({ message: 'Password reset link sent to your email' });
54    });
55
56    // Reset Password
57    router.post('/reset-password/:token', async (req, res) => {
58      const { password } = req.body;
59      const user = await User.findOne({
60        resetToken: req.params.token,
61        resetTokenExpiration: { $gt: Date.now() },
62      });
63
64      if (!user) return res.status(400).json({ message: 'Invalid or expired token' });
65
66      user.password = await bcrypt.hash(password, 10);
67      user.resetToken = undefined; // Clear the reset token
68      user.resetTokenExpiration = undefined; // Clear the expiration
69      await user.save();
70
71      res.json({ message: 'Password has been reset' });
72    });
73
74    module.exports = router;
75  
```

BACKEND: POST ROUTE

- POST /: allows clients to create a new post.
- Creates a new instance using Post Model.
- Saves post to database using ‘await newPost.save()’
- If successful, returns status code 201 with created post in JSON format.
- GET /: Retrieves all posts from the database.
- Uses ‘Post.find()’ to fetch all documents in the posts collection
- Sends back to the client in JSON format.

```
feddit-backend > src > routes > posts.js > router.get('/') callback
1  const express = require('express');
2  const Post = require('../models/Post');
3
4  const router = express.Router();
5
6  // Create a post
7  router.post('/', async (req, res) => {
8      const newPost = new Post(req.body);
9      await newPost.save();
10     res.status(201).json(newPost);
11 });
12
13 // Get all posts
14 router.get('/', async (req, res) => {
15     const posts = await Post.find();
16     res.json(posts);
17 });
18
19 // Update a post
20 router.put('/: id', async (req, res) => {
21     const { id } = req.params;
22     const updatedPost = await Post.findByIdAndUpdate(id, req.body, { new: true });
23     res.json(updatedPost);
24 });
25
26 // Delete a post
27 router.delete('/:id', async (req, res) => {
28     const { id } = req.params;
29     await Post.findByIdAndDelete(id);
30     res.status(204).send();
31 });
32
33 module.exports = router;
```

BACKEND: POST ROUTE

- PUT /:id: updates existing posts
- The id of the post to be updated is extracted from the request parameters ('req.params').
- Post.findByIdAndUpdate(id, req.body, { new: true }) is used to find the post by its ID and update it with the new data provided in the request body.
- The { new: true } option ensures that the updated document is returned.
- The updated post is sent back to the client in JSON format.

```
feddit-backend > src > routes > posts.js > router.get('/') callback
1  const express = require('express');
2  const Post = require('../models/Post');
3
4  const router = express.Router();
5
6  // Create a post
7  router.post('/', async (req, res) => {
8      const newPost = new Post(req.body);
9      await newPost.save();
10     res.status(201).json(newPost);
11 });
12
13 // Get all posts
14 router.get('/', async (req, res) => {
15     const posts = await Post.find();
16     res.json(posts);
17 });
18
19 // Update a post
20 router.put('/:id', async (req, res) => {
21     const { id } = req.params;
22     const updatedPost = await Post.findByIdAndUpdate(id, req.body, { new: true });
23     res.json(updatedPost);
24 });
25
26 // Delete a post
27 router.delete('/:id', async (req, res) => {
28     const { id } = req.params;
29     await Post.findByIdAndDelete(id);
30     res.status(204).send();
31 });
32
33 module.exports = router;
```

BACKEND: POST ROUTE

- DELETE /:id: allows clients to delete a post.
- The id of the post to be deleted is extracted from the request parameters.
- Post.findByIdAndDelete(id) is used to find and delete the post by its ID.
- A response with a status code of 204 (No Content) is sent back to indicate that the deletion was successful and there is no content to return.

```
feddit-backend > src > routes > posts.js > router.get('/') callback
1  const express = require('express');
2  const Post = require('../models/Post');
3
4  const router = express.Router();
5
6  // Create a post
7  router.post('/', async (req, res) => {
8      const newPost = new Post(req.body);
9      await newPost.save();
10     res.status(201).json(newPost);
11 });
12
13 // Get all posts
14 router.get('/', async (req, res) => {
15     const posts = await Post.find();
16     res.json(posts);
17 });
18
19 // Update a post
20 router.put('/:id', async (req, res) => {
21     const { id } = req.params;
22     const updatedPost = await Post.findByIdAndUpdate(id, req.body, { new: true });
23     res.json(updatedPost);
24 });
25
26 // Delete a post
27 router.delete('/:id', async (req, res) => {
28     const { id } = req.params;
29     await Post.findByIdAndDelete(id);
30     res.status(204).send();
31 });
32
33 module.exports = router;
```

CHALLENGES & TESTING

- Challenges:
 - Dynamic vote management.
 - CSS layout consistency.
 - Implementing share functionality.
- Testing:
 - Manual testing of upvote/downvote, comments, share, and navigation.

FUTURE ENHANCEMENTS

- Dark mode and nested comments.
- Redux for advanced state management.

CONCLUSION

- Created a dynamic Reddit-like SPA.
- Key skills: React, state management, component-based design.
- Implemented user authentication features.
- Implemented CRUD operations for posts.
- Overcame challenges in functionality and design.

THANK YOU