WEB APPLICATION DEVELOPMENT

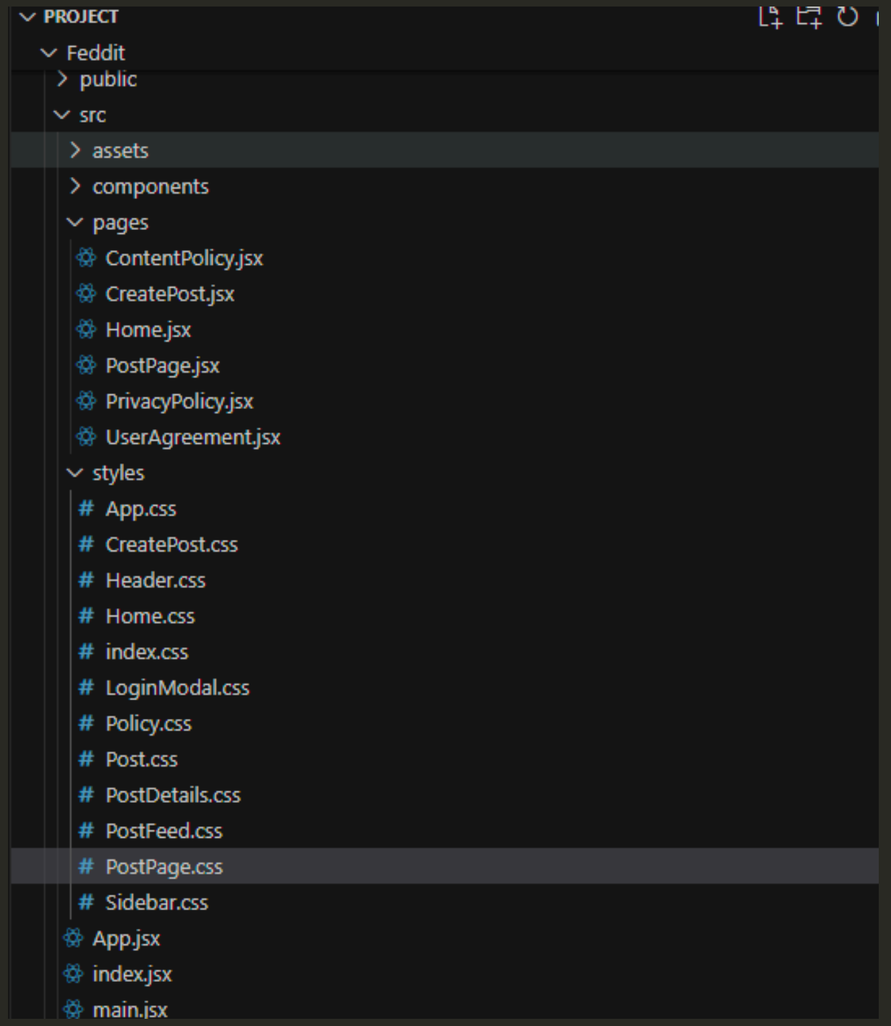
FEDDIT FORUM

PRESENTATION OF GROUP DQ

TABLE OF CONTENT

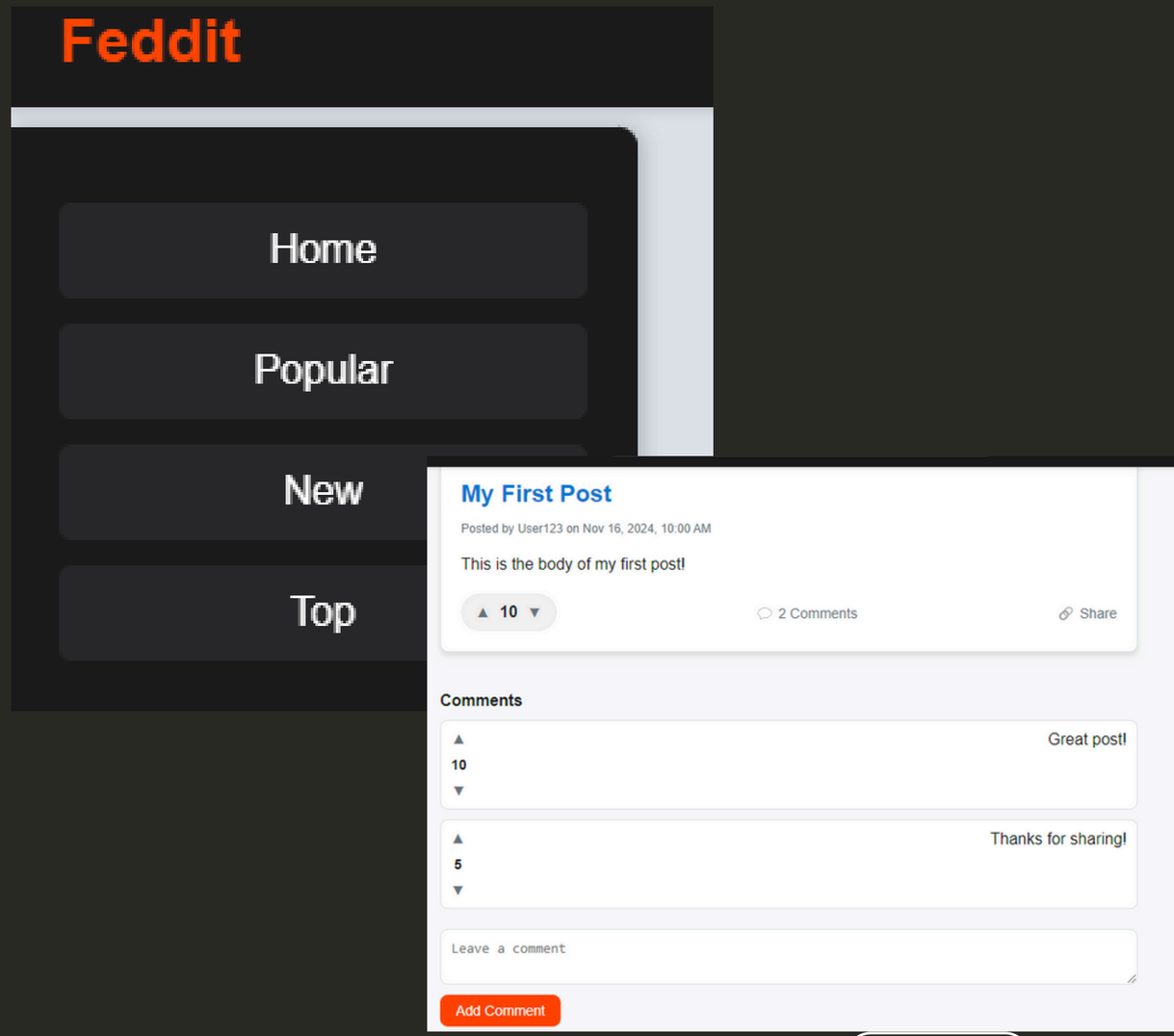
PROJECT ARCHITECTURE

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



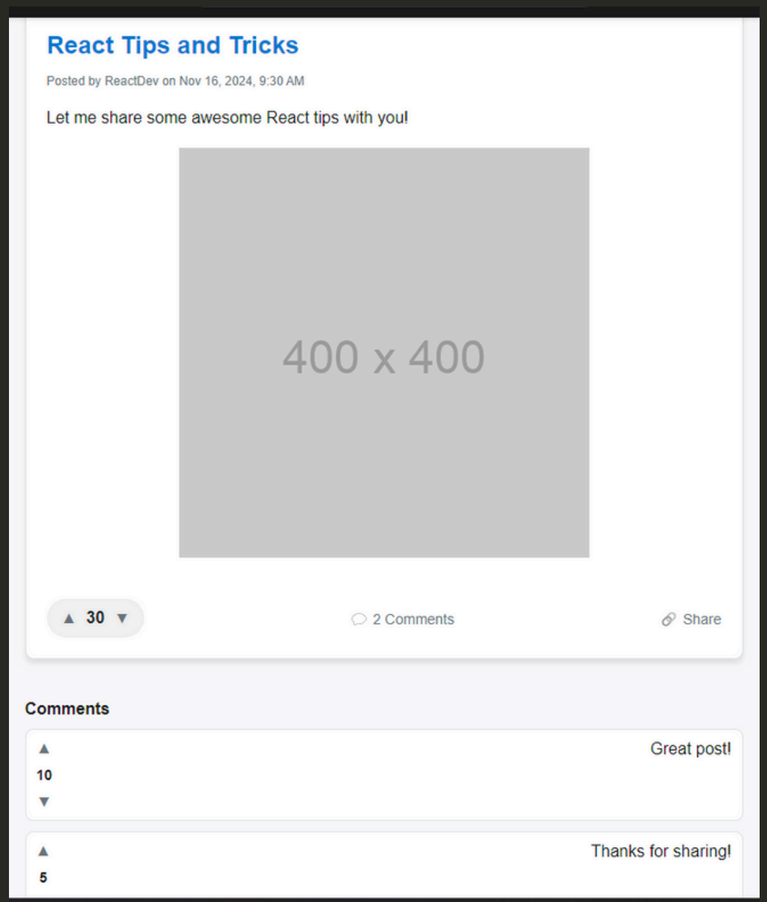
CORE FEATURES

- 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 > ...

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const dotenv = require('dotenv');
const routes = require('./routes');


dotenv.config();

const app = express();
const PORT = process.env.PORT || 5001;

// Middleware
app.use(cors());
app.use(express.json());
routes(app);

// Connect to MongoDB
mongoose.connect('mongodb+srv://vuvandovuvan:Blackiscool123@cluster0.rcltcol.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0')
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.log(err));

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});

console.log('MongoDB URI:', process.env.MONGODB_URI);

//Test server
app.get('/',(req, res) => {
  return res.send('Server is running');
})
```

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 > ...

```
// models/Comment.js
const mongoose = require('mongoose');

const commentSchema = new mongoose.Schema({
  body: { type: String, required: true }, // Body of the comment
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User ', required: true }, // Reference to the User model
  post: { type: mongoose.Schema.Types.ObjectId, ref: 'Post', required: true }, // Reference to the Post model
  timestamp: { type: Date, default: Date.now }, // Timestamp for when the comment was created
  upvotes: { type: Number, default: 0 }, // Number of upvotes for the comment
});

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

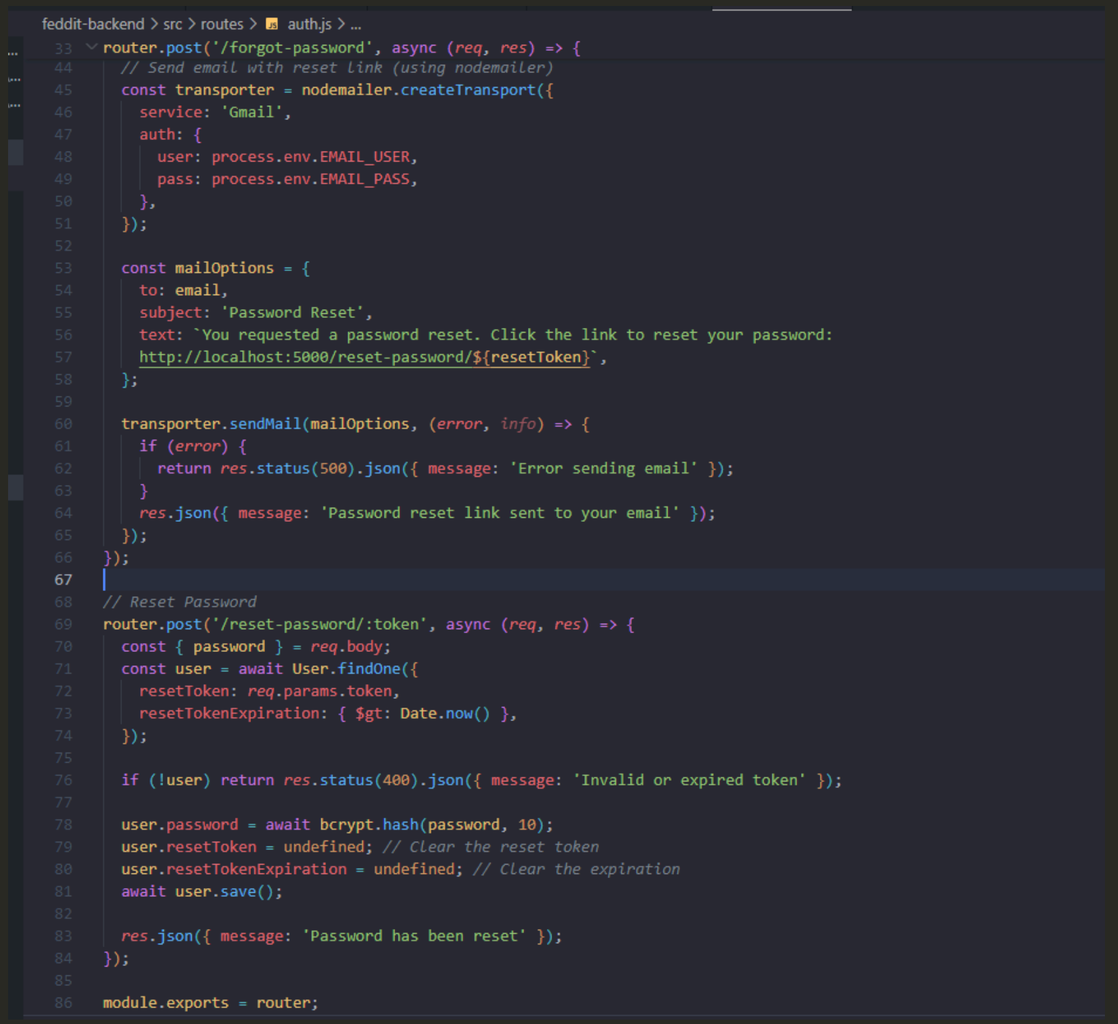
- POST /register: allows new user to register.
- Extracts username and password from request body.
- hashes the password using bcrypt.
- Saves to database and returns status code.

- POST /login: Checks username and compare password.
- Compares provided password with stored hased password.
- Returns JWT to client if successsful.

BACKEND: AUTHENTICATION

- POST /forgot-password: allows user to request password resset.
- Checks user's email and generate reset token.
- Sends email with link to resset using nodemailer.

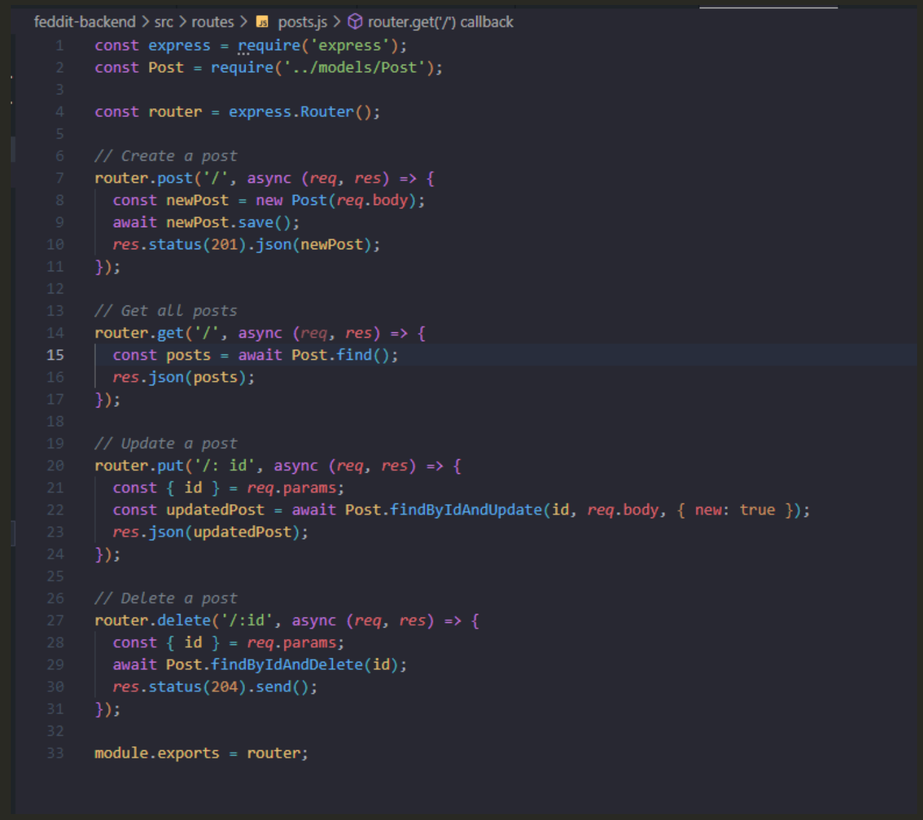
- POST /reset-password: Resets password using token.
- Checks validity and expiration of token.
- Hashes new password and update in the database.
- Removes reset token to prevent reuse.



BACKEND: POST ROUTE

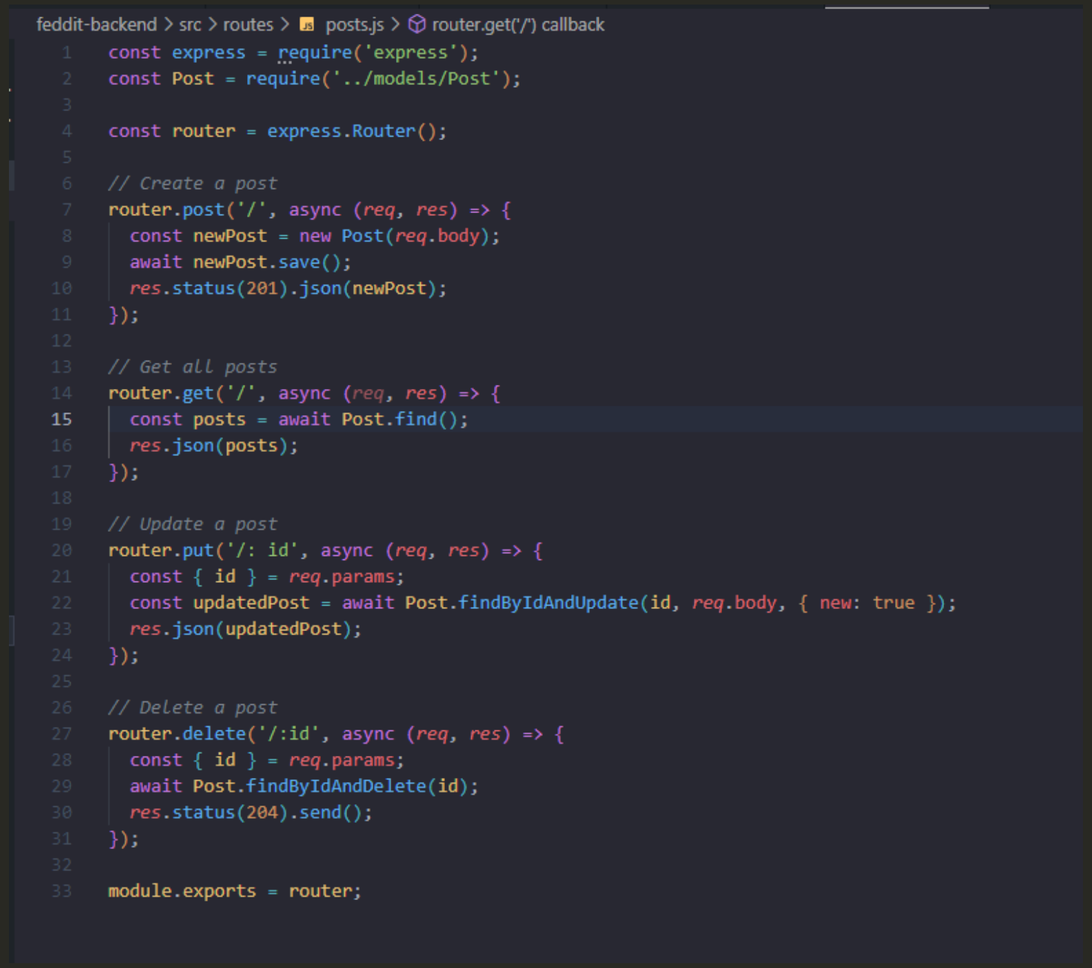
- POST /: allows clients to create a new post.
- Creates a new instance using Post Model.
- Saves post to database using ‘await newPost.save()’
- If successful, returns status code 201 with created post in JSON format.

- 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.



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.



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