

**Lesson 5-Building RESTful APIs with MERN Stack and Redux Toolkit**

**Activity 7 –PostIT – Post Message**

The objectives of this activitiy are to:

1. Implement the posting of messages from the user and save these to the backend.
2. Fetch the posts from the backend and display them on the client side.

**IMPLEMENTING THE USER POST MESSAGE**

**SERVER SIDE**

1. Create a model to represent the posts made by the users. In server/Models, we create a new file: PostModel.js.

import mongoose from "mongoose";

const PostSchema = mongoose.Schema({

  postMsg: {

    type: String,

    required: true,

  },

  email: {

    type: String,

    required: true,

  },

  likes: {

    count: { type: Number, default: 0 },

    users: { type: [String], default: [] }

  },

 },

  {

    timestamps: true

  }

);

const PostModel = mongoose.model("posts", PostSchema);

export default PostModel;

**Note for the users field:**

type: [String]: This indicates that the "users" field should be an array where each element is of type String. In other words, it expects an array of strings.

default: []: This sets the default value for the "users" field to an empty array.

**Example data for this model:**

[

{

"postMsg": "This is my first post!",

"email": "user1@example.com",

"likes": {

"count": 2,

"users": ["user2@example.com", "user3@example.com"]

}

},

{

"postMsg": "Learning MERN stack is fun!",

"email": "user2@example.com",

"likes": {

"count": 3,

"users": ["user1@example.com", "user3@example.com", "user4@example.com"]

}

},

]

1. In **server/index.js**:
   1. Import the PostModel

import PostModel from "./Models/PostModel.js";

* 1. Create the save post API.

//POST API - savePost

app.post("/savePost", async (req, res) => {

    try {

      const postMsg = req.body.postMsg;

      const email = req.body.email;

      const post = new PostModel({

        postMsg: postMsg,

        email: email,

      });

      await post.save();

      res.send({ post: post, msg: "Added." });

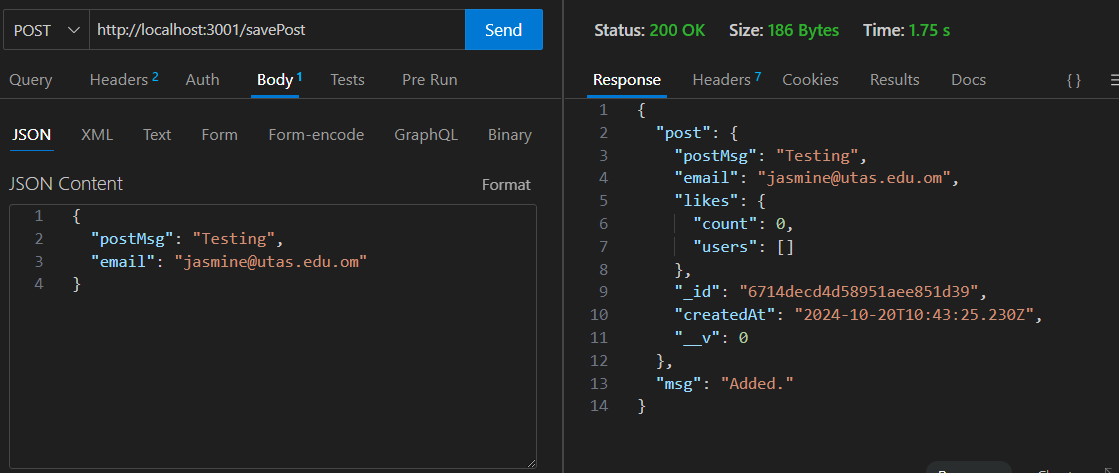
    } catch (error) {

      res.status(500).json({ error: "An error occurred" });

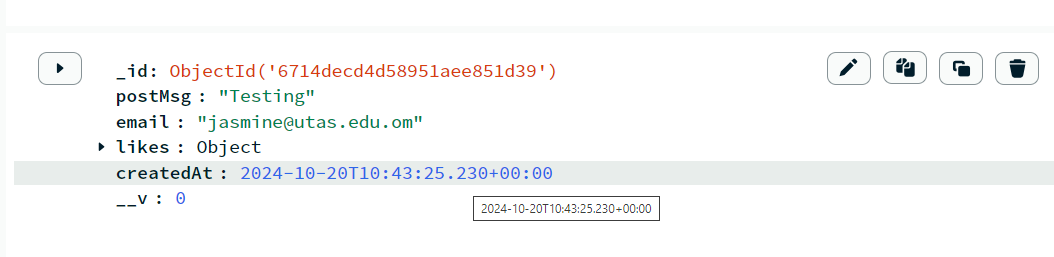
    }

  });

1. Test the API in Thunder Client, provide example request body in JSON format and send the request.

****

1. You may check also in MongoDb posts collection the data is saved.

****

**CLIENT SIDE**

1. In **src/Features** create a new file: **PostSlice.js**. In this file do the following:
2. Import createSlice, createAsyncThunk and axios.

import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";

import axios from "axios";

1. Create the initial state.

const initialState = {

status: "idle",

    posts: [],

    comments: [],

    likes: [],

  };

1. Create the Posts slice.

const postSlice = createSlice({

    name: "posts",

    initialState: initialState,

    reducers: {},

    extraReducers: ()=>{

    }

  });

1. Create the save post thunk.

export const savePost = createAsyncThunk("posts/savePost", async (postData) => {

    try {

      const response = await axios.post('http://localhost:3001/savePost', {

        postMsg: postData.postMsg,

        email: postData.email,

      });

      const post = response.data.post;

      return post; //Return the new post to Redux

    } catch (error) {

      console.log(error);

    }

  });

The code sends a request to the server through the API URL: [**http://localhost:3001/savePost**](http://localhost:3001/savePost).

The **postData** variable contains the postMsg and email, which are being sent in the request body to the backend.

The **response.data.post** is the response from the backend API with the post object.

The line return post; in your savePost function is there to pass the new post data back to Redux. When createAsyncThunk completes successfully, it returns a "fulfilled" action that contains the result of the asynchronous operation, which in this case is post.

1. Create the extrareducer for the savePost thunk. Then, export the postSlice.

const postSlice = createSlice({

    name: "posts",

    initialState,

    reducers: {},

    extraReducers: (builder) => {

      builder

        .addCase(savePost.pending, (state) => {

          state.status = "loading";

        })

        .addCase(savePost.fulfilled, (state, action) => {

          console.log(action.payload);

          state.status = "succeeded";

          // Update the state with fetched posts adding the latest post in the beginning

          state.posts.unshift(action.payload);

        })

        .addCase(savePost.rejected, (state, action) => {

          state.status = "failed";

          state.error = action.error.message;

        });

    },

  });

  export default postSlice.reducer;

1. Add this slice to the store in Redux store **src/Store/store.js**.

import { configureStore } from "@reduxjs/toolkit";

import usersReducer from "../Features/UserSlice";

import postReducer from "../Features/PostSlice";

export const store = configureStore({

  reducer: {

    users: usersReducer,

    posts: postReducer,

  },

});

1. In **SharePost** Component, we will dispatch the action in this component.
2. Import the following:

import { useSelector, useDispatch } from "react-redux";

import { useNavigate } from "react-router-dom";

import { useState } from "react";

1. Create a state variable for the user input.

const SharePosts = () => {

    const [postMsg, setpostMsg] = useState("");

    …

    }

1. Create variables for navigate and dispatch.

const SharePosts = () => {

    const [postMsg, setpostMsg] = useState("");

    const navigate = useNavigate();

    const dispatch = useDispatch();

    …..

    }

1. Get the current value of the user email from the Redux store.

const SharePosts = () => {

    const [postMsg, setpostMsg] = useState("");

    const navigate = useNavigate();

    const dispatch = useDispatch();

    const email = useSelector((state) => state.users.user.email)

    …..

    }

1. Import the savePost async thunk from the PostSlice file.

import { savePost } from "../Features/PostSlice";

1. Define the **handlePost()** function, which will be invoked when the user clicks the 'Share Post' button..

const handlePost = async () => {

    // Validate that postMsg is not empty

    if (!postMsg.trim()) {

      alert("Post message is required."); // Display an alert or set an error state

      return; // Exit the function early if validation fails

    }

    const postData = {

      postMsg: postMsg,

      email: email,

    };

    dispatch(savePost(postData)); // Dispatch the savePost thunk from the Posts Slice.

  };

1. Call the function in the share post button.

<Button onClick={()=>handlePost()}> PostIT</Button>

1. Clear the textarea after the user sent the post.
   1. The textarea must be a controlled component. Ensure that the value prop of the textarea is correctly bound to the postMsg state.

<Input

…

value={postMsg}

onChange={(e) => setpostMsg(e.target.value)}

…

/>

1. In the handlePost function, reset the value of postMsg to an empty string ("") after the dispatch call.

const handlePost = async () => {

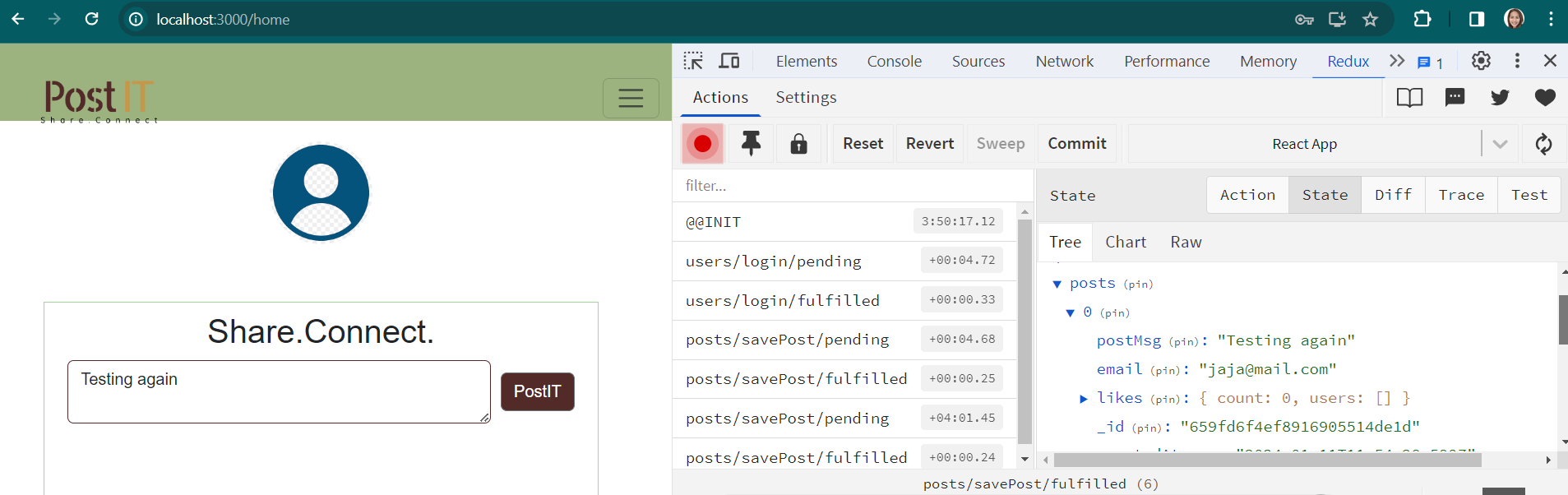
    …

        dispatch(savePost(postData)); // Dispatch the savePost thunk from the Posts Slice.

    setpostMsg(""); //clear the text area after posting

      };

1. Test your action using Redux Dev Tool.



**IMPLEMENTING THE DISPLAY POSTS**

**SERVER SIDE**

1. In **server/index.js**.
2. Create the API to get all the posts.

//GET API - getPost

app.get("/getPosts", async (req, res) => {

    try {

      // Fetch all posts from the "PostModel" collection, sorted by createdAt in descending order

      const posts = await PostModel.find({}).sort({ createdAt: -1 });

      const countPost = await PostModel.countDocuments({});

      res.send({ posts: posts, count: countPost });

    } catch (err) {

      console.error(err);

      res.status(500).json({ error: "An error occurred" });

    }

  });

1. Test the API in Thunder Client.

A screenshot of a computer

Description automatically generated

**CLIENT SIDE**

1. In the Posts slice in **src/Features/PostSlice.js.** Create a new thunk.

export const getPosts = createAsyncThunk("post/getPosts", async () => {

    try {

      const response = await axios.get("http://localhost:3001/getPosts");

      return response.data.posts;

      console.log(response);

    } catch (error) {

      console.log(error);

    }

  });

**const response = await axios.get("http://localhost:3001/getPosts");**: This line makes an asynchronous HTTP GET request to http://localhost:3001/getPosts using axios. The response from the API is stored in the response variable.

**return response.data.posts;**: Returns the array of posts (extracted from response.data.posts) as the fulfilled result of the getPosts action. In Redux, this data will be accessible as action.payload in any reducers handling the fulfilled state of this action.

1. Create extrareducers for the getPosts thunk function.

extraReducers: (builder) => {

    builder

….

      .addCase(getPosts.pending, (state) => {

        state.status = "loading";

      })

      .addCase(getPosts.fulfilled, (state, action) => {

        state.status = "succeeded";

        // Update the state with fetched posts

        console.log(action.payload);

        state.posts = action.payload;

      })

      .addCase(getPosts.rejected, (state, action) => {

        state.status = "failed";

        state.error = action.error.message;

      });

  },

});

1. In **src/Components/Posts.js**, import the following:

import { useSelector, useDispatch } from "react-redux";

import { Link, useNavigate } from "react-router-dom";

import { useEffect, useState } from "react";

import { getPosts } from "../Features/PostSlice";

import { Table } from "reactstrap";

1. Retrieve the current values of the state from the Redux Store.

const Posts = () => {

    const posts = useSelector((state) => state.posts.posts);

    const email = useSelector((state) => state.users.user.email);

    const userId = useSelector((state) => state.users.user.\_id);

  ….

  }

1. Declare variable for dispatch and navigate.

const Posts = () => {

    const posts = useSelector((state) => state.posts.posts);

    const email = useSelector((state) => state.users.user.email);

    const navigate = useNavigate();

    const dispatch = useDispatch();

  …..

  }

1. In the useEffect() hook, dispatch the getPosts action so that all posts are retrieved when the page loads.

const Posts = () => {

    ….

      useEffect(() => {

        dispatch(getPosts());

      }, []);

    ….

    }

1. Iterate through the posts array using the map() method and display the results in a table.

return (

    <div className="postsContainer">

      <Table className="table table-striped">

        <thead></thead>

        <tbody>

          {posts.map((post) => (

            <tr key={post.\_id}>

              {/\* Ensure to add a unique key for each row \*/}

              <td>{post.email}</td>

              <td>

                <p>{post.createdAt}</p>

                {post.postMsg}

              </td>

            </tr>

          ))}

        </tbody>

      </Table>

    </div> /\* End of posts \*/

  );

};

You should see all the posts.

A screenshot of a computer

Description automatically generated

1. In the client folder, install moment package. Moment is a JavaScript date library for parsing, validating, manipulating, and formatting dates.

**npm install moment**

We will use this to format to create date from this:

A white rectangular object with black text

Description automatically generated

To this:

A close-up of a white background

Description automatically generated

1. Import the moment library.

import moment from "moment";

1. Use the **fromNow()** method of the moment object.

<td>

                <p> {moment(post.createdAt).fromNow()}</p>

                {post.postMsg}

</td>