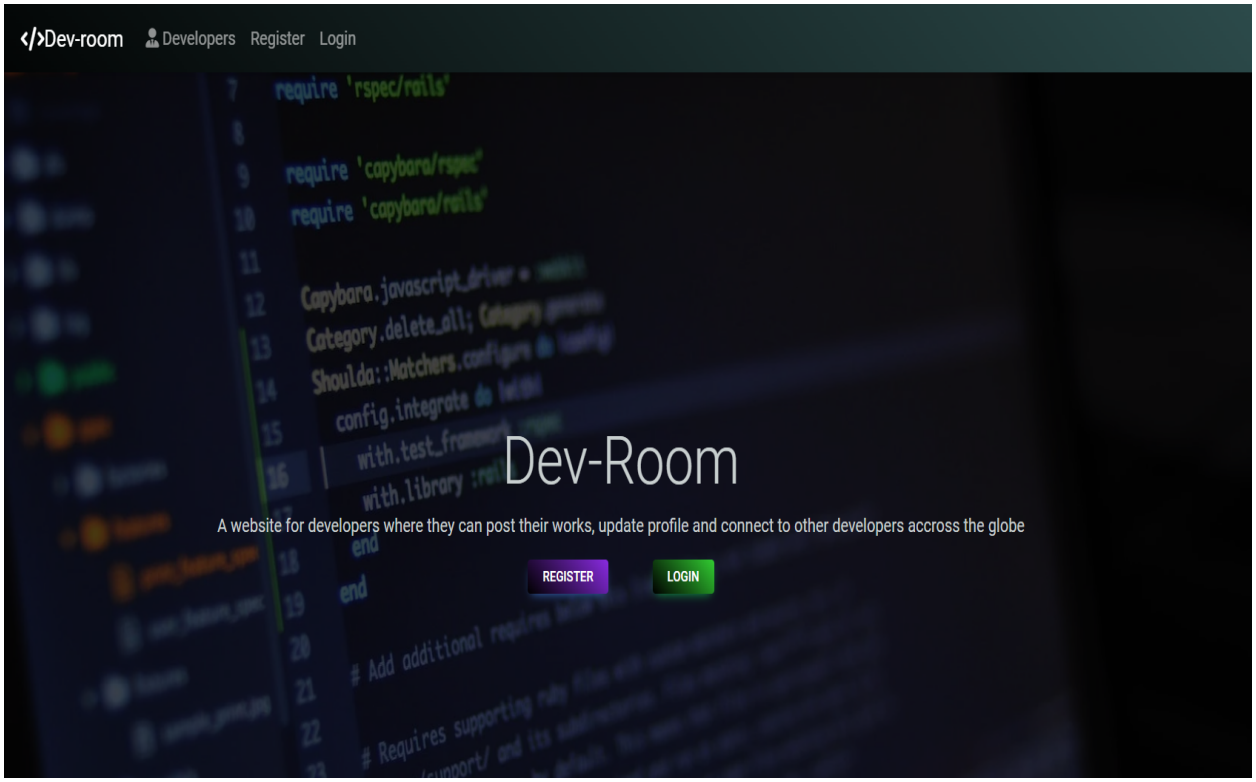

Dev-room

Project Link : https://github.com/jaydip1235/placewit-js/tree/master/Dev-room

Jaydip Dey
Jadavpur University
https://linktr.ee/jaydipdey

Its a website for developers where they can post their works, update profile and connect to other developers across the globe

Functionalities

1. Users can login and register.

2. They can create , update and delete the profiles , experience and education to keep it latest updated.

3. Users can also view the profile of other developers registered in this platform.

4.They can also post text, post text with image, like and comment on the post and can also view the post posted by other developers.

5.User can also delete their own posts and comments.

6. Users can follow/unfollow other users and see the posts of the users which they followed (If you did Task-8 properly)

BACKEND

index.js

Introduction

The provided code appears to be the primary entry point for a Node.js-based server-side application using the Express.js framework. The application leverages additional packages including CORS (for handling cross-origin resource sharing), dotenv (for managing environment variables), mongoose (for interaction with MongoDB), and body-parser (for handling HTTP request body parsing).

Details

Middleware Configuration

The application begins with the configuration of middleware.

The CORS middleware is used to enable Cross-Origin Resource Sharing from all origins by default. This is necessary when the server needs to handle requests from different domains, protocols, or ports than its own.

The body-parser middleware is used to parse incoming request bodies in a middleware before the handlers. It exposes these parsed request bodies on req.body. The bodyParser.json() and bodyParser.urlencoded() methods are used to parse JSON and URL-encoded data, respectively. The configuration allows large payloads, up to 50MB.

Environment Variable Configuration

Dotenv is used to load environment variables from a .env file into process.env. This aids in managing project settings in a secure and scalable manner. It's especially useful for managing sensitive data (e.g., API keys, database URLs, etc.) that should not be included directly in the code.

Database Configuration

The mongoose package is used to establish a connection to a MongoDB database. The database URL is retrieved from the environment variables, and options for useNewUrlParser and useUnifiedTopology are passed to ensure the usage of the new URL string parser and unified topology features of the MongoDB Node.js driver.

Router Configuration

The application is set up to handle routes related to users, posts, and profiles. These routes are presumably defined in separate modules, which are imported and used as middleware for their respective base URLs.

authenticate.js middleware

The authenticate.js file contains a middleware function in Node.js that is designed to perform authentication checks on incoming HTTP requests. The middleware is used in conjunction with the JSON Web Token (JWT) technology to authenticate users.

Function: authenticate

This middleware function is used for authenticating and verifying incoming requests in the system.

Implementation Details

1. Token Extraction - This function first checks if the Authorization header is present in the request and whether it begins with 'Bearer'. If it does, it splits the header value to extract the token.

2. Unauthorized Error - If no token is present, the function sends a 401 HTTP status (Unauthorized) and a JSON response body with a message "User unauthorized".
3. Token Verification - The function then attempts to verify the extracted token using jwt.verify(). This requires the environment variable JWT_SECRET_KEY to be defined. If the token is successfully verified, the request.user property is set to the decoded user information from the token. A log of the user information is then printed to the console.
4. Error Handling - If an error occurs during any of these steps, such as if the token cannot be verified, a 500 HTTP status (Internal Server Error) is sent along with a JSON response body containing the message "Invalid Token". This helps ensure that the system can gracefully handle authentication failures.

Dependencies

This file relies on the following npm package:

- jsonwebtoken: This package is used for token generation and verification.

userRouter.js

- User Registration: This feature provides an API endpoint (/api/users/register) that allows users to register themselves into the system. The input fields required for registration are name, email, and password. The system checks if the user already exists based on their email. If the user is new, the password is hashed using bcrypt before storing it into the database. A default avatar is also assigned to the user.
- User Login: This feature provides an API endpoint (/api/users/login) that allows registered users to log in. The system checks if the provided email exists in the database, and then it validates the entered password against the stored hashed password using bcrypt. Upon successful login, a JSON Web Token (JWT) is generated and sent back to the client. This JWT can be used for authentication in subsequent requests.
- User Information Retrieval: This feature provides an API endpoint (/api/users/me) that allows the retrieval of user information. This endpoint is authenticated using the middleware 'authenticate'. It returns user information excluding the password field.

profileRouter.js

- Get User Profile (GET /api/profiles/me) - This function retrieves the authenticated user's profile and sends it as a response. If no profile is found, a message indicating this is sent. This endpoint is private and requires authentication.

- Create User Profile (POST /api/profiles/) - This function allows an authenticated user to create their profile. User can input details like company, website, location, designation, skills, bio, GitHub username, and various social media handles. Once the profile is created, a success message along with the created profile is returned.
- Update User Profile (PUT /api/profiles/) - Authenticated users can update their profiles. Similar to the Create Profile function, user can update various details of their profile. If a profile doesn't exist for the authenticated user, an error message is returned. Otherwise, a success message along with the updated profile is sent.
- Get Profile by User ID (GET /api/profiles/users/:userId) - This function retrieves a user's profile by the user ID. It is a public endpoint and doesn't require authentication.
- Add Experience to Profile (PUT /api/profiles/experience/) - Authenticated users can add their work experience to their profile. They can input details like title, company, location, and a description.
- Delete Experience from Profile (DELETE /api/profiles/experience/:expId) - Allows an authenticated user to delete an experience from their profile using the experience ID.
- Add Education to Profile (PUT /api/profiles/education/) - Authenticated users can add their educational background to their profile. They can input details like school, degree, field of study, and a description.
- Delete Education from Profile (DELETE /api/profiles/education/:eduId) - Allows an authenticated user to delete an education entry from their profile using the education ID.
- Get All Profiles (GET /api/profiles/all) - This public endpoint retrieves all the profiles in the database.
- Get Profile by Profile ID (GET /api/profiles/:profileId) - This function retrieves a profile using the profile ID. It is a public endpoint and doesn't require authentication.

postRouter.js

- Create a new Post (POST /api/posts/): This endpoint enables authenticated users to create a new post. The required fields include text and image. If the image field is empty, null or "undefined", it will default to an empty string.
- Get All Posts (GET /api/posts/): This endpoint allows authenticated users to retrieve all posts. Each post includes the associated user's _id and avatar.
- Get a Post by PostId (GET /api/posts/:postId): This endpoint allows authenticated users to retrieve a specific post by its postId.
- Delete a Post by PostId (DELETE /api/posts/:postId): This endpoint allows authenticated users to delete a specific post by its postId.

- Like/Unlike a Post (PUT /api/posts/like/:postId): This endpoint enables authenticated users to like or unlike a specific post. If the user has already liked the post, then this action will unlike the post, and vice versa.
- Create Comment to a Post (POST /api/posts/comment/:postId): This endpoint enables authenticated users to add a comment to a specific post. The required field is text.
- Delete Comment of a Post (DELETE /api/posts/comment/:postId/:commentId): This endpoint allows authenticated users to delete a specific comment of a specific post. The user must be the author of the comment to delete it.

FRONTEND

App.js

- Navbar: Navigation bar to provide navigation links.
- Home: Home page of the application.
- UserRegister: User registration form.
- UserLogin: User login form.
- DeveloperList: A list showing all registered developers.
- DeveloperDetails: Detailed view of an individual developer profile.
- Dashboard: User profile dashboard.
- CreateProfile: Form for creating a new profile.
- EditProfile: Form for editing an existing profile.
- AddEducation: Form for adding educational details.
- AddExperience: Form for adding professional experience.
- PostList: A list displaying all posts.
- PostDetails: Detailed view of an individual post.

UserRegister.jsx

- State Management: The component uses React's useState hook to manage the state of the user input and any errors that occur during the registration process. There are two main state objects - user which stores the entered name, email, and password, and userError that stores any validation errors associated with these fields.

- Form Validation: There are validation functions for each of the form's inputs - validateUsername, validateEmail, and validatePassword. They update the state of the user input and check the entered data against specified criteria using regular expressions.
- Form Submission: The submitRegistration function handles the form submission. If all fields are non-empty, it makes a POST request to the https://devroom-backend.onrender.com/api/users/register endpoint with the user's information. Depending on the status code returned, it gives the user feedback using the Swal library for pop-up alerts.
  If a user is successfully registered (status 200), they receive a success message and are navigated to the login page. If the user already exists (status 201), an error message is shown. If the fields are empty or not filled out properly, a generic error message is displayed.

UserLogin.jsx

- The component leverages the useState and useEffect hooks from React for managing the state of the form inputs and to check if a user is already logged in.
- The useNavigate hook is used from 'react-router-dom' to enable navigation to different routes based on the user's authentication status.
- When a user fills in their email and password and submits the form, the function submitLogin is triggered. This function first checks that the email and password fields are not empty. If they aren't, a POST request is made to the provided API endpoint.
- Depending on the response status from the server, the component decides whether to show a success message and navigate to the "/developers" route or to display an error message. In case of a successful login, it also saves the returned token to the local storage.

Navbar.jsx

- User Authentication: The component checks if a user is logged in or not using localStorage. If a user token ("devroom") is present in localStorage, it implies the user is logged in, otherwise not.
- User Information Fetching: If a user is logged in, it fetches the user's information using an asynchronous function (getUser()) from the provided API endpoint. This information includes user details such as avatar, which is displayed on the Navbar.
- Dynamic Content Display: Depending on whether a user is logged in or not, different sets of navigation links are displayed on the Navbar. For non-logged-in users, the links are to "Register" and "Login" pages. For logged-in users, the links are to "Posts", "Dashboard", and a "Logout" option.

- Logout: When a user clicks on the "Logout" link, the user token is removed from localStorage, effectively logging the user out, and then redirects the user to the "Login" page.

PostList.jsx

- User Authentication: The code checks for user authentication by looking for a specific token (devroom) in the local storage. If the token does not exist, it redirects the user to the login page.
- Post Retrieval: The application retrieves the posts from the API after the user has been authenticated. This is done through an asynchronous function (getPosts) which makes a GET request to the API. The posts are then set to the component state.
- User Retrieval: Information about the current user is fetched through another asynchronous function (getUser) that makes a GET request to the user endpoint of the API. This data is used to populate user-related information in the UI.
- Create Post: Users can create a new post using the submitCreatePost function, which makes a POST request to the API. After the post is created successfully, the posts list is updated and the input fields are reset.
- Delete Post: The code enables users to delete their own posts. This is done through the clickDeletePost function which sends a DELETE request to the API and then updates the posts list in the component state.
- Like Post: Users can also like posts using the clickLikePost function. This function sends a PUT request to the API to update the post, and then updates the posts list in the component state.
- Time Difference: The timeDifference function calculates the time difference between the current time and the post creation time, and presents it in a user-friendly format.

PostDetails.jsx

- Uses React hooks (useState, useEffect, useParams, and useNavigate from react-router-dom) and Axios for HTTP requests.
- The component starts by initializing state variables for comment, user, selectedPost, loading, and loggedIn.
- useParams().postId is used to get the post id from the URL parameters and useNavigate is used for programmatic navigation.
- An effect hook is used to check if a user is logged in by checking if a local storage item 'devroom' exists. If it doesn't exist, the user is navigated to the login page.
- A timeDifference function is declared to calculate the time elapsed from a given timestamp to the current time and returns a human-readable string describing the time elapsed.

- getUser and getPost asynchronous functions are declared to fetch the logged-in user's details and the selected post's details respectively.
- The submitCreateComment function is used to create a comment for the post and clickDeleteComment to delete a comment.
- The data loading state is handled by rendering a Spinner component while the post data is loading.
- The component renders post details, allows users to write a new comment, and shows existing comments. It also allows the user to delete their comments. Comments and posts display the time elapsed since their creation.

CreateProfile.jsx

- Navigation: Utilizes useNavigate from react-router-dom to manage the navigation of the application.
- Data Management: Utilizes useState and useEffect from React for state management and side effects respectively. State is primarily used for storing the profile form data.
- Local Storage: Checks the local storage for an item "devroom". If not present, redirects the user to the login page.
- Image Upload: Provides an image upload feature with a function uploadImage. It converts uploaded files to base64 format using FileReader.
- Form Submission: The form data is submitted through the submitCreateProfile function which sends a POST request to the backend API.
- Alerts: Uses Swal.fire from sweetalert2 to give a success message after the profile is created.

EditProfile.jsx

- An async function uploadImage is implemented to convert an uploaded image file to a base64 string. This is stored in the localProfile state.
- The getProfile and getUser functions fetch the user's profile data from the API. The fetched data is then used to set the localProfile state.
- The updateInput function handles changes to the form inputs by updating the corresponding fields in the localProfile state.
- The submitUpdateProfile function sends a PUT request to update the user's profile with the current data in localProfile, displays a success message using Swal, and redirects the user to the dashboard.

Dashboard.jsx

- User Verification: The code ensures that only authenticated users are allowed to view their dashboard. If no user credentials are found in local storage, the user is redirected to the login page.
- User and Profile Retrieval: On successful login, the code makes asynchronous HTTP GET requests to fetch the user and their corresponding profile information from the backend server.
- Loading Spinner: A spinner is displayed while fetching data from the server, providing visual feedback to the user during the loading process.
- Dashboard Display: The dashboard presents the user with details about their profile, experience, and education, along with options to edit their profile, add experience, and add education.
- Experience and Education Deletion: Users can delete specific entries of their experience or education by making an asynchronous HTTP DELETE request to the server.

DeveloperList.jsx

- State Management: Two states are initialized - 'profiles' (an empty array) and 'loading' (a boolean set to true initially).
- Fetching Data: An async function 'fetchProfiles' is declared that makes a GET request to "https://devroom-backend.onrender.com/api/profiles/all" using axios. The response is used to set the 'profiles' state and 'loading' is set to false indicating that data fetching is complete.
- Rendering: If 'loading' is true, a Spinner component is rendered indicating that data is being loaded. Once the data is fetched, a list of developer profiles is displayed. Each profile contains the user's avatar, name, website, designation, company, location, and their skills. Each listed profile has a 'View Profile' button that routes to a more detailed view of the developer's profile.

DeveloperDetails.jsx

- Use of React Hooks: The component uses useState and useEffect React hooks for managing the state of the application and performing side-effects (asynchronous data fetching).
- Fetching of Data: Data is fetched from an API using the Axios library. The URL for the API is dynamically constructed using a developerId which is derived from the React Router useParams hook. This allows the application to fetch data specific to the selected developer.

- Loading Spinner: The application includes a loading spinner (the Spinner component), which is displayed while data is being fetched from the API.
- Data Presentation: Upon receiving the data, the application presents the developer's profile information in a well-structured and detailed format. This includes the developer's personal details (like name, website, designation, company, and location), social media links, a biography, skills, work experiences, and education history.
- External Linking: The application provides links to the developer's social media profiles via the react-external-link library.
- Dynamic Rendering: The component contains logic to handle scenarios where no data is present. For example, in the case of missing experience or education data, the component gracefully handles these scenarios by not rendering the respective card components.

AddExperience.jsx

- State Management: The component uses React Hooks for state management with the useState hook. An 'education' object is initialized with keys: school, degree, fieldOfStudy, from, to, current, description, all set to empty strings.
- User Authentication: On component mount, a useEffect hook checks if there's an authentication token ("devroom") present in localStorage. If not found, the user is redirected to the login page.
- Form handling: The function 'updateInput' is used to handle the state updates when the user types into the input fields in the form. For checkboxes, the checked state is recorded, while for other inputs, the value is recorded.
- Data Submission: On form submission, the function 'submitAddEducation' sends a PUT request to the backend server. The user's education data, as stored in the component's state, is included in the request body. The 'devroom' token from localStorage is sent in the Authorization header.
- User Feedback: After successful submission of the form, a success alert is displayed using sweetalert2 and the user is navigated back to the dashboard.
- UI Elements: The component uses a form for data collection. It has several text input fields for school, degree, field of study, from date, to date, and description, a checkbox for 'current' status, and a 'Submit' button. There's also a 'Back' button to navigate back to the dashboard.

AddEducation.jsx is similar to AddExperience.jsx