**PROJECT PRESENTATION**

* What is the requirement for this project? What is the problem it is going to solve?
* Why did you select these particular tools, platform?
* Know the structure of the project.
* Should know the files, folder, libs, configs
* Should explain the workflow of the project
* Difficulties faced during project
* Future scope of the project

**DevConnector**

Introduction-

This is a social media website specifically designed for developers. I chose to make this project as I feel there are only very few social media websites that addresses and caters to the different needs of developers. This website helps in creating an online space for developers to connect.

This website allows users to register or login to the website. Once registered, a user has to set up their profile and can specify their designation, work experiences, education background, skills and their socials. Users can also link their Github account with their profile so that other users can see their recent Github repos on their profiles. There is also a feature to post on the website where other developers can like/dislike or even comment on the post. A User can also delete their post or their comments.

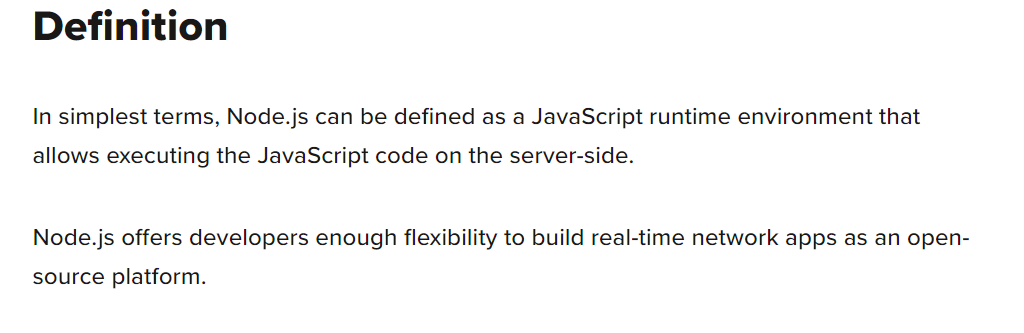
For the front-end development, I chose to use HTML, CSS, and JavaScript. I also used the React framework. I choose React because React has a component-based architecture that facilitates code reusability and maintenance. React also allows JSX implementation.

For the back-end, I used Node.js as JS runtime env and Express.js as the web application framework for NodeJs. I used NodeJs as it can handle multiple concurrent requests. Using NodeJS, we can code for backend in JavaScript, we can use JS for front-end as well as backend. I used Expressjs because it is a lightweight and flexible framework and most of the code is written in JS. I found Express.js quite easy and efficient to work with. It provides simple routing for requests. It also provides a middleware module that helps in making decisions to give correct responses. Without Express.js we have to write our own code to build a routing component which is time-consuming.

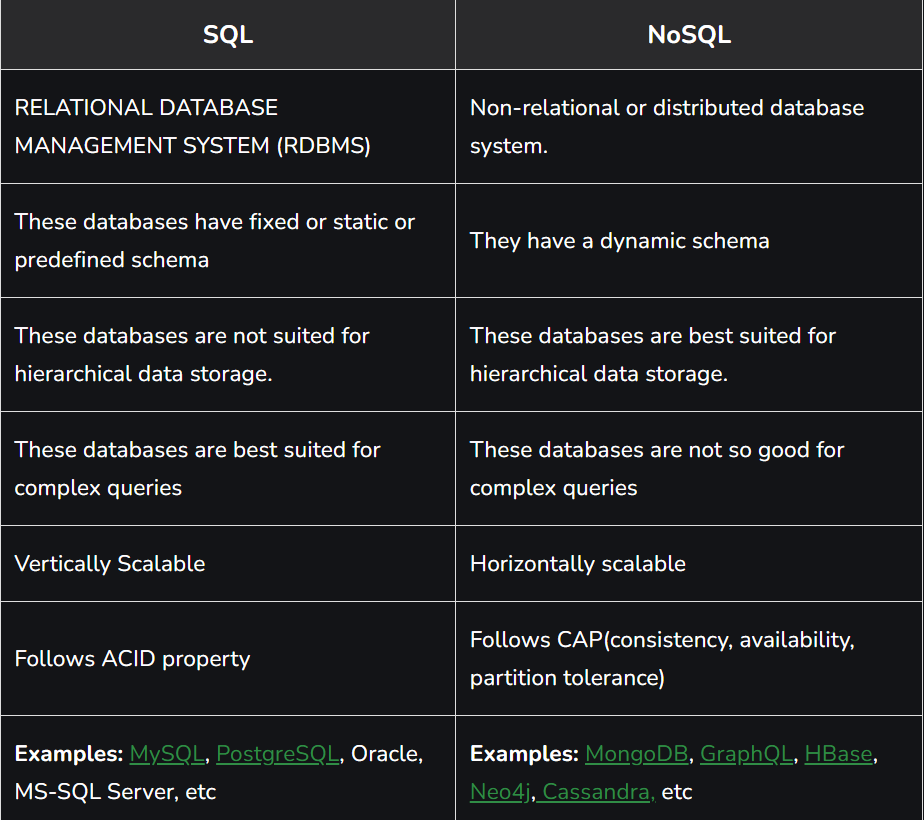
For Database, I used MongoDB , a NoSQL Database. MongoDB's document-based structure was a natural fit for the dynamic nature of developer profiles and project details. It offered the flexibility to adapt to changing requirements and provided efficient querying capabilities.

SQL requires you to use predefined schemas to determine the structure of your data before you work with it. Also, all of your data must follow the same structure. A NoSQL database has a dynamic schema for unstructured data. Data is stored in many ways which means it can be document-oriented, column-oriented, graph-based, or organized as a key-value store.

---------------------------------------------------------------------------------------------------------



NodeJs: <https://syndicode.com/blog/nodejs-for-back-end/#:~:text=The%20first%20reason%20to%20use,client%2Dside%20handles%20HTML%20rendering>.



/client

/server

1. /middleware
2. Auth.js ->middleware that authorizes access to protected routes; takes token from req header and decodes the token using jwtsecret and put it in the req.user = decoded.user so we can access req.user for user.id
3. checkObjectId.js-> to check a valid object id; used to post routes to unlike a liked post.
4. /models
   1. Post.js -> defining schema for post routes
   2. Profile.js -> defining schema for profile routes
   3. User.js -> defining schema for user routes
5. /routes/api
   1. Auth.js ->
      * Get the authenticated user; uses auth middleware to find the user.id, and find and return that user by searching that user.id in the database (Load a user)
      * Authenticate user and get token(login a user)-> check if email and password is valid using (express-validator) and then search the user in the database using the email, match the password with searched by email password using bcryptjs dependency and then sign and return a jwt with user.id as payload. (login a user)
   2. Post.js -> create a post, get all posts, get post by id, delete a post by id, like a post, unlike a post, comment on a post, delete a comment
   3. Profile.js -> get current users, profile,create/update a user profile, get all profiles, get profile by user id, delete a profile user posts, add profile experience, delete experience from profile, add profile education, delete education from profile, get user repos from github
   4. User.js ->

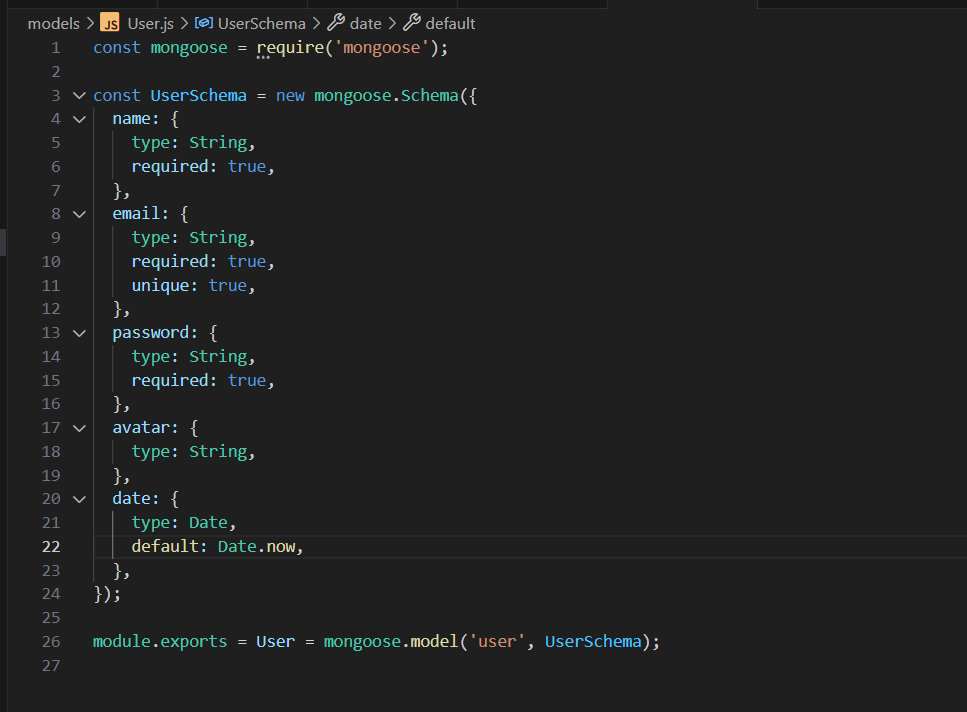
* Register a new user-> check and validate, name email password using express-validator, get name, email, password from req.body and find the user through the email, make a avatar using gravatar, create a new user object with name, email, password, then encrypt the password using bcrypt, and then sign and return jwt.

(whenever a user logs in or registers, it should sign )

**Creating User Routes->**

Before creating routes, we have to **create a model**, for User Routes we have to make a User model first to define the schema.

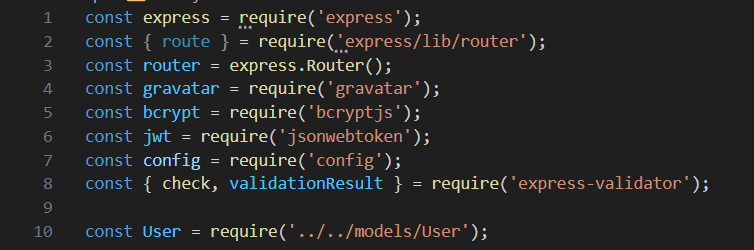
A model file will look like this.



After creating models, we will make a user routes file.

In Users Route, there is only a post request that user will make during registration, where he will send the user details to register on the website.

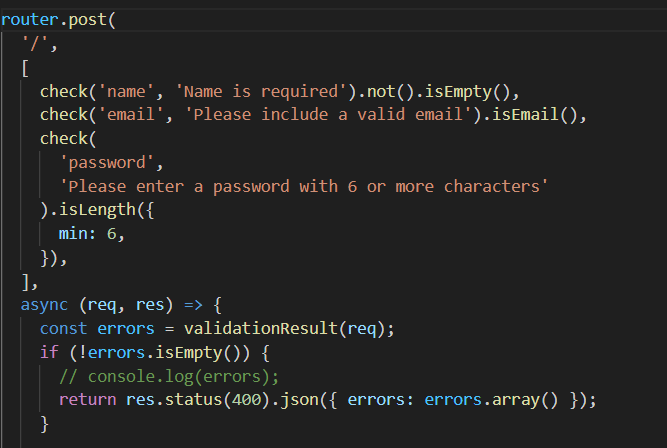
**The dependencies used**

****

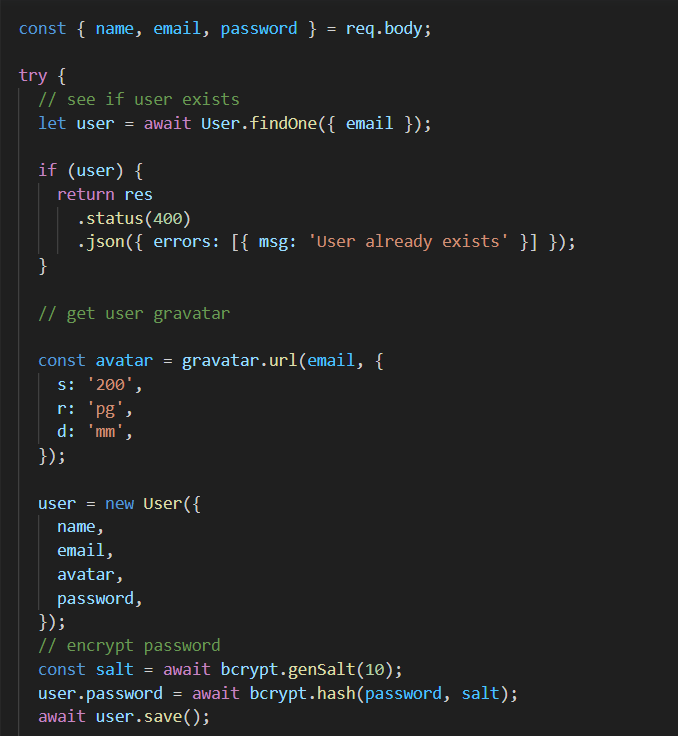
To Check whether user have entered the correct data, for e.g., name string in not empty, email string is in valid email format, password is atleast 6 digits big.

We will use **express-validator:** <https://express-validator.github.io/docs/guides/getting-started>

The check helps in checking the required format. validatonResult reports the errors to the users.



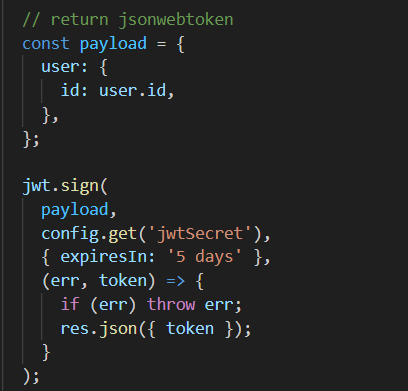
Express-validator is a set of middleware that allow us to validate express request very easily and efficiently. It has tools to determine if the req is valid or not, which data was not matched according to your validators. Otherwise we have to write multiple condition statements to check and validate req data, express validator saves us of this cumbersome job. And using it, we can all the errors at once in an array format.



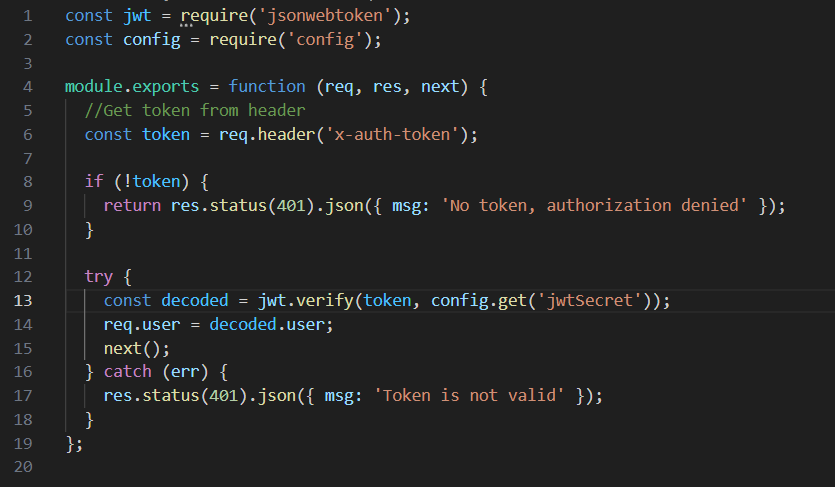
I used gravatar package to generate a user avatar

I used bcryptjs package for encrypting the password. Enables storing hashed passwords instead of plaintext. It uses bcrypt as a password hashing algo. Bcryptjs is safer because it allows to choose different iteration count for hashing, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks.

I also used JSONwebtoken so that users can authenticate and access protected routes.



We make a custom middleware to access protected routes. We sign the jwt with payload and add a special jwtsecret, and a callback function that return the token. For a protected route we send a request and jw token will be in the header. We take the token from req header and verify it.



For Login User route, we will compare the password with the encrypted password using bcryptjs. It will also return a jwt.

**Profile Routes-**

I will similarly made the profile model, the schema will contain fields like user, company, designation, skills, bio, work experience, education, etc.

There will be many routes for the profiles,

* one will get the current user’s profile it will take the **authentication middleware,** find the user in the db with req.user.id, and will return the profile with the user.id from the decoded token.
* One route for creating or updating the profile. I used express-validator again to check and validate the data entry. And made the profile object from all the fields and check if the profile already exist we will update it otherwise it will add a new profile into our database.
* One route will get all user profiles, it is not a protected route, it won’t require the auth middleware. It will simply just find all the profiles in the database and return it.
* One route will get a specific user’s profile.
* One route will delete a user’s profile, as it is a protected route, it will require a auth middleware to verify the token. First, we will delete all the posts associated with the user.id, and we will delete the profile and then delete the user with the user.id decoded by the jwttoken.
* One route will add profile experience, it is also a protected route. We will also use express-validator in this. And it will make an object of new experience field and add that in the profile of the authorized user.
* One route will delete a particular user experience, first we will find the index of the experience to remove; for that it will first get the profile with the user.id of the token from the database, then will map all the items based on their id and find the one specified by the user in the params and finally remove it.
* Similar routes for adding and removing user’s education.
* **One route get user’s recent Github Repos, its takes in the github username as input and then the backend is going to make a request to github api to get the recent github repos of the user. Make a account on github OAuth and get the clientId and cliendSecret, it defined the function such that it will get the recent 5 repos, we can change that to as many as we want. For sending request for Api, I used request package bcz it makes the making the http calls easy and convenient.**

**Post Routes**

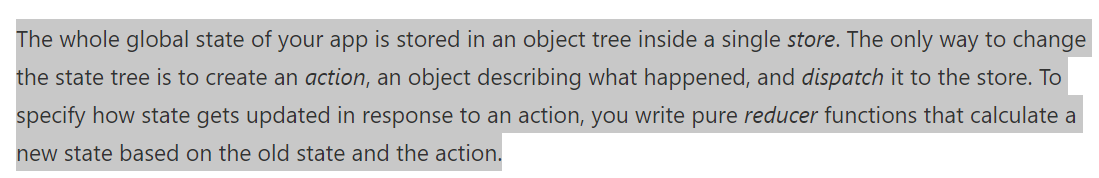
Post models are made similarly to the other models.

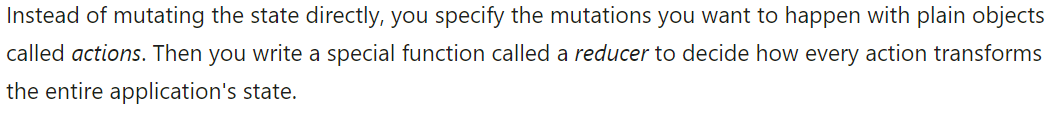
* Route for making a post – auth middleare, express-validator, find the user from database using the id decoded from token, make new object for post, and then save it into the db.
* Route for getting all posts- protected bcz only logged in users can see, return all post from db sorted in most recent(date: -1)
* Route to get a specific post by postid- protected bcz only logged in user can see, search the db with the req.params, not found then 404, otherwise return the post.
* Route to delete post, protected route, find post from the url params, check the user – if the post made by the user is not equal to the user of userid from the token, not authorization denied, otherwise delete from db.
* Route to like a post, protected route, find post using the url params, check if the post has already been liked(in the db of the specified post check among the likes if the userid of a like is equal to the user.id of the curr user) then it return that post has been liked otherwise add the like in the likes array of the curr post.
* Route to unlike a post-> protected route, find post from the db using the url params, check if the post has not been liked yet(in the db for the specified post if there doesn’t exist a like with the same user.id as the curr user.id then return that post has not been liked yet) then store all the post like other than the curr user’s in the database.
* Route to comment on a post->protected route, express-validator, take the current user with userid from the token, find the post from the database using the url params, make new obj of comment, add it to the database.
* Route to delete a comment from the post-> find the post and comment from the ids from the url params. Checks if the post exist, checks if the user is authorized to delete the comment. Find the deleteIndex from the comment array of the specified post with the comment and then remove it.

**Frontend**

The frontend structure is like this ->

The homepage has a Landing page and Navbar component. I **used React-router-dom for page routing**. I used Redux for state management rather than context-api, because it is a bigger project and I thought using context-api wouldn’t be very efficient. It also uses react-redux to connect react and redux.





#### Actions[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#actions)

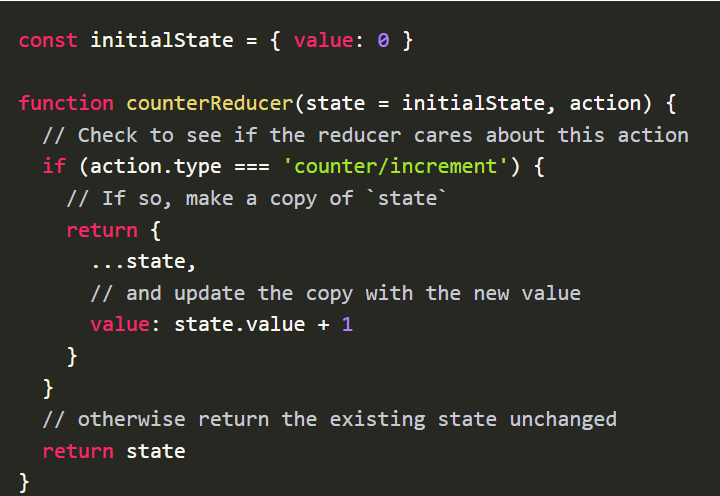
An **action** is a plain JavaScript object that has a type field. **You can think of an action as an event that describes something that happened in the application**.

The type field should be a string that gives this action a descriptive name, like "todos/todoAdded". We usually write that type string like "domain/eventName", where the first part is the feature or category that this action belongs to, and the second part is the specific thing that happened.

An action object can have other fields with additional information about what happened. By convention, we put that information in a field called payload.

#### Reducers[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#reducers)

A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: (state, action) => newState. **You can think of a reducer as an event listener which handles events based on the received action (event) type.**



**React Redux**

React Redux includes a <Provider /> component, which makes the Redux store available to the rest of your app:

React provides two major mechanisms for providing data to components, props and state. Props are read-only and allow a parent component to pass attributes to a child component. State is local and encapsulated within the component. It can change at any time in the component’s lifecycle.

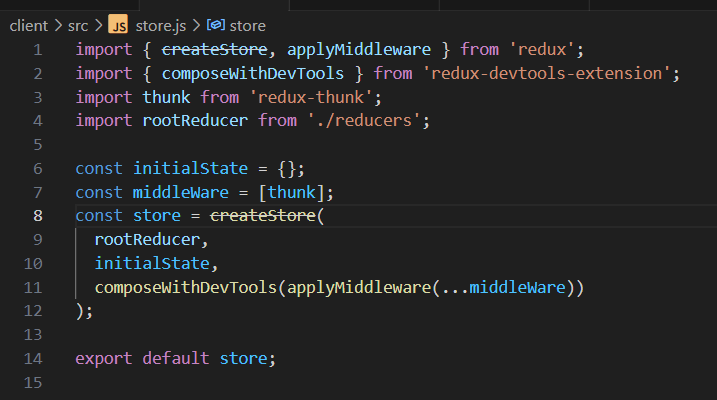
Connect() - easy to [connect a React application to a Redux store](https://react-redux.js.org/api/connect).

Presentational components are only concerned with how things look. They are not aware of the Redux state. Presentational components get their data from props and may trigger callbacks passed to them via props. – reason to map state to props

The connect() function provided by React Redux can take up to four arguments, all of which are optional. Calling the connect() function returns a higher order component, which can be used to wrap any React component.

There is a root reducer that combines all the sub reducers we make.

There is a store for redux.

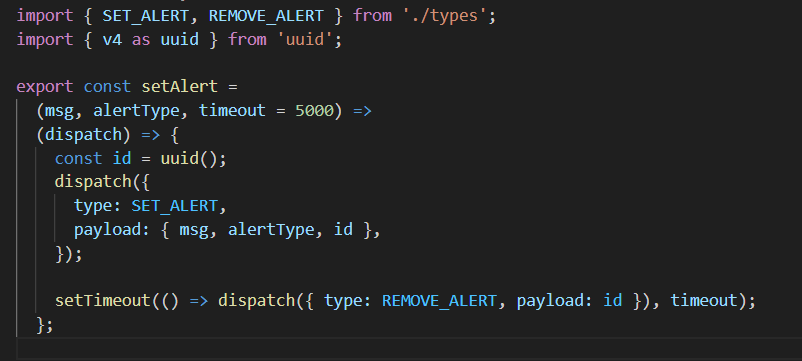


In App.js, all routes are in <switch>, the <switch> and components are in <Router> which will be inside <Provider> for redux state management.

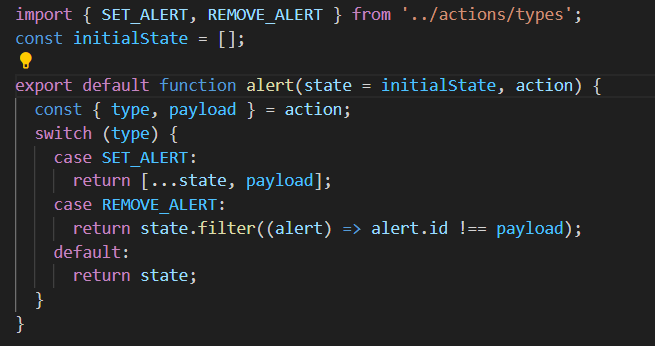


Example of alert with action, reducers, ->

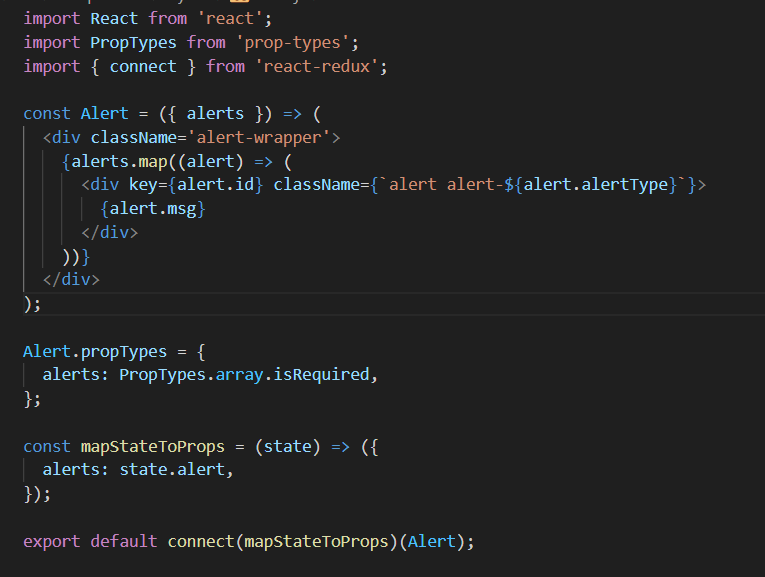
**Alert Actions**

****

**Alert Reducers-**

****

**Alert Component-**

****

Axios is a [*promise-based*](https://javascript.info/promise-basics) HTTP Client for [node.js](https://nodejs.org/) and the browser. It makes HTTP request from node.js.

Token will be stored in the local storage.

To register a user

In the action->

We will get the user’s data from the backend using axios, then we will dispatch an action with type of REGISTERSUCCESS and payload of the user’s data.

In the reducers->

Set the token in the local storage and return the state, payload. If the register fails, remove the token from the localstorage and just return the state.

To load a user->

In the actions->

If there exists a token in the localstorage, it calls a function in the utils that uses axios and put the token in the headers. And in the actions folder it gets the users data by making a axios call to the backend and dispatch the action type USERLOADED and payload as the userdata.

In the reducers->

It return the state, and the user is set to the payload if the loaduser Is success other wise it removes the user from localstorage.

In the Register component, import the register action and call it when register call is to be made.

**IN ACTIONS, USE AXIOS TO MAKE REQUEST TO THE BACKEND AND GET THE DATA, WHICH IS PASSED IN THE PAYLOAD IN THE ACTIONS AND DISPATCH IT WITH THE APPROPRIATE TYPE, IN THE REDUCERS IN THE PARTICULAR ACTION TYPE RETURN STATE AND SET NECESSARY STATE TO THE PAYLOAD.**

**Functionality**

**Load User:**

1. In the frontend, actions: get the token from the localStorage and then put it in the headers, then use axios to request to backend ‘/api/auth’.
2. In the backend auth routes, GET /api/auth, uses middleware to authorize protected route access(takes token from the headers and decodes it using jwt and then put the decoded user.id in req.user) use the req.user to find the user from the database and then return the user else return error.
3. After dispatch an action having type(USER\_LOADED) with user as the payload.
4. Now in the auth reducer, the parameters are initial state and the action(having type and payload), reducer returns the state as it is but puts the user: payload (as payload contains the user from the actions)
5. Else in case of error, dispatch an action(type: AUTH\_ERROR): which removes the token from the localStorage.

**Login a user**

1. In the frontend in the actions, we have email, password; we make a request to the backend POST(/api/auth) and send the email, password in the request.
2. In the backend routes, check the email and password using express validator and incase of error return errors, and get email, password from the req.body, find user from the email (if not found return error), check if the password matches with the one in the database using bcrypt and now return a jwtoken with user.id as the payload or else return error.
3. Get the output from the api call and store it in res. And dispatch an action(type: LOGIN\_SUCCESS) and res as the payload.
4. In the reducers, the action(LOGIN\_SUCCESS) puts the token in the localStorage and return state as it is.
5. In the actions, dispatch another action that is loadUser(). In case of error dispatch an action of type: LOGIN\_FAIL
6. In the reducer for the type LOGIN\_FAIL, remove the token from the localStorage.

**Register a user**

1. In the frontend, in the actions, there is a func register, having name, email, password as ip, stringify the ip and stores it in body, make an api call (/api/users) along with the body containing the ip and store it in ‘res’.
2. In the backend route, first checks the name, email and password using express validator, return if errors, find the user in db using email, if user already exists return user and msg that it already exists. Make a avatar of the user using gravatar dep, create a user obj with name email password avatar, encrypt the password using bcrypt and save user in the db, also sign a jwtoken with the payload as user.id and return the token.
3. After that in the actions, dispatch an action with type: REGISTER\_SUCCESS with payload as the res(token)
4. In the reducers, action type REGISTER\_SUCCESS add token into the localStorage
5. then also dispatch function (loadUser), in case of error dispatch an action REGISTER\_FAIL that removes the token from localStorage.

**Create or Update Profile**

1. In the frontend, actions, there is a function createProfile with formData as inp. Make a all to the backend (/api/profile) which returns profile store it in res.
2. Using express validator check whether the formdata is correct and req fields are not empty otherwise return error. Make obj of all the formdata and find the profile of the (user) req.user.id from db and If already present then update it, other wise create a profile obj and saves it in the db and returns the profile obj. otherwise error.
3. Dispatch an action (type: GET\_PROFILE) with the profile(‘res’) as the payload.
4. In the reducers, for type GET\_PROFILE, return the initial state as it is and in the profile return the payload(profile).
5. Else return error.

**Delete the Account**

1. In frontend, in the actions, there is a function which first makes an api call DELETE /api/profile.
2. In the backend, this call fill get the user.id from the req bcz It is a protected route and we have used middlware to authorize access and will first delete all the post done by the user.id and then delete the profile of user.id and then delete the user using the user.id
3. Then it dispatches an action of type(CLEAR\_PROFILE) which changes the initial state and put null into the profile. Also dispatches an action of type ACCOUNT\_DELETED which removes token from the localStorage. Else return error.

**Delete education/experience from the profile**

1. In the frontend, in the actions, there is a function with parameter as the edu/exp id(id bcz there can be multiple edu/exp columns) which makes an api call in the backend DELETE /api/profile/edu/{$id} and store op in ‘res’
2. In the backend route, (protected route, auth middleware)first gets the profile using req.user.id and from the profile find the idx that is same of the edu\_id in the params. And remove it from the database. Then save the database. And return the profile and in case of error return them.
3. The dispatches an action with type: UPDATE\_PROFILE with payload equals to res.data.
4. In the reducers, in the intial state set profile = payload(updated profile).

**Add education/experience in the profile.**

**Get current users profile**

1. From the FE, in the actions there is a func, that makes an api call to the backend GET /api/profile/me and save the op in ‘res’
2. In the backend, protected route, finds the profile in the db using req.user.id also populate name, avatar and the user; if doesn’t exists then return error else return the profile.
3. Then dispatch an action of type:GET\_PROFILE and payload: res.data
4. In the reducers, initial state remains the same, but payload is put into the profile.

**Get all the post**

1. In the FE, in the actions, there is a function, it first makes a request to the backend /api/posts and stores op in ‘res’
2. In the backend, it gets all the posts, sorted in descending order wrt to the date and then return all the posts.
3. Then it dispatches an action, type: GET\_POSTS, payload: res.data
4. In the reducers, return with initial state as it is with posts: payload

**Get a post by ID**

1. There is a fnc that has post\_id as inp, it first makes an api call to the bckned GET /api/post/{$id} and store data in the op in ‘res’
2. In the bcknd, it searches for the post in the database using the req.params, if post not present then return error. Else return the post.
3. Dispatch an action with type: GET\_POST and payload: res.data
4. In the reducers, return the initial state as it is but put posts: A

**Add a post**

1. In the func, takes formdata as inp then make a api post req to the bcknd with the formData and save the op in ‘res’
2. In bcknd, protected route, check whether text is empty or not using express validator. Find the user from DB using req.user.id and make a new post obj and then save the new post into the db and return the post.
3. Dispatches an action of type: add\_post with payload : res.data and which then we add payload post into the posts and return it.

**Delete a Post**

1. In the frontend, actions, there is a function with input parameter as id(post id) which first makes an api call to the backend DELETE (/api/posts/${id}).
2. In the backend, first find the post using params.id. if post not present return msg(post not found) then check if the post.user == req.user.id, to check whether the logged in user is authorized to delete the post(logged in user is the use that made the post) if not then return msg (User not authorized). Then remove the post from db. Else return error.
3. Then dispatch an action of type DELETE\_POST with payload as postid.
4. In the reducers, of type DELETE\_POST, return state as it is but filter through the state.posts and not take the postid. ( posts: state.posts.filter((post) => post.\_id !== payload))

**Add a Like**

1. In the frontend, actions, there is a function with the postid as input parameter. First make a api call to the backend PUT(/api/posts/like/${id}) store the output in the res.
2. In the backend, first it finds the post with the id = post.id from the db and store it into post. Check If the post has already been liked (post.likes.filter((like) => like.user.toString() === req.user.id).length > 0 then return msg that post has already been liked. Then add the req.user.id in the likes and save the db return post.likes.
3. Then Dispatch an action with type UPDATE\_LIKE with payload as {postid, likes = res.data}.
4. In the UPDATED\_LIKES reducers, return the state as it is with posts: state.posts.map((post) => post.\_id === payload.id ? {…post, likes.payload.likes}:post). First find the post with the postId and update its likes with payload.likes.

**Remove Like**

1. In the frontend actions, there is function with postid as input parameter. First make an api call to the backend PUT(/api/posts/unlike/${id}) and store the output in the res.
2. In the backend, first find the post with id = params.postid from the db and store it in post. Check if the post has not been liked. (!post.likes.filter((like) => like.user.toString() === req.user.id)) If yes then return msg post has not been like yet. From the post.like, remove the user with user.id === req.user.id (post.likes = post.likes.filter(({user}) => user.toString() !== req.user.id)) then save the db and return post.likes.
3. Then Dispatch an action with type UPDATE\_LIKE with payload as {postid, likes = res.data}.
4. In the UPDATED\_LIKES reducers, return the state as it is with posts: state.posts.map((post) => post.\_id === payload.id ? {…post, likes.payload.likes}:post). First find the post with the postId and update its likes with payload.likes.

**Add a comment**

1. In the frontend actions, the function takes (postid, formdata) as input then make an api call to the backend POST(/api/posts/comment/${postid}, formdata, config) and store the output in res.
2. In the backend, first find the user with the Id = req.user.id from the db and store it in user. Find the post with id = postid from the db and store it in post. If post not present return post not found. Make a new comment obj with text = req.body.text, user = user.id, name = user.name. and then add the comment in the post.comments and save the db. Return post.comments.
3. Then dispatch an action with type ADD\_COMMENT with payload = res.data
4. In the reducer of type ADD\_COMMENT, in the post, keep the state post as it is and in the post, make payload = comments. (post: {…state.post, comments: payload})

**Delete a Comment**

1. In the frontend actions, the function has a (postId, commentId) as the input parameter. Make an api call to the backend DELETE(/api/posts/comment/${postid}/${commentid})
2. In the backend, find the post from the db with postid and also find the comment from the db with commented. If comment does exist return comment not found. Check if user is authorized to delete a comment. comment.user.toString() !== req.user.id. find the index of the comment and then remove it from post.comments and then save the db and return post.comments.
3. Then dispatch an action with type REMOVE\_COMMENT with payload as commented.
4. In the reducer, return state as it is, and the posts in the state expect the comment that has the same id as payload.