Contact Keeper – Documentation

57. Edit & Update Contact Action

- When edit button is clicked, the contact is set it the current state which is initially set as null in ‘ContactStatejs’ > ContactState variable.

- We want to destructure addContact and current in ‘ContactFormjs’ by adding ‘const { addContact, current } = contactContext’. We can therefore, remove contactContext from onSubmit method.

- We want to fill form based on anything in the current value.

- Pass in useEffect hook into import React.

- Reacts useEffect hook combines reactClasses componentDidMount, componentDidUpdate and componentWillUnmount lifecycle methods. This is useful as it reduces the amount of code, simplifies the code an allows for multiple useEffect hooks to be called in a single component.

- In the useEffect hook, using arrow functions, pass in ifElse statement.

- If current is not equal to null, setContact to contact of which contact is being edited.

- Else, set contact to its default state.

setContact({

name: "",

email: "",

phone: "",

type: "personal",

});

- Pass in dependencies, as we only want this to happen if the contactContext is changed or the current has changed.

useEffect(() *=>* {

if(current !== null) {

setContact(current);

} else {

setContact({

name: "",

email: "",

phone: "",

type: "personal",

});

}

// Dependencies

}. [contactContext, current]);

- Update contactForm Header and Button

- Using ternary operator, if current editContact else addContact

<h2 className="text-primary">{current ? 'Edit Contact' : 'Add Contact'}</h2>

<div>

<input

type="submit"

value={current ? 'Edit Contact' : 'Add Contact'}

className="btn btn-primary btn-block"

/>

</div>

- Add clear button

- Add ternary operator with current and change button to btn-light and pass in function clearAll

{current && <div>

<button className="btn btn-light btn-block" onClick={clearAll}>Clear</button>

</div>}

- Create clear all button function

- Pass in clearCurrent to state and pass it into the function

*const* clearAll = () *=>* {

clearCurrent();

}

- Create conditional in onSubmit depending on whether we are updating or adding contact

- Create conditional statement

- If current is equal to null, addContact else updateContact (create updateContact functionality in contactState)

- In updateContact pass in contact and the type to UPDATE\_CONTACT

// Update Contact

*const* updateCurrent = (*contact*) *=>* {

dispatch({ type: UPDATE\_CONTACT, payload: *contact* });

};

- In contactReducer, set the case how it will be passed down in the state

- Using .map() method to map through contacts. for each contact, take contact.id and if equal to the action.payload.id (which is the whole contact) if that;s the case, then return the updated contact, else then just return the contact as it is.

case UPDATE\_CONTACT:

return {

...*state*,

contacts: *state*.contacts.map((*contact*) *=>*

*contact*.id === *action*.payload.id ? *action*.payload : *contact*

),

};

- Then add to contactContext.Provider in the contactState

<*contactContext.Provider*

value={{

contacts: state.contacts,

current: state.current,

addContact,

deleteContact,

setCurrent,

clearCurrent,

updateContact

}}

>

{*props*.children}

</*contactContext.Provider*>

- Pass in updateContact into contactContext state in ‘contactFormjs’. Then pass into else in onSubmit function

// onSubmit

*const* onSubmit = (*e*) *=>* {

*e*.preventDefault();

if (current === null) {

addContact(contact);

} else {

updateContact(contact);

}

setContact({

name: "",

email: "",

phone: "",

type: "personal",

});

};

58. Contact filter form & state

- Create state dedicated to filtered contacts

- In ContactState create a filtered state and set it to null

filtered: null

- Create filterContacts function in ContactState which takes in text as a parameter, FILTER\_CONTACTS as type and text as payload.

// Filter Contacts

*const* filterContacts = (*text*) *=>* {

dispatch({ type: FILTER\_CONTACTS, payload: *text* });

};

- Create clearFilter function

- Set dispatch back to null

- Handle dispatches in reducer in ‘contactReducerjs’

Take the filtered part of the state which by default is null in ContactsState and set to state.Contacts and use the filter method

- forEach contact, create a regular expression and pass in action.payload as well as ‘gi’ parameter which stands for ‘global and insensitive’ as we want to match whether is uppercase or lowercase.

- Return anything where the name matches the text we pass in or the email we pass in.

- Add CLEAR\_FILTER case and set filter to null.

case FILTER\_CONTACTS:

return {

...*state*,

filtered: *state*.contacts.filter(*contact* *=>* {

*const* regex = new *RegExp*(`${*action*.payload}`, 'gi');

return *contact*.name.match(regex) || *contact*.email.match(regex);

})

};

case CLEAR\_FILTER:

return {

...*state*,

filtered: null,

};

- Add to provider to make sure we can access them,

<*contactContext.Provider*

value={{

contacts: state.contacts,

current: state.current,

filtered: state.filtered,

addContact,

deleteContact,

setCurrent,

clearCurrent,

updateContact,

filterContacts,

clearFilter

}}

>

{*props*.children}

</*contactContext.Provider*>

- Create new component in contacts called contactFilterjs

- Import contactContext from contactContext

- Initialise contactContext to useContext so we can use the contacts

- Destructure contactContext

- React’s useContext uses the Context object, which has been previously created to retrieve the most recent value from it.

- Pass in useRef hook

- The useRef Hook is a function that returns a mutable [ref](https://reactjs.org/docs/refs-and-the-dom.html) object whose .current property is initialized with the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

- There are two main uses of useRef that are explained in the following sections: [Accessing the DOM nodes or React elements](https://reactjs.org/docs/refs-and-the-dom.html) and [keeping a mutable variable](https://reactjs.org/docs/hooks-faq.html#is-there-something-like-instance-variables).

- Initialise useRef by saying const text = useRef(‘’) and just set to default.

- Pass in <form> element with onChange method passed in.

- Create onChange method passing in event and get the value of the input by putting text.current.value

- Using if statement, if the value is equal to nothing then run filterContacts method from contactContext and then pass in e.target.value which is the actual text.

- else, clearFilter

- Bring in useEffect hook.

- In useEffect, if filter is equal to null text.current.value is equal to nothing.

- Embed into Home.js. and place above contacts

export *const* ContactFilter = () *=>* {

*const* contactContext = useContext(ContactContext);

*const* text = useRef("");

*const* { filterContacts, clearFilter, filtered } = contactContext;

useEffect(() *=>* {

if(filtered === null) {

text.current.value = '';

}

})

*const* onChange = (*e*) *=>* {

if (text.current.value !== "") {

filterContacts(*e*.target.value);

} else {

clearFilter();

}

};

return (

<div>

<form>

<input ref={text} type="text" placeholder="Filter Contacts..." onChange={onChange} />

</form>

</div>­­­­­­­­

);

};

- In contacts.js if there is no contact, please add a contact

if(contacts.length === 0) {

return <h4>Please add a contact</h4>

}

- In contacts.js, pull out filtered from context ­­­­­

- In fragment, if filtered is not equal to null, then map through it, foreach filteredcontact, then load contact item

- If there’s nothing is contacts then we’re going to show the contacts

<*Fragment*>

{filtered !== null ? filtered.map(*contact* *=>* (<*ContactItem* key={*contact*.id} contact={*contact*} /> )) :

contacts.map(*contact* *=>* (<*ContactItem* key={*contact*.id} contact={*contact*} />))}

</*Fragment*>

59. React animation add & delete

- <https://reactcommunity.org/react-transition-group/transition-group>

Backend – React/Express Authentication

60. Authenticate Context & Initial State

- Create Auth folder in context folder

- Create files AuthContextjs, AuthReducerjs and AuthStatejs

- Pass in types to typejs

export *const* REGISTER\_SUCCESS = "REGISTER\_SUCCESS";

export *const* REGISTER\_FAIL = "REGISTER\_FAIL";

export *const* USER\_LOADED = "USER\_LOADED";

export *const* AUTH\_ERROR = "AUTH-ERROR";

export *const* LOGIN\_SUCCESS = "LOGIN\_SUCCESS";

export *const* LOGIN\_FAIL = "LOGIN\_FAIL";

export *const* LOGOUT = "LOGOUT";

export *const* CLEAR\_ERRORS = "CLEAR\_ERRORS";

- In AuthContextjs, import createContext and initialise authContext as createContext(), similar to what is being done in ../../context/contact/contactContext

import { createContext } from "react";

*const* authContext = createContext();

export default authContext;

- In AuthStatejs, paste in everything from ContextStatejs and change the imports and types

- In the initialState set the token to localStorage.getItem() to allow the browser to store local storage

- Set isAuthenticated and initially set to null

- Initially set user to null

- Set loading to true and error to null initially

- Set useReducer to authReducer

*const* AuthState = (*props*) *=>* {

*const* initialState = {

token: localStorage.getItem('token'),

isAuthenticated: null,

user: null,

loading: true,

error: null,

};

- in Return, change to AuthContext.Provider

- Pass in the state values

return (

<*AuthContext.Provider*

value={{

token: state.token,

isAuthenticated: state.isAuthenticated,

user: state.user,

loading: state.loading,

error: state.error,

}}

>

{*props*.children}

</*AuthContext.Provider*>

);

};

- To utilise, import into App.js and put into function App() as the first provider

61. Register & Login Forms

- Create auth folder in components and create register.js and login.js files

- In registerjs import useState

- useState is a Hook(function) that allows you to have state variables in functional components.

- You pass the initial state to this function, and it returns a variable with the current state value and another function to update this value.

- In registerjs import useState

- Destructure variables

export *const* register = () *=>* {

*const* [user, setUser] = useState({

name: "",

email: "",

password: "",

password2: "",

});

*const* { name, email, password, password2 } = user;

- Create header and form in JSX

- Pass in props and onChange to input for each variable

<form>

<div className="form-group">

<label htmlFor="name">Name</label>

<input type="text" name="name" value={name} onChange={onChange}/>

</div>

</form>

<form>

<div className="form-group">

<label htmlFor="name">Email</label>

<input type="text" name="email" value={email} onChange={onChange}/>

</div>

</form>

<form>

<div className="form-group">

<label htmlFor="name">Password</label>

<input type="text" name="name" value={password} onChange={onChange}/>

</div>

</form>

<form>

<div className="form-group">

<label htmlFor="name">Confirm Password</label>

<input type="text" name="password2" value={password2} onChange={onChange}/>

</div>

<input type="value" value="Register" className="btn btn-primary btn-block"/>

</form>

- Create onChange function and setUser to the object user using the spread operator, and set to the name and set to the value using e.target

*const* onChange = *e* *=>* setUser({...user, [*e*.target.name]: *e*.target.value});

- Pass in onSubmit to the form and create the onSubmit function

<form onSubmit={onSubmit}>

- Create onSubmit function and pass in preventDefault().

*const* onSubmit = *e* *=>* {

*e*.preventDefault();

console.log('Register Submit');

}

- Import into App.js and create a react.router for the navbar

62. Alert Context & State Component

- Create AlertContextjs, AlertReducerjs and AlertStatejs

- Copy AuthState into AlertState and change imports and state

- Import SET\_ALERT and REMOVE\_ALERT from types

- Import uuid

- In actions, create setAlert Action

- The setAlert action will take in 2 parameters, both message and type and the unique id from uuid

- In dispatch pass in the type of SET\_ALERT

- Pass the uuid into an id variable to use as a parameter, as well as message and type into the payload

- Payload is what is used as actions and passed around between reducers in the redux application.

- Actions must have a type field that indicates the type of action being performed

- Every payload has a type, in this case the type passed in from the type.js

// Set Alert Action

*const* setAlert = (*msg*, *type*) *=>* {

*const* id = uuid.v4();

dispatch({

type: SET\_ALERT,

payload: { msg, type, id },

});

};

- Create a setTimeout so the alert disappears over a certain amount of time

- In dispatch pass in the type of REMOVE\_ALERT and the payload if going to be the id of which one we are going to remove as well as 4000 for the time it will take to disappear

- In AlertContext.Provider pass in the context of alert:state which passes in the whole alert array as well as the setAlert action

<*AlertContext.Provider*

value={{

alert: state,

setAlert,

}}

>

{*props*.children}

</*AlertContext.Provider*>

- Import into app.js

*function* App() {

return (

<*AuthState*>

<*ContactState*>

<*AlertState*>

<*Router*>

<*Fragment*>

<*Navbar* />

<div className="container">

<*Switch*>

<*Route* exact path="/" component={Home} />

<*Route* exact path="/about" component={About} />

<*Route* exact path="/register" component={Register} />

<*Route* exact path="/login" component={Login} />

</*Switch*>

</div>

</*Fragment*>

</*Router*>

</*AlertState*>

</*ContactState*>

</*AuthState*>

);

}

- In AlertReducer create payload actions using switch statement for SET\_ALERT and REMOVE\_ALERT

- For SET\_ALERT return any state that is inside the array and the action.payload which is the alert that gets sent, in this case the SET\_ALERT created in AlertState

- For case REMOVE\_ALERT, filter out the correct alert which will be by ID. For each alert, check the alert.id and if it is not equal to action.payload, which is the ID.

import { SET\_ALERT, REMOVE\_ALERT} from '../types';

export default (*state*, *action*) *=>* {

switch (*action*.type) {

case SET\_ALERT:

return [

...*state*, *action*.payload];

case REMOVE\_ALERT:

return *state*.filter(*alert* *=>* *alert*.id !== *action*.payload)

default:

return *state*;

}

};

- Create Alert component in layout to display the Alerts called Alerts.js

- In return, if alert.length is greater than zero, so if there is an alert then map through the alerts.

- It will check the alerts in the state and if there is anything there, then it will loop through them and output the correct JSX

import React, { useContext } from "react";

export *const* Alerts = () *=>* {

*const* alertContext = useContext(AlertContext);

return (

alertContext.alerts.length > 0 &&

alertContext.alerts.map((*alert*) *=>* (

<div key={*alert*.id} className={`alert alert-${*alert*.type}`}>

<i className="fas fa-info-circle" /> {*alert*.msg}{" "}

</div>

))

);

};

export default Alerts;

- Import into App.js

- Create onSubmit method in register.js

- If name, email, password equal nothing setAlert function

*const* onSubmit = (*e*) *=>* {

*e*.preventDefault();

if(name === '' || email === "" || password === '') {

setAlert('Please enter all fields', 'danger');

} else if (password !== password2) {

setAlert('Passwords do not match', 'danger');

} else {

console.log('Register Submit');

}

};

- Add required to each input e.g.

<input type="text" name="name" value={name} onChange={onChange} required />

63. User Registration

- Import axios into AuthState.js. Axios is used to make the backend data request

- Under register user action, create a register variable. This will be asynchronous and take in formData, the data required to register the user

- Because we are making a post request, and we are sending data to the backend, we require a type header of application/json

- Create a config object within that create a headers object with a content-type of application/json

- As a request is being made, we need a try/catch

- In try create a response variable and we want to await on the axios post request which will take in a promise, which will take in a url of api/users, formData and config

- If everything goes ok, we want to dispatch to our reducer where the type will be REGISTER\_SUCCESS and payload will be the res.data which is the token

- In if there is an error, in catch, dispatch a type of REGISTER\_FAIL and in the payload, pass in err.response.data.msg, which is taken from the JSON error message in users.js

// Register User

*const* register = async (*formData*) *=>* {

*const* config = {

headers: {

"Content-Type": "application/json",

},

};

try {

*const* res = await axios.post("/api/users", *formData*, config);

dispatch({

type: REGISTER\_SUCCESS,

payload: res.data,

});

} catch (err) {

dispatch({

type: REGISTER\_FAIL,

payload: err.response.data.msg,

});

}

};

- Next we have to handle these request in our reducer, AuthReducer.js

- Import types into AuthReducer

- Export default and pass in state and action like we have done in previous reducers

- Set default to return state

- Create switch to action.type and in the REGISTER\_SUCCESS case and set the localStorage to setItem which takes in ‘token’ and action.payload.token which is the object.

- Return the …state, …action.payload, set isAuthenticated to true which is set to null in the AuthState and loading to false, which is initially set to true in AuthState

- In the next case, REGISTER\_FAIL, create the localStorage to removItem(‘token’) and reset everything. Token: null, isAuthenticated: false, loading: false, user: null and the error to action.payload which will give the payload message from the registerUser action in AuthState which is the JSON msg from the files in the route folder.

- It is required to call the Register function which will be required to do from the register component.

- Import Auth contacts into register.js

import {

REGISTER\_SUCCESS, REGISTER\_FAIL, USER\_LOADED, AUTH\_ERROR, LOGIN\_SUCCESS, LOGIN\_FAIL, LOGOUT, CLEAR\_ERRORS

} from '../types';

export default (*state*, *action*) *=>* {

switch (*action*.type) {

case REGISTER\_SUCCESS:

localStorage.setItem('token', *action*.payload.token);

return {

...*state*,

...*action*.payload,

isAuthenticated: true,

loading: false

};

case REGISTER\_FAIL:

localStorage.removeItem('token');

return {

...*state*,

token: null,

isAuthenticated: false,

loading: false,

user: null,

error: *action*.payload

}

}

}

- Import AuthContext into register.js

export *const* Register = () *=>* {

*const* alertContext = useContext(AlertContext);

*const* { setAlert } = AlertContext;

*const* { register } = AuthContext;

*const* [user, setUser] = useState({

name: "",

email: "",

password: "",

password2: "",

});

- Pass in register into onSubmit and pass in object which includes name, email, password

*const* onSubmit = (*e*) *=>* {

*e*.preventDefault();

if (name === "" || email === "" || password === "") {

setAlert("Please enter all fields", "danger");

} else if (password !== password2) {

setAlert("Passwords do not match", "danger");

} else {

register({

name,

email,

password,

});

}

};

- To use register, to go AuthState and make sure it is added to the value

- Add a user in browser and in the console, in Context.Provider, the property isAuthenticated is set to true and a JSON token will be provided

A screenshot of a social media post

Description automatically generated

- In MongoDB cloud, it will have added a user to the database

- If the same user email trys to register again, the error property will show as “User already exists”

A screenshot of a cell phone

Description automatically generated

- This is not visible in the UI, therefore the user would have no idea that user already exists.

- So, if we go back into the register component and pull out error from AuthContext state

*const* { register, error } = authContext;

- So, if we go back into the register component and pull out error from AuthContext state

- We want to run this in the useEffect hook and check the error

- If the error is equal to the actual message “User already exists”, setAlert with the type of danger

- We want to run this when the error is added to the state, so pass in error as a dependency to useEffect

- To clear the errors, we want to dispatch to the AuthReducer from AuthState with the type of CLEAR\_ERRORS

// Clear Errors

*const* clearErrors = () *=>* {

dispatch({type: CLEAR\_ERRORS})

}

- Then create the CLEAR\_ERRORS state in AuthReducer which will return whatever is in the state and then change the error back to null

case CLEAR\_ERRORS:

return {

...*state*,

error: null

}

- Pull from authContext in register.js and call it in useEffect

- This will now show the error if we try to register the same user in the browser

A screenshot of a cell phone

Description automatically generated

64. Load User & Set Token

- In AuthState, make the Load User action async as we are making a request to the backend

- Try/Catch

- Const res = await axios.get(‘/api/auth’), check token to see if it is a valid user

- If that comes back OK, then set dispatch to USER\_LOADED and the payload is going to be res.data.

- In catch, if something goes wrong, then dispatch AUTH\_ERROR

*const* loadUser = async () *=>* {

try {

*const* res = await axios.get('/api/auth');

dispatch({

type: USER\_LOADED,

payload: res.data

})

} catch (err) {

dispatch({

type: AUTH\_ERROR,

})

}

}

- Next go to Authreducer to handle both the USER\_LOADED and AUTH\_ERROR

- Make the USER\_LOADED case to return the …state, isAuthenticated to true, loading to false and the user to action.payload

- For AUTH\_ERROR, make it the same case as REGISTER\_SUCCESS

export default (*state*, *action*) *=>* {

switch (*action*.type) {

case USER\_LOADED:

return {

...*state*,

isAuthenticated: true,

loading: false,

user: *action*.payload

};

case REGISTER\_SUCCESS:

localStorage.setItem('token', *action*.payload.token);

return {

...*state*,

...*action*.payload,

isAuthenticated: true,

loading: false

};

case REGISTER\_FAIL:

case AUTH\_ERROR:

localStorage.removeItem('token');

return {

...*state*,

token: null,

isAuthenticated: false,

loading: false,

user: null,

error: *action*.payload

};

case CLEAR\_ERRORS:

return {

...*state*,

error: null

};

default: return *state*;

}

}

- In order to make a request to api/auth, we require a token as it is a private route and we want to store a header within mongodb atlas, with that token, which we want to call in the load user in AuthState.js

- To do this, create a separate folder and file

- In Src, create a folder called ‘utils’ and create a file called setAuthToken.js

- In setAuthToken.js, import axios

- Create a setAuthToken variable and pass in token as the parameter.

- If token then call axios.default.headers.common[x-auth-token] the key token and set to whatever the token is which is what we passed in the mongoDB atlas, else we want to delete it.

- Export as default.

import axios from "axios";

*const* setAuthToken = (*token*) *=>* {

if (*token*) {

axios.defaults.headers.common["x-auth-token"] = *token*;

} else {

delete axios.defaults.header.common["x-auth-token"];

}

};

export default setAuthToken;

- Import into authState

- In authState inside load user, check localStorage.token, setAuthToken to localStorage.token

- We also want to call it in the main App.js as we want this to load every time the main component loads

A screenshot of a cell phone

Description automatically generated

- Go back into Authstate and we want to call the loadUser() function within the Register User action underneath dispatch

try {

*const* res = await axios.post("/api/users", *formData*, config);

dispatch({

type: REGISTER\_SUCCESS,

payload: res.data,

});

loadUser();

- Once we register the user, we should get logged in, it should get the token, load the user and isAuthenticated should be set to true

- Go back into register.js component, because we want it to redirect if it authenticated.

- Bring in isAuthenticated into the state from authContext and bring into useEffect.

- If it isAuthenticated we want to redirect and in React to redirect we use props.history.push which will be redirected to the home page using (‘/’).

- Since we now have more things we are depending on in useEffect, we have to pass the dependencies such as isAuthenticated and props.history to prevent any error messages

- Make sure to pass in props to the Register function

export *const* Register = (*props*) *=>* {

*const* alertContext = useContext(AlertContext);

*const* authContext = useContext(AuthContext);

*const* { setAlert } = alertContext;

*const* { register, error, clearErrors, isAuthenticated } = authContext;

useEffect(() *=>* {

if(isAuthenticated) {

*props*.history.push('/');

}

if(error === 'User already exists') {

setAlert(error, 'danger');

clearErrors();

}

}, [error, isAuthenticated, *props*.history]);

- To remove the error

[1] ./src/components/auth/Register.js

[1] Line 21:4: React Hook useEffect has missing dependencies: 'clearErrors' and 'setAlert'. Either include them or remove the dependency array react-hooks/exhaustive-deps

- Make sure to pass in // eslint-disable-next-line to remove the errors

- Now, once we register a user, it should call loadUser() and we should get the user from the back-end

- Now go to the browser and in ContextProvider make sure there is no token within the localStorage

A screenshot of a cell phone

Description automatically generated

- Once we register a new user, the user should get put inside the user property value.

A screenshot of a social media post

Description automatically generated

- Once the user has registered, it will redirect the user to its contact page, because isAuthenticated has been set to true

- Now, when the page is reloaded, the user in the localStorage is going to be removed, the user Token will still be there however, user will be set to null and inAuthenticated will be set to null because loadUser() has not been called in the Home.js component

A screenshot of a social media post

Description automatically generated

- In Home.js, import useContext hook into react and initialise within the home function constructor using the useContext hook

- Import the useEffect hook

- We want to call the authContext variable as well as the loadUser function within the useEffect hook because we want it to run as soon as the component loads.

- This will look at the token and hit the backend and validate the user within the state as soon as the component loads.

- Because we only want this to happen when the component loads, we have to add a dependency at the bottom of just an empty array bracket [].

- To prevent any error messages, pass in the eslint-disable-next-line comment

- When we reload the home dashboard page, the use stays authenticated.

A screenshot of a cell phone

Description automatically generated

65. User Login

- Go to AuthStae.js where we have our login function and copy the register function into login

- Login is going to require form data, so keep the headers

- However, the request data is going to be sent to api/auth

- Change try dispatch to LOGIN\_SUCCESS with the payload to res.data

- loadUser() function remains where it is

- If there is an error in catch, then dispatch LOGIN\_FAIL and keep the payload msg in the state

// Login User

*const* login = async (*formData*) *=>* {

*const* config = {

headers: {

"Content-Type": "application/json",

},

};

try {

*const* res = await axios.post("/api/auth", *formData*, config);

dispatch({

type: LOGIN\_SUCCESS,

payload: res.data,

});

loadUser();

} catch (err) {

dispatch({

type: LOGIN\_FAIL,

payload: err.response.data.msg,

});

}

};

- Next, go into authReducer, the LOGIN\_SUCCESS with have the same effect as REGISTER\_SUCCESS because login and register both return a token, so we are going to take the tokens put it in local storage, add everything to the the state and login

- For LOGIN\_FAIL, place under REGISTER\_FAIL and AUTH\_ERROR, which will remove the locaStorage and set the state back to null.

- Now, go into the login component and copy a few things from register.js import authContext and alertContext and pass in the useContext and useEffect hooks.

- Initialise, alertContext and authContext in variables however, pass in login as a prop instead of register to authContext

- In useEffect, change the error message to “Invalid Credentials” and make sure it is the same text as is in routes/auth.js

- In the onSubmit function, if email or password is equal to nothing then setAlert with the message “please fill in all fields”, with the color of danger, else call login with the email and password

- Pass in required to the input tag elements

- Now once we login we will like register, be redirected because we will have been authenticated through the backend.

import React, { useState, useContext, useEffect } from "react";

import AlertContext from "../../context/alert/alertContext";

import AuthContext from "../../context/auth/AuthContext";

export *const* Login = (*props*) *=>* {

*const* alertContext = useContext(AlertContext);

*const* authContext = useContext(AuthContext);

*const* { setAlert } = alertContext;

*const* { loginUser, error, clearErrors, isAuthenticated } = authContext;

useEffect(() *=>* {

if (isAuthenticated) {

*props*.history.push("/");

}

if (error === "Invalid Credentials") {

setAlert(error, "danger");

clearErrors();

}

// eslint-disable-next-line

}, [error, isAuthenticated, *props*.history]);

*const* [user, setUser] = useState({

email: "",

password: "",

});

*const* { email, password } = user;

*const* onChange = (*e*) *=>* setUser({ ...user, [*e*.target.name]: *e*.target.value });

*const* onSubmit = (*e*) *=>* {

*e*.preventDefault();

if (email === "" || password === "") {

setAlert('Please fill in all fields', 'danger')

} else {

loginUser({

email, password

});

}

};

66. Logout & Navbar

- In navbar.js component, we need to import AuthContext.js and initialise in the navbar constructor

- Pull out isAuthenticated, logout, user in authContext.

import React from "react";

import PropTypes from "prop-types";

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

import AuthContext from "../../context/auth/AuthContext";

export *const* Navbar = ({ *title*, *icon* }) *=>* {

*const* authContext = useContext(AuthContext);

*const* {isAuthenticated, logout, user} = authContext;

- We need to divide these up into links dependending on whether the user is logged in or not

- Using fragment and JSX for the list items for the auth links.

- Use a list item for the user with hello to check if there is a user and if there is put the user.name

- Add another li element for the logout text surrounded by the span element with class class hide-sm which is on the core css and will hide on small screens, which will contain an anchor tag, as well with the log-out font-awesome icon.

*const* authLinks = (

<>

<li>Hello {user && user.name}</li>

<li>

<a href="!#">

<i className="fas fa-sign-out-alt"></i>

<span className="hide-sm">Logout</span>

</a>

</li>

</>

);

- For guestLinks, we only need the Register and Login in the navbar

*const* guestLinks = (

<>

<li>

<*Link* to="/Register">Register</*Link*>

</li>

<li>

<*Link* to="/Login">Login</*Link*>

</li>

</>

);

- We can remove the navbar home link, as the only page the user requires is their contact dashboard, and this will only be available to them when they log in.

- Instead, we can include isAuthenticated, if the user is, then show the authLinks, if not then show the guestLinks

return (

<div className="navbar bg-primary">

<h1>

<*Link* to="/">

<i className={*icon*} /> {*title*}

</*Link*>

</h1>

<ul>

{isAuthenticated ? authLinks : guestLinks}

<li>

<*Link* to="/about">About</*Link*>

</li>

</ul>

</div>

);