Code Guide for projects in APPWRITE and REACT

Project Setup & Installing additional dependencies

Npm create vite@latest

Npm i

Install=>

1. npm i react-redux
2. npm i react-router-dom
3. Appwrite
4. TinyMCE (rich text editor)
5. Html-react-parser(use to parse html)
6. React hook form (to handle input forms)

SIGNUP ON APPWRITE

1. Create account on appwrite
2. Take project id and api endpoint from there
3. And then other imp variable ids
4. Then setup them in .env file
5. And add .env file in .gitignore before pushing to github

Setting up environment variables

VITE\_APPWRITE\_URL="" //create project-> take api endpoint from settings

VITE\_APPWRITE\_PROJECT\_ID="" //create project-> take project id from settings

VITE\_APPWRITE\_DATABASE\_ID="" //create database-> take database id

VITE\_APPWRITE\_COLLECTION\_ID="" //create collection in database-> take collection id

// change permissions, create attributes, create indexes

VITE\_APPWRITE\_BUCKET\_ID="" // storage->bucket names as images->take id

// change permissions

Accessing environment variables

1. If someone has created react app using create-react-app, then it is mandatory,

* To set environment variables, use REACT\_ APP\_Keyword.

Eg-> APPWRITE\_URL (this is wrong)

REACT\_APP\_APPWRITE\_URL (this is right)

* To access them, we have to use *process.env.(variable name)*

Eg*-> APP.JS =>*

*console.log(process.env.REACT\_APP\_APPWRITE\_URL)*

(bcz environment variable file is defined in process)

1. If someone has created react app using VITE ,then it is mandatory,

* To set environment variables, use VITE\_ Keyword.

Eg-> APPWRITE\_URL (this is wrong)

VITE\_APPWRITE\_URL (this is right)

* To access them, we have to use *import.meta.env.(variable name)*

Eg*-> APP.JS =>*

*console.log(import.meta.env.VITE\_APPWRITE\_URL)*

Implementing better way to access environment variables (for production grade apps)

1. Create conf folder in src
2. Create conf.js in this folder
3. Create a conf named object in this file and export it.
4. We are doing these, bcz using *import.meta.env.variable* can cause error, bcz maybe it won’t load and this will lead to app crash.

VITE\_APPWRITE\_PROJECT\_ID="664f203100339a8b1c0a"

And, in above example if there are no alphabets , then maybe it can be treated as a number which will lead to error

Because environment variables are always in string, so we will export key value pair in conf.js

***const conf = {***

***appwriteUrl: String(import.meta.env.VITE\_APPWRITE\_URL),***

***appwriteProjectId: String(import.meta.env.VITE\_APPWRITE\_PROJECT\_ID),***

***appwriteDatabaseId: String(import.meta.env.VITE\_APPWRITE\_DATABASE\_ID),***

***appwriteCollectionId: String(import.meta.env.VITE\_APPWRITE\_COLLECTION\_ID),***

***appwriteBucketId: String(import.meta.env.VITE\_APPWRITE\_BUCKET\_ID)***

***}***

***export default conf;***

5. Here, use of string will make sure that each variable will be of type string

Vendor Lock-In

Vendor lock-in refers to a situation where the cost of switching to a different vendor is so high that the customer is essentially stuck with the original vendor.

To solve vendor lock-in for full-stack applications, you can create services in your code using the following strategies:

1. **Microservices Architecture**: Break your application into smaller, independent services. This allows you to change or replace parts without affecting the whole system.
2. **API-Driven Development**: Use standardized APIs for clear boundaries and interoperability, making it easier to switch out components.
3. **Containerization**: Package your application into containers (e.g., using Docker) to ensure it runs consistently across different environments and cloud providers.
4. **Service Mesh**: Use a service mesh (e.g., Istio) to manage communication between microservices, enhancing control and vendor independence.
5. **Infrastructure as Code (IaC)**: Manage infrastructure with code (e.g., using Terraform) for repeatable and consistent deployments across various vendors.

These practices promote modularity, flexibility, and portability, reducing

dependency on any single vendor.

Example Scenario

Imagine you have a full-stack application with a frontend, backend, and a database. By creating services in the following ways, you can mitigate vendor lock-in:

* **Frontend:** The frontend communicates with the backend through a set of well-defined APIs. These APIs remain consistent regardless of backend changes.
* **Backend:** The backend is composed of several microservices, each responsible for a specific function (e.g., user authentication, data processing). These microservices are containerized, allowing them to run on any container orchestration platform (e.g., Kubernetes).
* **Database:** Use a database abstraction layer or an ORM (Object-Relational Mapping) tool to abstract the database interactions, making it easier to switch database providers if needed.

In JavaScript full-stack applications, addressing vendor lock-in through services doesn't necessarily require them to be written as classes. Services can be any part of the code that handles specific tasks, such as accessing a database or calling an external API. You can write these services using classes, but you can also use simple functions or modules.

The key is to create an abstraction layer—that means to build a layer in your code that separates the core functions of your application from the specific technologies it uses (like a certain type of database or cloud service). This means writing your code so that it doesn't depend directly on one particular technology. When you do this, you can switch to a different technology or service provider more easily without needing to overhaul the main parts of your application. By creating this separation, either using classes or functions, you help ensure that changing vendors or technologies is straightforward, reducing your reliance on any one supplier.

EXAMPLE:

// auth.js

**class AuthService** {

constructor(authProvider) {

this.authProvider = authProvider;

}

**login(username, password)** {

return this.authProvider.login(username, password);

}

**logout()** {

return this.authProvider.logout();

}

**// Other authentication-related methods**

}

module.exports = AuthService;

NOTE:

In this example, **AuthService** is a class that handles authentication. It uses an **authProvider**, which could be any authentication service. If you decide to switch from one authentication service to another, you only need to change the **authProvider** implementation, not the rest of your application.

This approach helps in tackling vendor lock-in by making it easy to replace or update specific services without rewriting your entire application.

Build authentication service with appwrite

1. Create a folder for services , here we are using appwrite, so we will name that folder appwrite.
2. Make a file named auth.js in it for authentication services.
3. Now, import client, account and ID from appwrite.
4. Create a class named AuthService and export it, then export it.
5. To use this class , we have to create object and then call methods on that object everytime.

So, we are creating an object already and will use it directly and then we will call methods on that object in need everytime.

1. Create an object and export it.

const authService = new AuthService();

export default authService;

Now, if anyone imports it , then he can access all the methods that are created in that object. Eg: authService.signup, etc.

1. We have to create a client and account, because all the functions are called on account.

*export class AuthService(){*

*client = new Client();*

*account;*

*}*

*const authService = new AuthService();*

*export default authService*

* Here, we are not creating account using new keyword like client.

Because it doesn’t make sense as it will be a wastage of resource

Whenever someone creates an object, then account should be created.

And the method, that is automatically called on creation of object is constructor.

And in this constructor, we will create account( by setting endpoint and projectid on client)

*export class AuthService(){*

*client = new Client();*

*account;*

*constructor(){*

*this.client*

*.setEndpoint(conf.appwriteUrl)*

*.setProject(conf.appwrite);*

*this.account = new Account(this.client);*

*}*

*}*

*const authService = new AuthService();*

*export default authService*

1. Now we will list all the service(methods), like createAccount, login, logout

***async createAccount({email, password, name})****{*

*//use try catch for avoiding errors and read docs of appwrite to avoid mistakes*

*try{*

*const userAccount = await this.account.create(ID.unique(), email, password, name);*

*if(userAccount){*

*//call another method like direct login or any msg (account created successfully)*

*return this.login({email,password});*

*}else{*

*return userAccount;*

*}*

*}*

*catch(error){*

*throw error;*

*}}*

***async login({email, password})****{*

*//use try catch for avoiding errors and read docs of appwrite to avoid mistakes*

*try{*

*return await this.account.createEmailSession(email, password);*

*}*

*catch(error){*

*throw error;*

*}*

*}*

***async getCurrentUser()*** *{*

*try {*

*return await this.account.get();*

*} catch (error) {*

*console.log("Appwrite service :: getCurrentUser :: error", error);*

*}*

*return null; //in case there is no account*

*}*

***async logout()*** *{*

*try {*

*await this.account.deleteSessions();*

*} catch (error) {*

*console.log("Appwrite service :: logout :: error", error);*

*}*

*}*

Appwrite database, file upload and custom queries

1. Make a file in appwrite folder named config.js
2. This file will contain database services.
3. And, file upload services , custom queries too.
4. Then make methods like createPost, updatePost, deletePost, getPost, getPosts for database services.

5. createPost=> add parameters in this method as an object,

That will use destructuring . {title, slug, content,

featuredImage, status, userId}

* Use slug as document id
* And read docs of appwrite

6. updatePost (slug,{title,content,featuredImage,status}) => add

parameters in this method as an object,

That will use destructuring .(slug, {title, content,

featuredImage, status, userId})

* Use slug as document id
* And read docs of appwrite

7. deletePost(slug)=> add parameters in this method => slug

* Use slug as document id
* And read docs of appwrite

8. getPost(slug)=> use slug as document id and parameter to get

The required document.

9. getPosts(queries = [Query.equal("status","active")]) =>

Add queries in parameter

Or we can also directly add queries in method

Without assigning it to a keyword like queries

async getPosts(queries = [Query.equal("status","active")]){

try{

return await this.databases.listDocuments(

conf.appwriteDatabaseId,

conf.appwriteCollectionId,

queries

)

}

catch(error){

console.log("Appwrite serive :: getPosts :: error", error);

}

}

Or

async getPosts(){

try{

return await this.databases.listDocuments(

conf.appwriteDatabaseId,

conf.appwriteCollectionId,

[Query.equal("status","active")]

)

}

catch(error){

console.log("Appwrite serive :: getPosts :: error", error);

}

}

### Now for File Upload Services =>

1. Make methods for file upload services now.
2. uploadFile, deleteFile, getFilePreview , etc.
3. uploadFile(file) => use ID.unique() as file id
4. deleteFile(fileId)
5. getFilePreview(fileId)

  async uploadFile(file){

            try{

              return  await this.bucket.createFile(

                    conf.appwriteBucketId,

                    ID.unique(),

                    file

                )

            }

            catch(error){

                console.log("Appwrite serive :: uploadFile :: error", error);

                return false

            }

        async deleteFile(fileId){

            try{

               await this.bucket.deleteFile(

                conf.appwriteBucketId,

                fileId

               )

               return true;

            }

            catch(error){

                console.log("Appwrite serive :: deleteFile :: error", error);

                return false

            }  }

        getFilePreview(fileId){

          return this.bucket.getFilePreview(

            conf.appwriteBucketId,

            fileId

          )

        }

**We will use redux, and use its store to know whether the user is logged in or not**

How to configure redux toolkit in big projects

1. **Setting up a store using Redux toolkit in a big project**

We can also put store and slices in 1 folder, and here we will put slices and store in 1 folder named store.

1. **Creating and configuring redux store and authentication slice**

* Configure store and import main reduce from authSlice.js
* We can import the main reducer that we exported in slice with any name in store.So, we are importing it with name authReducer.
* **CODE =>**

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

import authReducer from "./authSlice";

const store = configureStore({

reducer:{

auth: authReducer

} })

1. **Now, we will create authentication slice named authSlice.js**

* This silce file will contain reducers and actions.
* In initialState , we will give two states, status and userData

as these 2 can help in tracking whether the user is logged in or not.

* initialState:{

state:false,

userData:null

}

* Then, create slice and add name, initalState, reducers in it.We will make 2 reducers here, login and logout.
* Each reducer has accesse of state and action.
* And, action provide payload.(payload is an object which provide information from component to store)

const authSlice = createSlice({

    name:"auth",

    initialState,

    reducers:{

          login:(state,action)=>{

            state.status=true;

            state.userData= action.payload.userData;

          },

          logout☹state,action)=>{

            state.status=false;

            state.userData=null;

          }

    }

})

* Then export main reducer and actions separately.

export const {login,logout} =authSlice.actions;

export default authSlice.reducer;

1. **Whenever app loads, we have to check whether the user is logged in or not, so will work on App.jsx now**

* Check it from state that is user is logged in or not.
* And , if user is logged in then we will show some things to user and if not, then we will show what is needed.
* Make a state loading, so that when we fetch data from appwrite, because it may take some time to fetch data. So, we will do conditional rendering.
* Hence, if loading is true=> show loading

And if loading false=> show data

**const [loading,setLoading] = useState(true)**

* In starting, we have set its default value true because we will use useEffect. And, in useEffect we will change its state from true to false
* Make a variable named dispatch using useDispatch hook from react-redux. We need it to change state

(user interacts-> dispatch an action-> check store for suitable reducer for the action dispatched-> reducer will process old state from store and send new state)

* Now, we need authentication service to fetch data from appwrite about user status

import authService from “./appwrite/auth”

* **In authService, we had created a method getCurrentUser, use it.**

**And also import login and logout methods, so that when we fetch user’s status we can update store as well and hence ui. We will use login and logout methods(reducers) by dispatching them.**

* Use this method inside useEffect hook so that whenever app renders, it will check the user’s login/logout status and show userData accordingly.
* useEffect(()=>{

authService.getCurrentUser()

.then((userData)=>{

if(userData){

dispatch(login({userData}))

} else{

dispatch(logout())

}

})

.finally(()=>setLoading(false))

},[])

Here, .finally always runs. .then also provides a callback.

return **!loading ? (**

<div classname=””>

<div classname=””>

<Header/>

<main>

<Outlet/> // will use it after react-router-dom

</main>

<Footer/>

</div>

</div>

**) : (null)**

}

export default App;

1. **Set Provider in main.jsx for store.**

import {Provider} from ‘react-redux’

import store from ‘./store/store.js’

<React.StrictMode>

<Provider store={store}>

<App/>

</Provider>

</React.StrictMode>

1. **Now, make a folder named pages in src**

In pages , we will call all components like signup, login, cards, realtimeeditor(RT).

Production grade react components

1. In production grade projects, for example we have a login form , the inputs in that , we make separate react components of that input field only with props.
2. The Header and Footer folders in components folder , we will have 2 parts of header itself=> basic header and logout button.
3. In this component specific approach, first of all me make a folder named **container** in components folder which is in src folder.
4. And, make a file named container.jsx in container folder. It will accept properties as children.
5. It is just a box with styling properties, that can be used in future.

**FOOTER => ADD footer code from github**

1. Link is imported from React-router-dom
2. Create Logo Component and it will contain a image and add width =100px

**HEADER =>**

1. Header is slightly different, because it is optional. And we don’t want to show all values at once (acc to logout and login info).
2. Create 2 components in Header folder => Header.jsx and LogoutBtn.jsx
3. In Logout , for action we need information from store, so we will import useDispatch from react-redux and logout action from reducers, we will import it from reducers in store.
4. Import authService from config folder which contains authentication services in authService file.
5. Make a constant dispatch using useDispatch()
6. Make a logoutHandler.

*const logoutHandler = () => {*

*authSerrvice.logout().then(()=>{*

*dispatch(logout())*

*})*

*}*

*In appwrite most of the methods after completing returns a promise and as we have used backend authentication method to logout . now we need to inform and update store , so that backend and frontend can be on same page*

1. Create a button logout and add classes
2. Now, in Header.jsx we will put all the links, but we will render logout button conditionally.
3. If user is logged in => show logout button

And if not => then don’t show logout button

1. Import container, logo logout button components in

Header.jsx.

Import Links from react-router-dom (for redirection)

Import useSelector from react-redux (to know whether

the user is logged in or not by checking into the store)

Import useNavigate from react-router-dom

1. Now, first of all we will check the status of user , whether

it is logged in or not. We will do it by checking the store

using useSelector hook.

const authStatus = useSelector((state) => state.auth.status)

1. Make a const navigate using useNavigate()

const navigate = useNavigate()

1. Whenever, such navigation bars are created, an array is

created which contain objects and this array is iterated in

a loop when in use, and if we want to add any button in

navigation bar in future we can add easily an object.

1. Now add key value pairs in objects=>

**const navItems = [**

**{**

name: ‘Home’,

slug: “/”,

active: true

**},**

**{**

name: “Login”,

slug: “/login”,

active: !authStatus

**}**,

**{**

name: “Signup”,

slug: “/signup”,

active: !authStatus

**}**,

**{**

name: “All Posts”,

slug: “/all-posts”,

active: !authStatus

**}**,

**{**

name: “Add Post”,

slug: “/add-post”,

active: !authStatus

**}**

**]**

1. Now, write code for header=>

We will assemble Logo, ui list and logout button(conditionally) in header

header -> Container -> nav -> logo, uilist(all buttons and logout button)

<header classname=””>

<Container>

<nav classname=””>

<div classname=””>

<Link to=’/’>

<Logo width=’70px’/>

</Link>

</div>

<ul classname=””>

{navItems.map((item)=> (

item.active? (<li key={item.name}>

<button

classname=””

onClick={()=> navigate(item.slug)}

>{item.name}</button>

</li>):null

))}

{authStatus && (

<li>

<LogoutBtn/>

</li>

)}

</ul>

</nav>

</Container>

</header>

1. Now, we will create a common Button component for

our convenience.

* We will give parameters to Button=>

function Button(

children,

type= ‘button’,

bgColor = ‘bg-blue-600’,

bgColor = ‘text-white’,

className = ‘’,

…props

){

return(

<button className= **{`px-4 py-4 rounded-lg ${className} ${bgColor} ${textColor}`} {…props}}>**

{children}

</button>

**Note: Here, …props is used => if user added any other props**

**And, in <button> classnames if there are more properties, then we can add those additional properties by using {…props} or we can say spreading them.**

**Now, we will learn about forwardRef hook =>**

**In big projects, for example in login form , we make separate components even for input fields. And then call them in login main page. But, how do both will interact, means input components in basic projects have states like for username , password there are 2 diff states.**

**But in this scenario we have made component for input fields, so how the main page will get the access of their states**

**And, hence we use forwardRef hook for giving reference of states.**

**(EK COMPONENT BNAKE SARE INPUT FIELDS K LIYE USE KRENGE DIFF DIFF VALUE K SATH)**

**INTERVIEW EXAMPLE FOR forwardRef HOOK =>**

**Hm ek login form bnare h, or usme input fields h . and input field k liye hm alg component bnare h. and whi component hm sb fields k liye use krenge username password, etc.**

**And, main page alg h jha p eventually sare input field ko assemble krna h( same component with diff state or value)**

**and hme un components ki state ka access main page p chyie hoga, so hme reference dena pdega. fr hm yha forwardRef use krte h.**

1. **Now, we will create an Input component. We’ll use forwardRef hook in it (to receive ref from user and give it to the main component eventually)**

* **useId for id, we have to give same id to label in htmlfor attribute and to id in its input tag, so that Each <label> element uses the for attribute to associate with the corresponding input element via the id.**
* **Use arrow function for better code readability.**
* **Wrap your function in React.forwardRef hook.**
* **Give label, type, classname,…props in a single object in parameters and then also add ref in parameters outside the object**
* **First check if the user has given the label name.If true then show label component with attributes classname and htmlFor. Add {label} for name of label in b/w tags.**
* **Then, show input tag with attributes classname given and yours which you’ve given in parameters, id, type, ref, {…props}.**

import React, { useId } from "react";

const Input = React.forwardRef(function Input({

    label,

    type = "text",

    className = "",//we’ll add some later

    ...props

}, ref) {

    const id = useId()

  return (

  <div className="w-full">

  {label && <label

  className="inline-block mb-1 pl-1"

  htmlFor={id}

  >

    {label}

  </label>}

  <input

    type = {type}

    className = {`px-3 py-2 rounded-lg bg-white text-black outline-none focus:bg-gray-50 duration-200 border border-gray-200 w-full ${className}`}

    ref = {ref}

    {...props}

    id = {id}

  />

  </div>

)

})

export default Input;

How to use React hook form in production

1. **Now, we will make a component for our dropdown button named Select.jsx**

* Give options for dropdown, label for name, classname, props in parameters in an object and a ref in parameter outside object.
* Make a variable id using useId()
* In main div, conditionally render label component If user has given label.
* Also add htmlFor = {id} in label attributes.
* Now, render select component (by react).
* Add attributes- {…props}, id={id}, ref={ref}, classname yours and the one given in parameters.
* Now, in select map the options array.
* We’ve to map the options array conditionally, for that we have to check whether the options array has values or not. And, if we map that array without checking, and if there is no value in array , then our app will crash for sure.
* In mapping, render option component for each item in array. Give value and key attribute. **value={option}**  & as we know during mapping we have to give keys for each item.
* Render {option} b/w option tags (for showing name of option
* **{options?.map(**(option)=>(

**<option>**

{option}

**</option>**

)**)**

import React, { useId } from 'react'

function Select({

    options,

    label,

className="",

    ...props

}, ref) {

    const id = useId()

  return (

    <div className='w-full'>

    {label && <label htmlFor={id} className=''></label>}

    <select

    {...props}

    id={id}

    ref={ref}

    className={`px-3 py-2 rounded-lg bg-white text-black outline-none focus:bg-gray-50 duration-200 border border-gray-200 w-full ${className}`}

    >

        {options?.map((option)=>(<option key = {option} value={option}>

          {option}

        </option>))}

    </select>

    </div>

  )

}

export default React.forwardRef(Select)

* We can also use forwardRef use like above, by wrapping component in it end while exporting.

1. **Now, we will make component for small cards that are visible when you are logged in , which show the whole post by clicking on them – named PostCard.jsx**

* We need info, for that we need to import service from appwrite
* Import Link( from react-router-dom), bcz we will make the whole postcard a link.
* **Now, to display the postcard, we have to pass some props, which we’ll get again from appwrite on applying query**
* Add, **$id, title, featuredImage** in props. (we can add more)
* Wrap the whole component in Link component.
* <Link to={`**/post/${$id**}`}> , in link we don’t need to give full url as path, we can give the path from our current location.
* Render a div in link with classname of your choice.
* In first div, we will show an image and a text only.
* Then, again add a div in above div.(for image).
* In image, we have to show preview of image. For that, we can use appwrite service method **getFilePreview by passing fileId**.

And the one who is calling the card is already passing the id.

* **featuredImage is id of image and $id is id of each post**
* In image tag, write classname of your choice , alt={title} and src={**service.getFilePreview(featuredImage)**}
* Outside the innermost div, add a h2 tag to show title, add classname of your choice

import React from 'react'

import service from '../appwrite/config'

import {Link} from 'react-router-dom'

function PostCard({$id, title, featuredImage}) {

  return (

<Link to={`/post/${$id}`}>

    <div className='w-full bg-gray-100 rounded-xl p-4'>

      <div className='w-full justify-center mb-4'>

        <img src={featuredImage}/>

      </div>

    </div>

    </Link>

  )

}

export default PostCard

1. **Now, we will make a login component.**

* We will use React hook form.
* In react Hook Form, register is a way of form handling.

And handleSubmit is a method.

* Import useState and {Link, useNavigate} from react-router-dom
* We also need login from authSlice (from store). We can also use it by name of our choice. **Import {login as authLogin}** from path
* Import Button, Input, Logo from index
* We need authentication service from appwrite, as we are making login component. For that, Import **authService** from appwrite.
* Import useDispatch from react-redux
* Import useForm from react-hook-form
* Make variables navigate & dispatch using useNavigate() & useDispatch()
* Now, we will take register & handleSubmit from useForm() **const {register, handleSubmit} = useForm()**
* Make state for errors using useState.

**const [error, setError] = useState(“”)**

* **We’ll make a method named login,** but we already have handleSubmit method, **And in react-hook-form handleSubmit is a method in which we give our handlesubmit method for form submission & here we’re making our handleSubmit method with “login” name** avoid confusion.

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

setError(“”)

**}**

* In login method, firstly empty the error state. **Always empty the error state in submit methods while making register or login forms, bcz after starting submission, error should be cleaned.**

**Also, use try catch block for catching error if any while sending info. In try block, use await as we’ve already used async.**

* In await call method login from authService, and store the data thet we get from it in a variable named **session.**

If we get session means user is logged in and if not then user is not logged in.

And if we get session, then fetch userData from **getCurrentUser() method of authService** (appwrite authentication service).

Again check if we get userData, then dispatch login **(here authLogin)**  & pass user data in authLogin

And if we’ve got the user, then navigate it to route using navigate.

**Note: While using Link, we’ve to click it. But in navigate, we can programmatically send anywhere.**

* Now, render a main div, & a div inside main div with classnames of our choice. And, a third div inside second div.
* In the innermost div, add Logo
* Outside the innermost div, add h2 with text – **Sign in to your account.**
* Then a p tag with text – **Don’t have any account.**  And, make it clickable by wrapping it in Link component with path to signup page.

**<p> <Link to =”/signup”** classname**> Sign Up</Link></p>**

* Now, we’ll display error if any by using p tag. We’ll render it conditionally.

**{error && <p classname>**{error}**</p>}**

* Now, we will render our form. In form attributes, in handleSubmit put our method.

**On submission of form, handleSubmit is actually an event which’ll get called. And this event is imp for all the input field values and as we use register there, then automatically their state manangement will be handled by this handleSubmit** (in short, it will take all the input field value states & while submission it’ll use tthem)

* Now, add classname of your choice and in form , add a div with classname of your choice. In this div, we’ll add our input components.
* **FOR EMAIL INPUT:**
* In Input component, pass

label = “Email: “

placeholder = “Enter your email”

type = “email”

{…register(“email”)}  **… is imp, bcz if we’ve used register in any**

**other Input component, then its value’ll get**

**override. And to avoid it , we’ve to spread it every**

**time.** **And, add email in register to make it unique**

Now, we are passing key value pair in register. Here, “email” is key. Now, we’ll pass an object as value. In that object, pass necessary options.

{…register(“email”, **{**

**required: true,**

**validate: {**

**matchPattern: (value) => /^\w+([.-]?\w+)**[**\*@\w+([.-**](mailto:*@\w+(%5b.-%5d?\w+)*(\.\w%7b2,3%7d)+$/)

**]?\w+)\*(\.\w{2,3})+$/**

**.test(value) ||**

**"Email address must be a valid address",**

**}**

**}**)}

**NOTE: validate is used for pattern. In validate, we will pass an object in which we pass the pattern we want.** Use matchPattern method which give a callback for passing pattern.

To search patterns of our choice, search regexr on google. And here we want for email, then search email and choose the one you want.

Our pattern is in slashes(/) before .test (value).

And, from **.test(v) is for test for value.** And pass the value that we are passing in matchPattern’s callback.

And **if the value that we’re getting is tested by the pattern expression.**

**If not, then use || and write “Email address must be a valid address”**

{...register("email", {

  required: true,

  validate: {

      matchPatern: (value) => /^\w+([.-]?\w+)\*@\w+([.-]?\w+)\*(\.\w{2,3})+$/.test(value) ||

      "Email address must be a valid address",

  }

})}

* **FOR PASSWORD INPUT:**
* Same as email. Use password in label type, placeholder.
* In register object, we’re passing only required: true option,

(if we need any other option, we can pass)

<Input

label="Password: "

type="password"

placeholder="Enter your password"

{...register("password", {

    required: true,

})}

/>

* **Button component for Sign in:**

<Button type="submit" className="w-full">

  Sign in

</Button>

Now, for revision make signup component same as login.

1. **Now, we’ll make Signup component**

* Make our handleSubmit method with name **create or signup.**
* Now, as we are creating an account. Use createAccount method from authService (appwrite authentication service).
* Store in a variable userData.

If we’ve got userData then fetch current user using **getCurrentUser method from authService.**

**And, if we are getting current user, store it in variable named userData(again)** //for conveneience

Now, if we’re getting currentUser, we’ve to update store.

To update store, use dispatch and send userData in login method through dispatch.

And after passing userData in store, navigate it to home route - **navigate(“/”)**

* **Render a main div.** Second div inside main div for logo.

**Wrap logo in span in second div.**

**Outside second div, use h2 – “**Signup to create account”

Then, a p tag – “Already have an account”

Inside p, use Link to navigate to sign in page.

<p>

  Already have an account?

  <Link to="/login">Sign In</Link>

</p>

* Now, display errors using p tag same as login.
* Then, render form. Pass our method in handleSubmit method of react-hook-form in onsubmit.

<form onSubmit={handleSubmit(create)}>

Your code

</form>

1. In form, add a div. In this div, add our Input components for Full Name, Email, Password same as login.

<Input

  label="Full Name: "

  placeholder="Enter your full name"

  {...register("name", {

    required: true,

  })}

/>

We are not giving more options in register key value pairs other than required

And, Email & Password same as login

1. Then, a button component for Sign up or Create Account text same as login

<Button type="submit" className="w-full">

Create Account

</Button>

AUTHENTICATION LAYOUT

It is actually a mechanism to protect pages and routes.

Till now, we haven’t written any protection mechanism, in auth we know we have states but we are not using them so how will we protect it.

So, we will make a container for it (a protected container). This container is empty be default, which will decide whether to show values in it or not.

1. Make a component AuthLayout. And we are naming function inside it **Protected.**

**(yes, file name and function name can be different)**

export default function Protected({children, authentication = true}) {

// YOUR CODE

}

Protected name bcz we’ll conditionally render its children.

1. In props, pass children and authentication. In starting, we’re giving value of authentication = true. But if someone is calling

this component, maybe he’ll give its value = false.

So, we will check status

(We will explain it later…..)

1. Import **useEffect, useState** from react, **useSelector** from react-redux, **useNavigate** from react-router-dom.
2. Then make variable navigate using useNavigate.

A state named loader using useState with default state true.

Now, we will ask store using useSelector, whether the user is logged in or not. And, store the received value in variable named **authStatus.**

const navigate = useNavigate()

const [loader, setLoader] = useState(true)

const authStatus = useSelector(state => state.auth.status)

1. **Now, useEffect will tell us on which route to send the current user. And on change of which fields, there is a need for recheck.**

So, for rechecking there are dependencies = authStatus, navigate, authentication. Put these in dependency array of useEffect.

**(authentication – user ne agr authentication wgrh bheja h to recheck it)**

**(authStatus changes will be handled by useSelector)**

**(navigate handles user kha se kha gya h)**

Now, check using if else block ->

useEffect(() => {

        //TODO: make it more easy to understand

        // if (authStatus ===true){

        //     navigate("/")

        // } else if (authStatus === false) {

        //     navigate("/login")

        // }

        //let authValue = authStatus === true ? true : false

        if(authentication && authStatus !== authentication){

            navigate("/login")

        } else if(!authentication && authStatus !== authentication){

            navigate("/")

        }

        setLoader(false)

    }, [authStatus, navigate, authentication])

### Why Only Two Conditions are Needed ?

 **Protected Route** (Meant for authenticated users) => Redirects unauthenticated users to login; allows authenticated users.

 **Guest Route** (Meant for unauthenticated users) => Redirects authenticated users to home; allows unauthenticated users.

These two conditions handle all possible scenarios for route access based on user authentication status

### Props

1. **children:** The content that will be rendered if the user meets the authentication requirements.
2. **authentication:** A boolean that specifies the type of protection for the route:

If true, the route requires the user to be authenticated (logged in).

If false, the route requires the user to be unauthenticated (logged out).

1. **State and Selectors**

**authStatus:** This value is fetched from the Redux store and represents the current authentication status of the user. It can be true if the user is logged in and false if the user is logged out.

### Variable Meanings:

* **Protected Route (authentication = true):**

This route requires the user to be authenticated (logged in).

* **Guest Route (authentication = false):**

This route does not require the user to be authenticated (meant for guests who are not logged in).

* **authStatus = true**: The user is authenticated (logged in).
* **authStatus = false:** The user is not authenticated (not logged in).

1. **useEffect Logic =>** The useEffect hook in the component checks condition and navigates the user accordingly:

### Protected Route (authentication = true):

* Meant for authenticated users.
* This route requires the user to be authenticated

1. **User is Authenticated (authStatus = true)**:
   * Condition: authentication && authStatus !== authentication
   * Evaluation: true && true !== true
   * Simplification: true && false (which is false)
   * **Outcome**: The condition is false. No redirection occurs. The user stays on the current page, accessing the protected route because they are authenticated.
2. **User is Not Authenticated (authStatus = false)**:
   * Condition: authentication && authStatus !== authentication
   * Evaluation: true && false !== true
   * Simplification: true && true (which is true)
   * **Outcome**: The condition is true. The user is redirected to the login page because they are not authenticated.

### Guest Route (authentication = false):

* Meant for unauthenticated users.
* This route does not require the user to be authenticated.

1. **User is Authenticated (authStatus = true)**:

* Condition: !authentication && authStatus !== authentication
* Evaluation: false && true !== false
* Simplification: true && true (which is true)
* **Outcome**: The condition is true. The user is redirected to the home page because they are authenticated and should not access guest routes.

1. **User is Not Authenticated (authStatus = false)**:

* Condition: !authentication && authStatus !== authentication
* Evaluation: false && false !== false
* Simplification: true && false (which is false)
* **Outcome**: The condition is false. No redirection occurs. The user stays on the current page, accessing the guest route because they are not authenticated.

1. **In return statement =>** 
   * **Loading State (loader = true)**: Shows "Loading..." to indicate ongoing data fetch or process.
   * **Loaded State (loader = false)**: Renders either the main content (children) of the route or performs redirection logic based on authentication status (authStatus).

**Protected Routes**

**Protected Routes** are parts of an application that are only accessible to authenticated users. These routes require the user to be logged in to access them. Examples include:

* User dashboard
* Profile page
* Account settings
* Any page that displays user-specific data or allows users to perform actions that require authentication

**Guest Routes**

**Guest Routes** are parts of an application that are only accessible to unauthenticated users. These routes are meant for guests, or users who are not logged in. Examples include:

* Login page
* Registration page
* Forgot password page
* Any page that provides access to features or information meant for users who have not logged in

useEffect(() => {

        if (authentication && authStatus !== authentication) {

            navigate("/login"); // Protected route, Redirect unauthenticated users to the login page, & authenticated users stay on same page

        }

else if (!authentication && authStatus !== authentication) {

            navigate("/"); // Guest route, Redirect authenticated users to the home page & allows unauthenticated users.

        }

        setLoader(false); // Set loader to false after checking authentication status

    }, [authStatus, navigate, authentication]

**Usage:**

import React from 'react';

import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

import Protected from './Protected';

import Dashboard from './Dashboard';

import Login from './Login';

const App = () => {

  return (

    <Router>

      <Routes>

        <Route path="/login" element={<Protected authentication={false}><Login /></Protected>} />

        <Route path="/dashboard" element={<Protected authentication={true}><Dashboard /></Protected>} />

      </Routes>

    </Router>

  );

};

export default App;

* + Protected component is used to wrap routes (/login and /dashboard) based on authentication requirements.
  + authentication={true} for /dashboard ensures only authenticated users can access.
  + authentication={false} for /login allows unauthenticated users to access.

**Adding form and slug values**

**INTRO:**

RTE will be a separate component, just like input components in which we made separate input components and form and for state management and values we‘d used forwardref hook.

Similarly, we can use forwardref in rte. But , here as we have react-hook-form we will see how to do the same technique in it.

We will design this form in a way, so that it can be used for post submit, post edit.

Edit - sari values load krdo

New – sari values empty dedo

**AND, We will also track an input form, based on this we will design slug , in which we will add - , if there is any space used.**

* **Here, we will use controller from react-hook-form for giving reference just like forwardRef hook.**
* **For this, we can add our requirements in controller. It has arguments for this like name, control, render, defaultValue, rules, shouldUnregister, disabled** (as mentioned in docs)
* And, whenever this RTE component will be called, caller will provide control & all the props values.
* Now, start code=>

Create a main div with classname of ur choice.

Then, render label conditionally.

Then, render Controller component. We can pass name, control, rules, render(function).

-**name** name will be passed using || condition

- **control** when we’ll pass control, it will give control to parent

component which is calling this RTE component

(everything like events, values, data, states, etc.)

**-render={({ field }) => (...)**

* The render prop in Controller allows you to define how a form input or control should be rendered within a form managed by react-hook-form.
* The Controller component is used to connect form inputs from external libraries (like TinyMCE, DatePicker, etc.) or custom components to the react-hook-form's form state.
* It expects a render prop function where you can define how the control (input, select, custom component) should be rendered and how it interacts with react-hook-form.
* Props Passed: Inside the render function, Controller passes an object (often named field) containing properties like onChange, value, onBlur, name, and ref.
* Responsibility: You are responsible for correctly integrating these field properties into your custom UI or component to ensure proper interaction with react-hook-form.

**The render prop within react-hook-form's Controller component returns JSX elements, such as form inputs or custom components, that you define inside its function.**

**<input {...field} placeholder="First Name" />:**

Spreads the field props into the <input> element.

This connects the input to the form state managed by react-hook-form.

When you spread {...field} into your input component, you are effectively connecting your input to the form state.

 **value**:

* Sets the current value of the input.
* Keeps the input value in sync with the form state.

 **onChange**:

* Updates the form state when the input value changes.
* Allows react-hook-form to manage the value of the input.

 **onBlur**:

* Triggers validation and other onBlur-related functions when the input loses focus.
* Useful for touch validation (validating the input when the user interacts with it).

 **name**:

* Specifies the name of the input, used as a key in the form data object.
* Ensures that the value is stored correctly in the form state.

 **ref**:

* Provides a reference to the input element.
* Necessary for focusing the input programmatically and for validation purposes.

### How to use thes properties or methods in it?

* **Basic Usage**: Spread {...field} directly into your input component to connect it to the form state.

const MyForm = () => {

  const { control, handleSubmit } = useForm();

  const onSubmit = data => {

    console.log(data);

  };

  return (

    <form onSubmit={handleSubmit(onSubmit)}>

      <Controller

        name="firstName"

        control={control}

        defaultValue=""

        render={({ field }) => (

          <input

            {...field} // Spreads the field properties into the input

            placeholder="First Name"

          />

        )}

      />

      <button type="submit">Submit</button>

    </form>

  );

};

* **Selective Usage**: Destructure the field object to use specific properties like value, onChange, onBlur, and ref.

<Controller

  name="firstName"

  control={control}

  defaultValue=""

  render={({ field: { value, onChange, ...rest } }) => (

    <input

      value={value} // Set the input value

      onChange={onChange} // Handle change events

      placeholder="First Name"

      {...rest} // Spread the remaining field properties

    />

  )}

/>

* **Custom Components**: Use the field properties with custom or third-party components to manage their state and behavior within the form.

//Custom Input Component Code(Input.jsx)

const Input = ({ value, onChange, onBlur, name, ref, ...props }) => (

  <input

    value={value}

    onChange={onChange}

    onBlur={onBlur}

    name={name}

    ref={ref}

    {...props}

  />

);

Or Third Party Components. Here, we are using Editor component from tinyMCE library

//Form Component Using Custom Input(RTE.jsx)

//1. Basic Usage (Spreading all field properties)

const MyForm = () => {

  const { control, handleSubmit } = useForm();

  const onSubmit = data => {

    console.log(data);

  };

  return (

    <form onSubmit={handleSubmit(onSubmit)}>

      <Controller

        name="firstName"

        control={control}

        defaultValue=""

        render={({ field }) => (

          <Input

            {...field} // Spread all field properties into CustomInput

            placeholder="First Name"

          />

        )}

Or

render={({ field }) => (

          <Editor

{...field} // Spread all field properties into the Editor

            init={{

              height: 500,

              menubar: true,

              plugins: [],

              toolbar: '',

            }}

          />

        )}

      />

      <button type="submit">Submit</button>

    </form>

  );

};

//2. Specific Usage (Using specific field properties)

const MyForm = () => {

  const { control, handleSubmit } = useForm();

  const onSubmit = data => {

    console.log(data);

  };

  return (

    <form onSubmit={handleSubmit(onSubmit)}>

      <Controller

        name="firstName"

        control={control}

        defaultValue=""

        render={({ field: { value, onChange, onBlur, ref, ...rest } }) => (

          <Input

            value={value} // Set input value

            onChange={onChange} // Handle change events

            onBlur={onBlur} // Handle blur events

            ref={ref} // Set ref for input

            placeholder="First Name"

            {...rest} // Spread remaining field properties

          />

        )}

      />

Or

render={({ field: { value, onChange, onBlur, name, ref } }) => (

          <Editor

            initialValue={value} // Set the initial value

            init={{

              height: 500,

              menubar: true,

              plugins: [],

              toolbar: '',

            }}

            onEditorChange={onChange} // Handle change events

            onBlur={onBlur} // Handle blur events

            id={name} // Set the ID

            ref={ref} // Provide a reference to the editor

          />

        )}

      <button type="submit">Submit</button>

    </form>

  );

};

### Explanation:

* **initialValue**: Sets the initial value of the editor.
* **onEditorChange**: Connects the editor's change event to the form state using onChange.
* **onBlur**: Handles blur events.
* **id**: Sets the id of the editor (using name).
* **ref**: Provides a reference to the editor.

**As we are making a separate component for Rich Text Editor, we will use Editor component from tinyMCE library in RTE component.**

**(Just like we’ve made separate input Component name Input.jsx in which we used input component from React Library)**

* **Plugins** in TinyMCE provide additional features like image handling, advanced list formatting, code editing, and more.
* **Toolbar**: Buttons corresponding to the functionalities of these plugins are automatically added to the toolbar when you configure them in the init object.

When you specify plugins in the init object of the TinyMCE Editor component, they provide additional functionalities that will be accessible through the toolbar and other parts of the editor interface.

Here, is the final code:

import React from 'react'

import {Editor } from '@tinymce/tinymce-react';

import {Controller } from 'react-hook-form';

export default function RTE({name, control, label, defaultValue =""}) {

  return (

    <div className='w-full'>

    {label && <label className='inline-block mb-1 pl-1'>{label}</label>}

    <Controller

    name={name || "content"}

    control={control}

    render={({field: {onChange}}) => (

        <Editor

        initialValue={defaultValue}

        init={{

            initialValue: defaultValue,

            height: 500,

            menubar: true,

            plugins: [

                "image",

                "advlist",

                "autolink",

                "lists",

                "link",

                "image",

                "charmap",

                "preview",

                "anchor",

                "searchreplace",

                "visualblocks",

                "code",

                "fullscreen",

                "insertdatetime",

                "media",

                "table",

                "code",

                "help",

                "wordcount",

                "anchor",

            ],

            toolbar:

            "undo redo | blocks | image | bold italic forecolor | alignleft aligncenter bold italic forecolor | alignleft aligncenter alignright alignjustify | bullist numlist outdent indent |removeformat | help",

            content\_style: "body { font-family:Helvetica,Arial,sans-serif; font-size:14px }"

        }}

        onEditorChange={onChange}

        />

    )}

    />

     </div>

  )

}

* **init:** Configuration object for setting up the TinyMCE editor.
* **initialValue**: Initial content displayed in the editor.
* **height**: Editor height in pixels.
* **menubar**: Controls visibility of the menu bar.
* **plugins**: Array of plugins to include.
* **toolbar**: Defines the buttons and their layout in the toolbar.
* **content\_style**: CSS styles applied to the content area.
* **onEditorChange**: Callback for handling content changes in the editor.

**PostForm component**

Now, we will create a component named PostForm.jsx in post-form folder. This PostForm is the main component which will be responsible for creation & updation of posts.

In this , we will assemble all the small separate components like Input, Select, RTE, Button for our form fields.

We’ll have these fields in our form which will be used for creation & updation of our posts:

* title – Input
* slug – Input
* Editor(for blogs) – RTE
* featuredImage – Input
* status(for active/inactive) – Select(Dropdown component)
* Update/Submit button – Button

### Steps for code of PostForm.jsx =>

### Import all important things: all hooks(useForm, useNavigate, useSelector,useCallback), all the small input components of this big form, service from appwrite for using database service(create&update)

### We'll get a post prop which user will provide

function PostForm({post}) {

//code

}

### Define all the variables using hooks that we imported: useForm, useNavigate, useSelector

### We’ve to make a submit function- it will contain logic of both creation & updation of post.

### LOGIC- if post is there, then updation happens, & if post is not there, then creation happens(will use if-else block)

### In return block, create a form, then create 2 sections in it vertically.

### One for title input, slug input & Editor with width 2/3

### Other one for featuredImage input, status select dropdown & submit Button with width 1/3

 <form>

         <div className="w-2/3 px-2">

            <Input/>

            <Input/>

            <RTE/>

         </div>

         <div className="w-1/3 px-2">

            <Input/>

            <Select/>

            <Button type="submit" className="w-full">

                {post? "Update":"Submit"}

            </Button>

         </div>

    </form>

### We need info from useForm.

### First we’ll learn about useForm:

### useForm is a custom hook for managing forms with ease.

### It takes one object as optional argument. The options object can include configuration like defaultValues, validation rules, etc.

### It returns an object with several properties and methods to manage form state, validation, and submission. These properties help manage forms efficiently with minimal re-renders.

### These include:

### register: Register input fields.

### handleSubmit: Handle form submission.

### watch: Watch and get the current value of specified inputs.

### setValue: Programmatically set the value of an input.

### control: Control object for custom components.

### getValues: Retrieve the current values of inputs.

### errors: Object containing validation errors.

### reset: Reset the form to default values.

### setError: Manually set an error for an input.

### clearErrors: Clear validation errors.

### trigger: Manually trigger validation.

### formState: Object containing form state like isDirty, isValid, etc.

### Destructuring - You extract these properties (register, handleSubmit, watch, setValue, control, getValues) from the returned object and assign them to corresponding variables. eg:

| **Aspect** | **Traditional Form Handling** | **react-hook-form** |
| --- | --- | --- |
| **State Management** | Managed via useState for each input | Managed via internal refs |
| **Change Handling** | Custom handleChange function | register function handles it internally |
| **Code Complexity** | More boilerplate for managing state | Less boilerplate, simpler syntax |
| **Performance** | Re-renders on each state change | Fewer re-renders, better performance |
| **Validation** | Manually implemented in handleChange | Built-in support with register |

react-hook-form leverages ref to manage form inputs, making them "uncontrolled" components by default. This approach allows for better performance since the input state is managed by the DOM rather than React, reducing unnecessary re-renders.

In react-hook-form, when you use the register function to link an input field, it assigns a ref to that input. This allows the library to track the input's value and state changes without requiring React state management.

Diff b/w traditional forms in react and react-hook-form

**Traditional Form Handling**

**State Management and Handle Change**

In traditional React form handling, you manage the state for each input and handle changes with a custom handleChange function:

import React, { useState } from 'react';

function TraditionalForm() {

  const [formData, setFormData] = useState({ title: '', slug: '', content: '', status: 'active' });

const handleChange = (e) => {

    setFormData({ ...formData, [e.target.name]: e.target.value });

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    console.log(formData);

  };

  return (

    <form onSubmit={handleSubmit}>

      <input name="title" value={formData.title} onChange={handleChange} />

      <input name="slug" value={formData.slug} onChange={handleChange} />

      <textarea name="content" value={formData.content} onChange={handleChange}></textarea>

      <select name="status" value={formData.status} onChange={handleChange}>

        <option value="active">Active</option>

        <option value="inactive">Inactive</option>

      </select>

      <button type="submit">Submit</button>

    </form>

  );

}

export default TraditionalForm;

**react-hook-form Handling**

**Registering Inputs**

In react-hook-form, you use the register function to register each input, which handles the value and onChange logic internally:

import React from 'react';

import { useForm } from 'react-hook-form';

function HookForm() {

  const { register, handleSubmit } = useForm({

    defaultValues: {

      title: '',

      slug: '',

      content: '',

      status: 'active',

    },

  });

  const onSubmit = (data) => {

    console.log(data);

  };

  return (

    <form onSubmit={handleSubmit(onSubmit)}>

      <input {...register('title')} />

      <input {...register('slug')} />

      <textarea {...register('content')}></textarea>

      <select {...register('status')}>

        <option value="active">Active</option>

        <option value="inactive">Inactive</option>

      </select>

      <button type="submit">Submit</button>

    </form>

  );

}

export default HookForm;

**Summary**

* Traditional Form Handling: You manage state and handle input changes manually with a custom handleChange function.
* react-hook-form Handling: The register function handles input registration, value tracking, and change handling internally, reducing boilerplate and improving performance.

AND

React Hook Form (react-hook-form) uses the following key features to simplify and streamline form handling in React:

* UseForm Hook: Centralizes form state management, validation, and submission handling.
* Register: Registers form inputs and manages their state internally.
* HandleSubmit: Manages form submission and validation.
* Watch: Tracks changes in specific input values.
* SetValue: Programmatically sets input values.
* Control: Integrates with custom components for controlled input behavior.
* Validation: Provides built-in support for form validation rules and error handling.

**useForm**

* **Centralized Form Management**: Manages overall form state, including default values, submission handling, and integration with custom components.
* **Form-Level Control**: Provides methods like handleSubmit for handling form submissions and watch for monitoring input changes.
* **Initialization**: Allows setting default values and form-wide configurations.

**register**

* **Input Registration**: Connects individual form inputs to React Hook Form, enabling state tracking, validation, and error handling for each input.
* **Validation Rules**: Defines validation rules and error messages for specific inputs.
* **Customization**: Supports customization of input behavior and validation logic using options passed to register.

**Now, again on PostForm.jsx->**

import { useForm } from 'react-hook-form';

  const { register, handleSubmit, watch, setValue, control, getValues } = useForm({

    defaultValues: {

      title: "",

      slug: "",

      content: "",

      status: "active",

    },

  });

return (

//code

)

Destructuring is a JavaScript feature that allows you to unpack values from arrays or properties from objects into distinct variables.

In this case, useForm returns an object with several properties, and you are using destructuring to extract those properties into individual variables

### Argument (options object): { defaultValues: { title: "", slug: "", content: "", status: "active" } }

### Returned object: { register, handleSubmit, watch, setValue, control, getValues, ... }

### In above we will put defaultValues conditionally,

### Here, defaultValues are those values which we'll use in this form & we need info for defaultValues which we'll get from post prop that is provided by user.

### For each value, do a query whether the user is gonna create or update. Check if post is available (means we've info), use its title or other values otherwise keep the values empty

const {register, control, getValues, watch, setValue, handleSubmit} = useForm({

    defaultValues:{

      title: post?.title || '',

      slug: post?.slug || '',

      content: post?.content || '',

      status: post?.status || 'active',

    }

  })

### Now, make variables using hooks:

### navigate from useNavigate()

### userData from useSelector. (use provided state & then fetch userData from reducer named auth)

  const navigate = useNavigate()

  const userData = useSelector((state)=> state.auth.userData)

### Now, make submit function:

### If user has submitted, means it has passed the data, so now we've 2 cases:

### If post is available- update (in if block)

### If post is unavailable- create (in else block)

## **Before that, we need to know from where we are getting the data that we will use in this submit function. In the PostForm component, the data comes from several sources:**

### post Prop: The post prop is passed to the PostForm component from its parent component. It contains data for the post being edited, including title, slug (or $id), content, status, and featuredImage.

### useForm: The useForm hook initializes the form with default values based on the post prop. If post is not provided, it uses default values like empty strings for title, slug, and content, and "active" for status.

### Redux Store: useSelector is used to retrieve userData from the Redux store. This data is used when creating a new post to associate the post with the current user.

### User(Form) Inputs: User input is managed by react-hook-form through the register function. Each form input's value is tracked and managed internally by react-hook-form.

### File Input: The featured image file input (image) is handled separately. When the form is submitted, the file data is uploaded using appwriteService.

**Data Flow Breakdown:**

1. **Form Initialization**:
   * useForm sets up the form with default values. If the post prop contains data, it pre-fills the form inputs with that data.
2. **User Input**:
   * Users interact with the form inputs. The register function from react-hook-form manages these inputs and tracks their values.
3. **Form Submission**:
   * When the form is submitted, handleSubmit gathers the input data.
   * If the post prop is provided (editing a post), it updates the post with new data.
   * If the post prop is not provided (creating a new post), it uploads the featured image and creates a new post associated with the current user.

### In react-hook-form, the form state is managed internally, and you can access all the data entered in the input fields through the data parameter in the submit function.

**Explanation**

1. **Registering Fields**:
   * Each form field is registered using the register function from react-hook-form.
   * The register function binds the field to the form state managed internally by react-hook-form.
2. **Handling Form Submission**:
   * The handleSubmit function from react-hook-form is used as the form's onSubmit handler.
   * This function takes another function (onSubmit in this case) as an argument. handleSubmit will validate the form data according to the specified validation rules (e.g., { required: true }).
3. **Accessing Form Data**:

* If the form is valid, handleSubmit will call the onSubmit function with the form data as its argument.
* The data parameter in the onSubmit function will contain all the form field values.

**Accessing Data**:

* The data object contains key-value pairs where the keys are the names of the form fields (e.g., title, content) and the values are the data entered by the user.
* You can use this data object to send the form data to a backend, display it on the screen, or perform any other required actions.

 **Accessing the File Data**:

* When the form is submitted, the data object passed to the onSubmit function will include the image file as a FileList object.
* data.image is a FileList object, and data.image[0] gives you the first selected file (assuming single file upload).

 **Handling the File**:

* You can now handle the file (e.g., send it to a backend or display a preview).
* If you need to handle multiple files, you can iterate over data.image as it is a FileList object.

Submit function code=>

* 1. First we’ll check if post ia available, if available, follow the update post code in if block otherwise else block code for new post creation.
  2. In if block, first we will handle file updation, if post is available, then make a variable named file.

The uploadFile method returns an object representing the uploaded file, which includes metadata such as the file's ID, name, size, and more, or false if there's an error, Store this in file variable if data.image[0] is present.

const file =

data.image[0]?awaitservice.uploadFile(data.image[0]):null;

1. If file is uploaded successfully, then remove the previous file from post using deletefile, which will take featureImage as arg(featuredImage is file id actually)

if(file){

  await service.deleteFile(post.featuredImage);

}

1. New file upload & previous file deletion is done.

Now, we will create a variable named dbPost,

in which we will spread previous data from our data

object and will override the required ones that need updation.

const dbPost = await service.updatePost(post.$id,{

        ...data,

        featuredImage: file?file.$id: undefined,

      })

1. Now, we've updated our post

(as Each post operation (create or update) in Appwrite generates or modifies a unique document in the collection, containing all specified data fields and metadata for that post, returned as the updated document object.)

and this dbPost have access to all the data of that document.

Check if dbPost is available, then navigate to that post.

if(dbPost){

        navigate(`/post/${dbPost.$id}`)

      }

} //updation code ends here

FINAL CODE (for updation)=>

 const submit = async (data) => {

       // updation code (if post is available, then we'll update existing post)

    if (post) {

        const file = data.image[0] ? await appwriteService.uploadFile(data.image[0]) : null;

        if (file) {

            appwriteService.deleteFile(post.featuredImage);

        }

        const dbPost =

await appwriteService.updatePost(post.$id, {

            ...data,

            featuredImage: file ? file.$id : undefined,

        });

        if (dbPost) {

            navigate(`/post/${dbPost.$id}`);

        }

    }

    // creation code (if post is not available, then we'll create a new post)

    else{

    }

  }

**Some doubts =>**

* **We’re getting post prop from parent component, we will read it later in further code**
* **We’re getting data from firstly by default values that we’ve given, then by register from user inputs in form, and simultaneously they are being updated in store as well.**
* **Why we’ve spreaded data? -**
* **Initial State:**

**When you initialize the form (data), you might prepopulate it with existing values fetched from post. This ensures that fields like title, slug, content, and status already have their current values from the existing post.**

* **Form Submission:**

**When the user interacts with the form, they might change some fields (e.g., update title or status) while leaving others unchanged (e.g., content remains the same).**

* **Handling Unchanged Fields:**

**If a field in data (e.g., content) is not explicitly updated by the user in the form, the spread operator ({ ...data }) ensures that the original value from post is retained for that field when constructing the update object.**

// Example `post` data representing existing post details

const post = {

  title: "Existing Title",

  slug: "existing-title",

  content: "Existing content of the post",

  status: "active",

  // other fields...

};

// Example `data` object representing form inputs (potentially updated by the user)

const data = {

  title: "Updated Title", // New user input

  slug: "updated-title", // New user input

  content: post.content, // Retaining original content from `post`

  status: "inactive", // New user input

  // other fields...

};

// Spread `data` to create update object for `appwriteService.updatePost`

const updateObject = {

  ...data, // Spread all fields from `data`

  // `featuredImage` and other fields can also be included as needed

};

// Perform update operation using `updateObject`

**Explanation**

* + data includes updated values for title and status entered by the user.
  + content retains its original value from post because it was not changed in the form.
  + The spread operator ({ ...data }) ensures that all fields, including unchanged ones (content in this case), are included in updateObject passed to appwriteService.updatePost.