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
2. Creating and configuring redux store and authentication slice

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.**