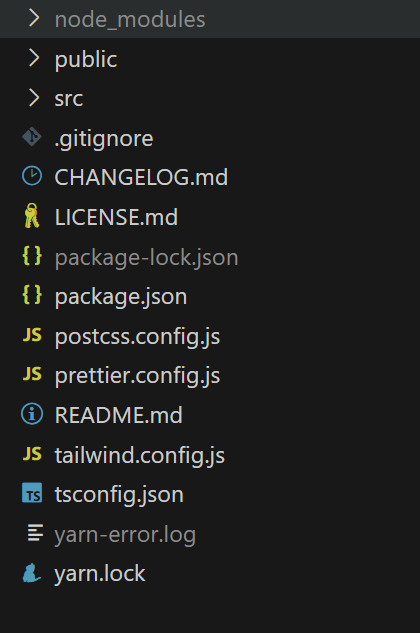
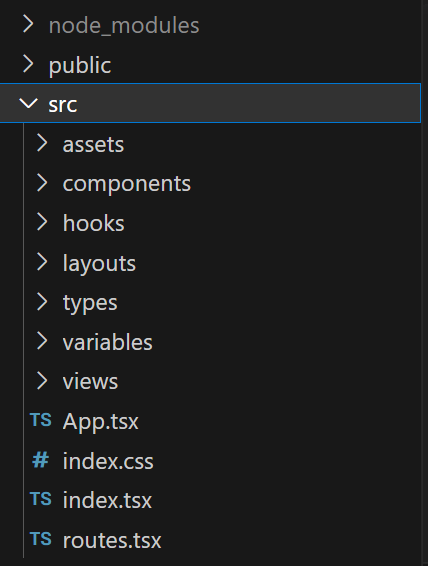
**Dao project explain (front-end side)**

**Tools:  
  
react** : react is a javascrip library that provides us development base to develop UI codes in an efficient way.   
<https://react.dev/>  
  
**tailwind**: tailwind is a css library that is used for designing. [https://tailwindcss.com/](https://react.dev/)  
  
**Javascript**: It’s a language that is responsible for operating functions in the website. [https://www.javascript.com/](https://react.dev/)  
  
**wagmi**: Wagmi is a javascrip library that is used for connecting wallet and also integration with the smart contract. [https://wagmi.sh/](https://react.dev/)  
  
  
  
**Project structure:**

In the project we have some folders and files that I explain main folders and files:  
  


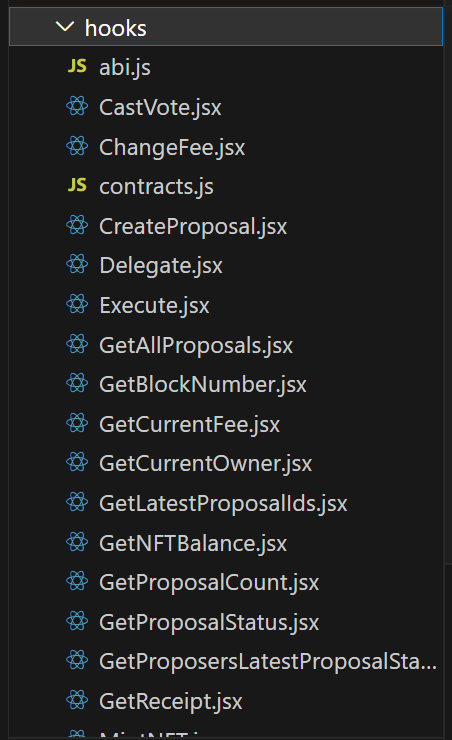
* **Node\_modules** folder contains installed libraries (with npm)
* **Public** folder in a React project is a special folder that contains static assets such as the index.html file, images, and other files that you don’t want to be processed by webpack. We don’t change this folder a lot.
* **Src** folder in a React project is the folder that contains the source code for your application. It is where all of your React-related code will live and is what you will primarily work in to build your app. This folder typically contains the entry point for your application, such as index.js, as well as other JavaScript files, CSS files, and any other assets that you want to be processed by webpack
* **Package.json** is a file that contains metadata related to the project, such as the project’s name, version, license, and dependency packages. It is used to store the list of dependency packages and to configure the project’s running and development environment. The npm install command can automatically download the required modules based on this file1. It is a kind of manifest file for the application and plays a very important role in React application development and deployment1.

**Src folder structure**

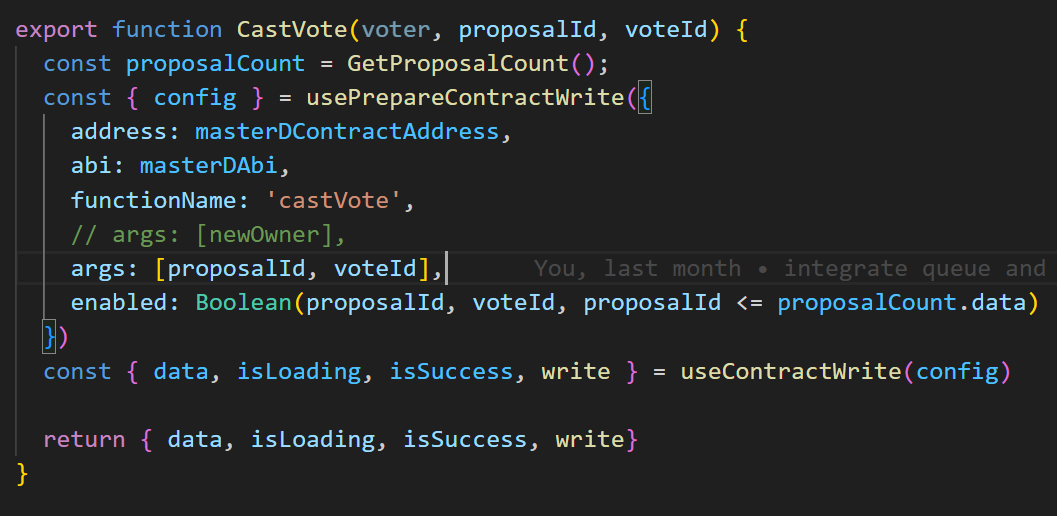


* **Assets** folder contains images and svg files that are used in the website
* **Components** folder in a React project is where you can store the React components for your application. Components are the building blocks of any React project and are used to create reusable pieces of UI that can be used across your application.
* **The hooks folder** in a React project is where you can store custom hooks that can be used across multiple components or pages in your application. Hooks are functions that let you use state and other React features in functional components. Custom hooks allow you to extract component logic into reusable functions. Hooks folder contains main functions that are connected to the smart contract.
* **The layouts folder** in a React project is not a standard folder that is included in every React project. However, some developers may choose to create a layouts folder to store layout components that can be used to define the structure of their pages. Layout components can be used to define common elements such as headers, footers, and sidebars that are shared across multiple pages. They can also be used to define the overall layout of a page, such as a two-column or three-column layout.
* **The types folder** in a React project is not a standard folder that is included in every React project. However, some developers may choose to create a types folder to store TypeScript type definitions that can be used across their application. This is especially useful when using TypeScript in a React project, as it allows you to define and organize your types in a central location.
* **The variables folder** is not a standard folder in a React project. However, some developers may choose to create a variables folder to store environment variables that can be used across their application. Environment variables are variables that are defined outside of the application and can be used to store configuration information such as API keys, database credentials, and other sensitive information.
* **The views folder** in a React project is not a standard folder that is included in every React project. However, some developers may choose to create a views folder to store higher-level components responsible for rendering specific pages or sections of the app. These components are often responsible for fetching data, managing state, and rendering the layout of the page. They can also be used to define the overall structure of a page, such as the header, main content, and footer sections.
* **The App.tsx file** in a React project is the main component of your application. It is typically the entry point for your application and is responsible for rendering the top-level components of your app. This file is usually located in the src folder of your project and is written in TypeScript if you are using TypeScript in your React project.
* **The index.tsx file** in a React project is the entry point for your application. It is typically the first file that is executed when your application starts and is responsible for rendering the root component of your app, usually the App component. This file is usually located in the src folder of your project and is written in TypeScript if you are using TypeScript in your React project.

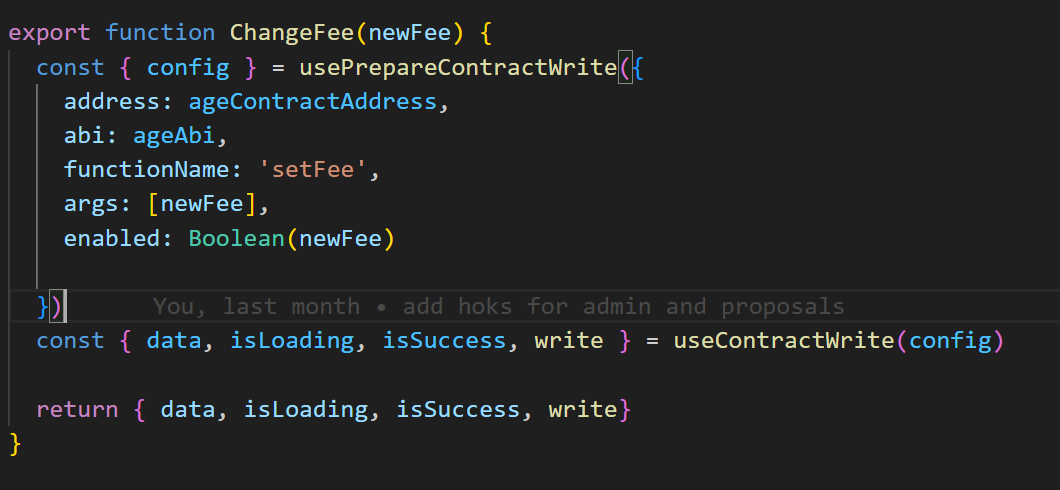
**Components folder structure**

****

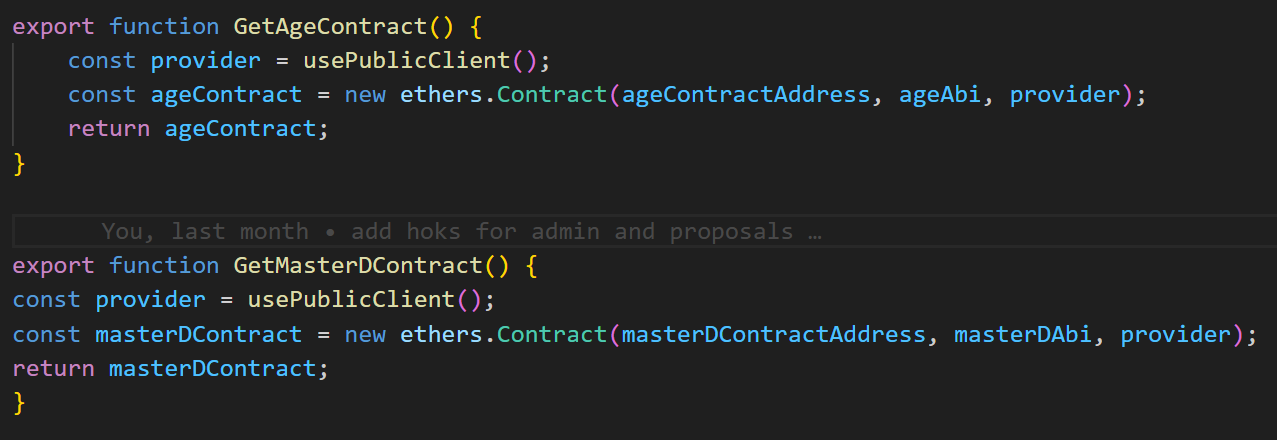
* **Abi.js file** contains abi and contract addresses that are necessary for connecting to smart contract.
* **CastVote.jsx** contains the function that handles voting of a user for the specific proposal. Each user for smart contract should has token to be able to vote.



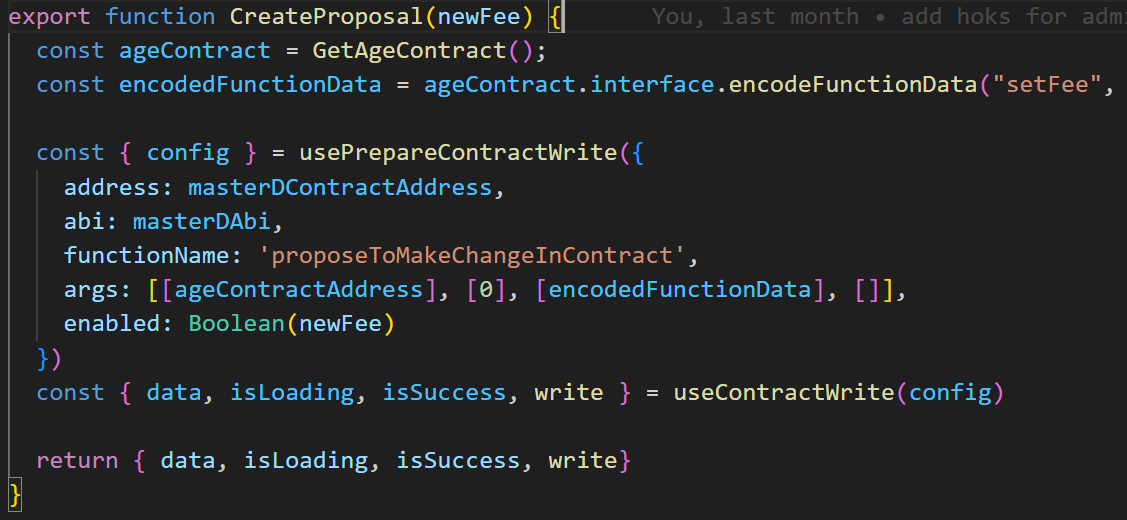
* **ChangeFee.jsx file** has function for changing fee that if owner is an account (not a dao smart contract), this account can change the fee using this function.

****

* **Contracts.js file** contains smart contract instances that we want to integrate with them using hooks functions.

****

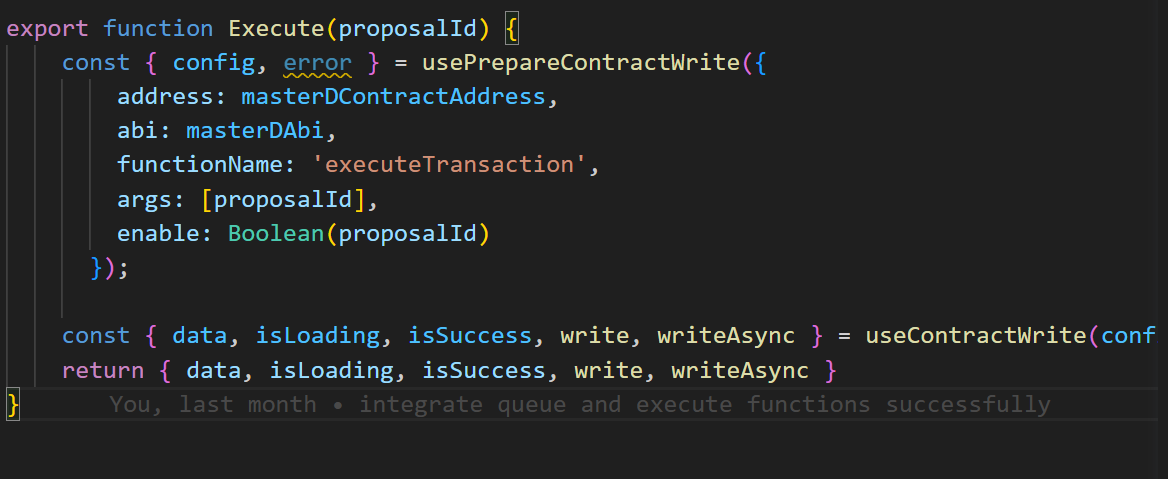
* **CreateProposal.jsx** we call this function when we want to make change in the contract (in this case we propose to make change in the fee)



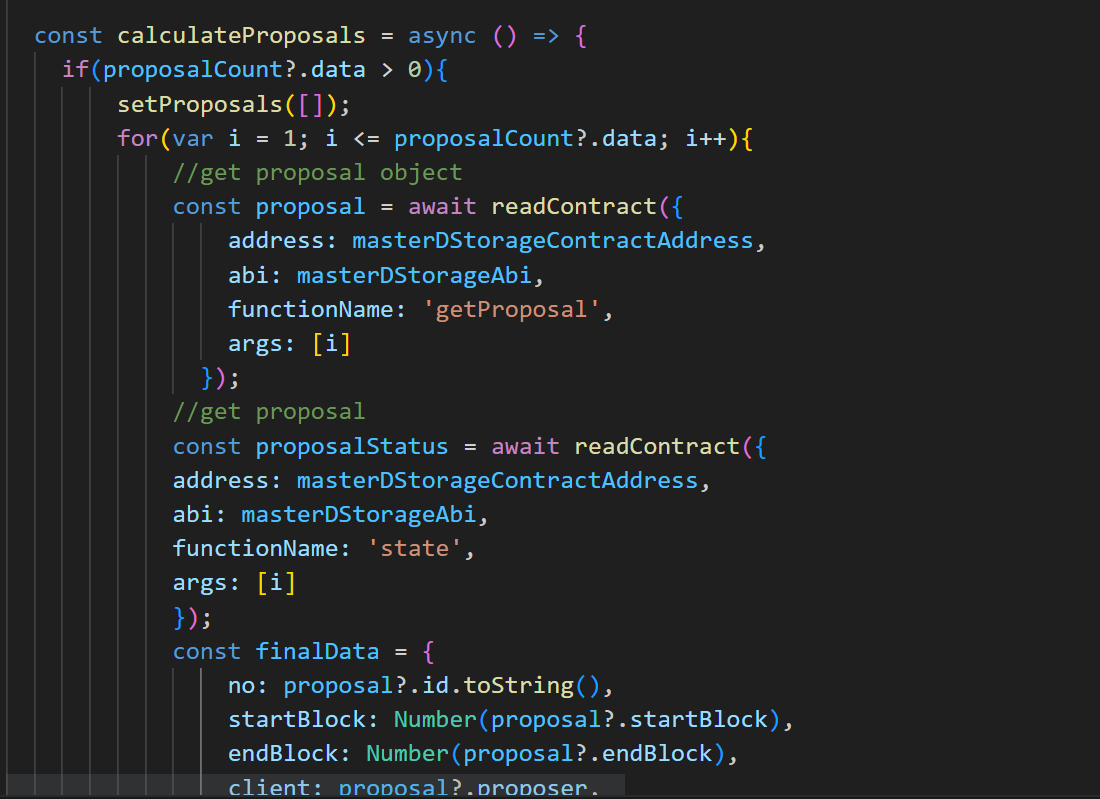
* **Delegate.jsx** has a function to delegate vote of the user.In the context of OpenZeppelin, a delegate vote refers to the ability of a token holder to delegate their voting power to another address, known as a delegate. This allows the delegate to vote on proposals on behalf of the token holder. The voting power is tracked through checkpoints, which keep a history of each account’s vote powertoken holder. The voting power is tracked through checkpoints, which keep a history of each account’s vote power.



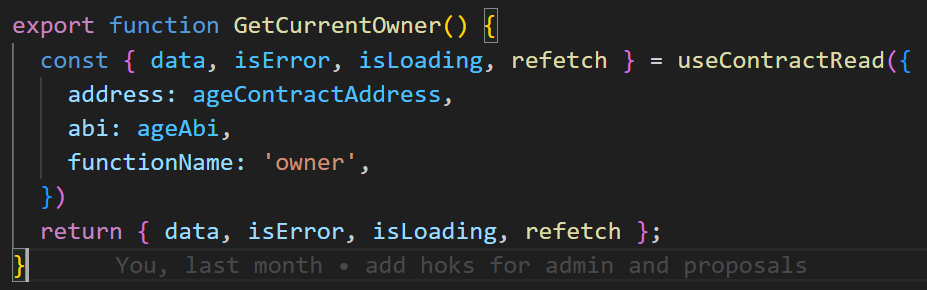
* **Execute.jsx file** contains functions for executing proposals.



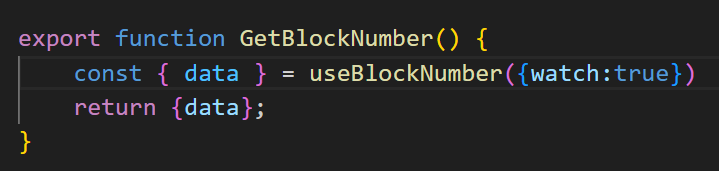
* **GetAllProposals.jsx file** gives us all created proposals to show in the website.



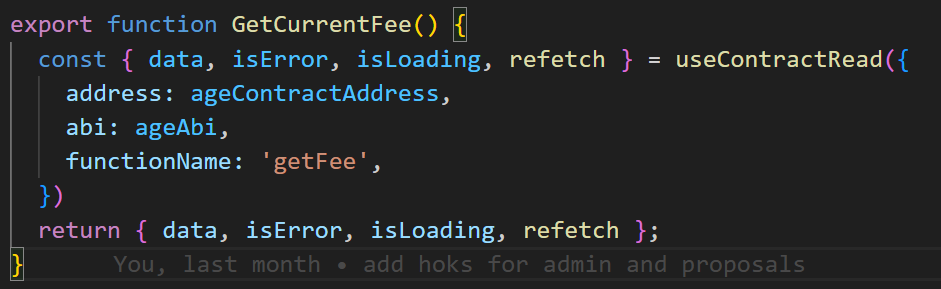
* **GetCurrentOwner.jsx** returns the owner of the fee contract.



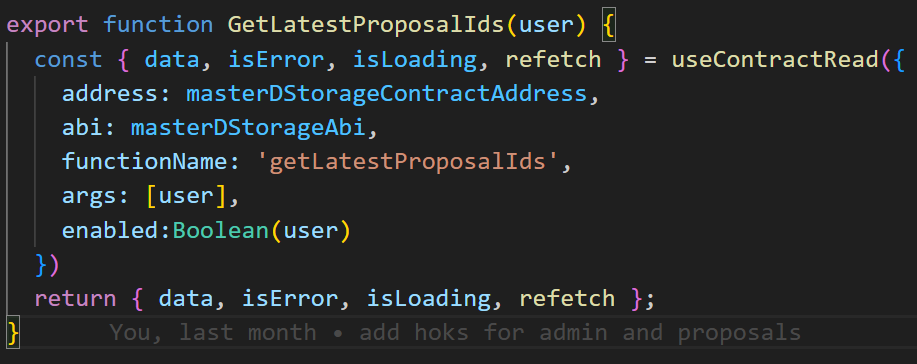
* **GetBlockNumber.jsx** gives us current block number to show in the website:

****

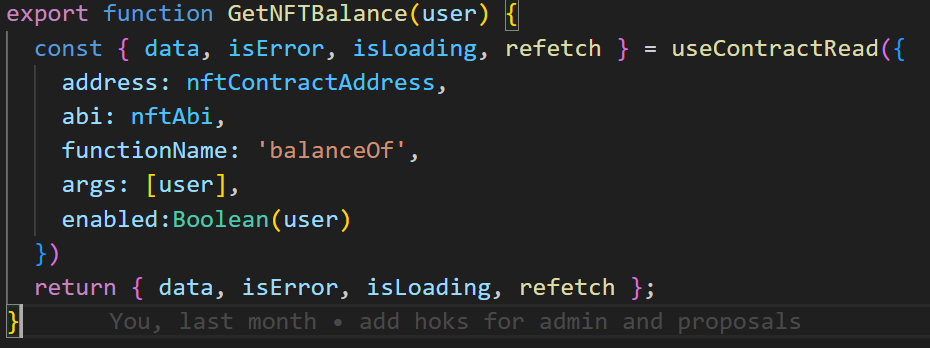
* **GetCurrentFee.jsx** returns the fee that can be changed by dao proposals.

****

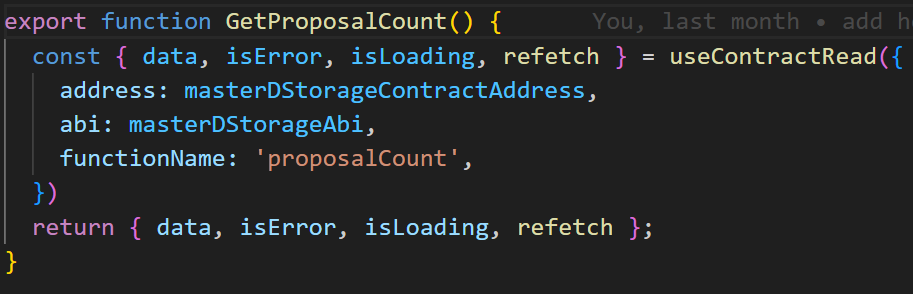
* **GetLatestProposalIds.jsx** returns the proposals of a specific user address.



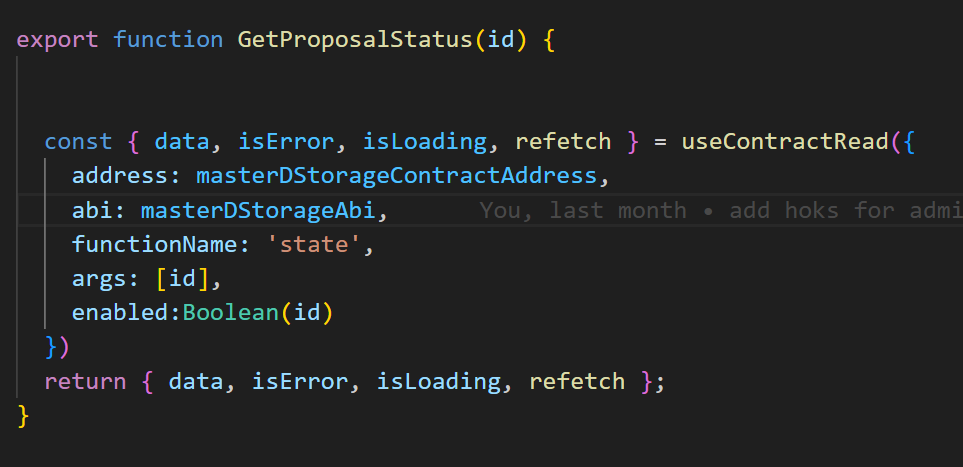
* **GetNFTBalance.jsx** returns the nft balance of a specific address.

****

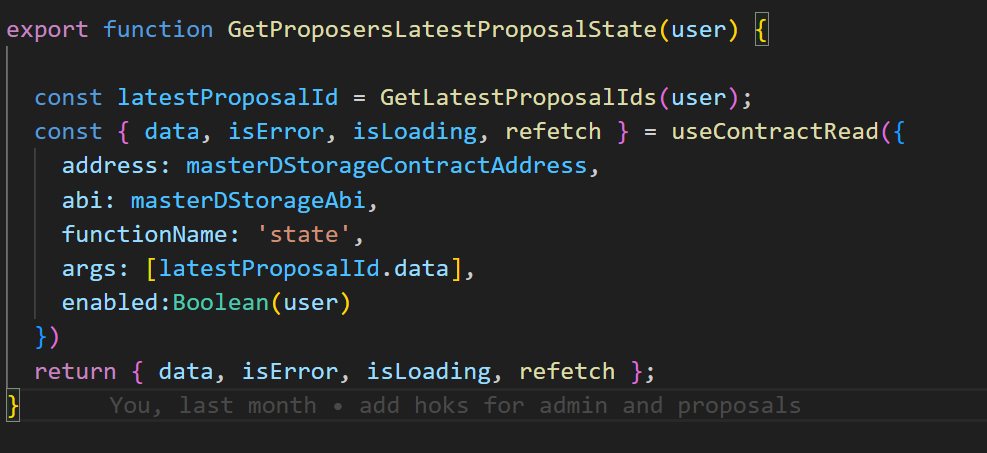
* **GetProposalCount.jsx** returns total proposals number.

****

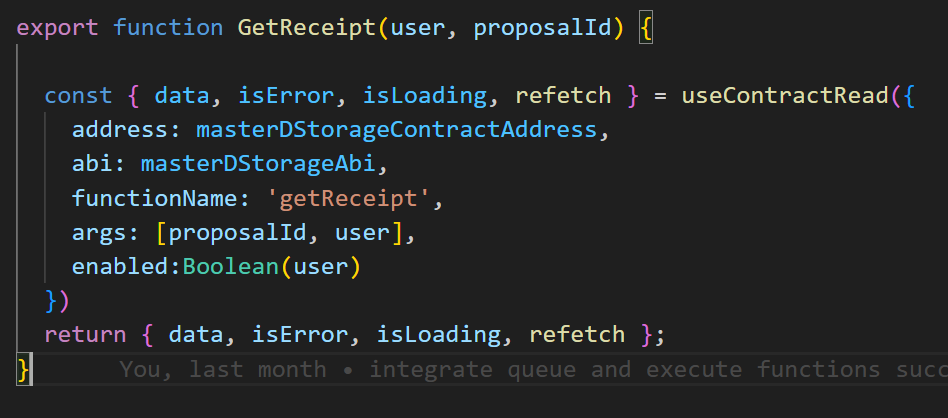
* **GetProposalStatus.jsx** returns the status of given proposal.

****

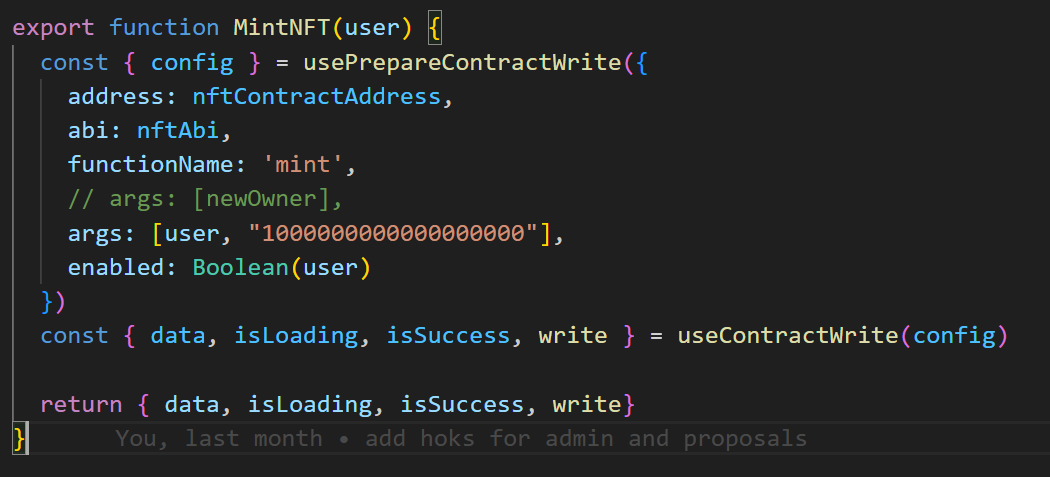
* **GetProposersLatestProposalState.jsx** returns the state of the latest proposal.

****

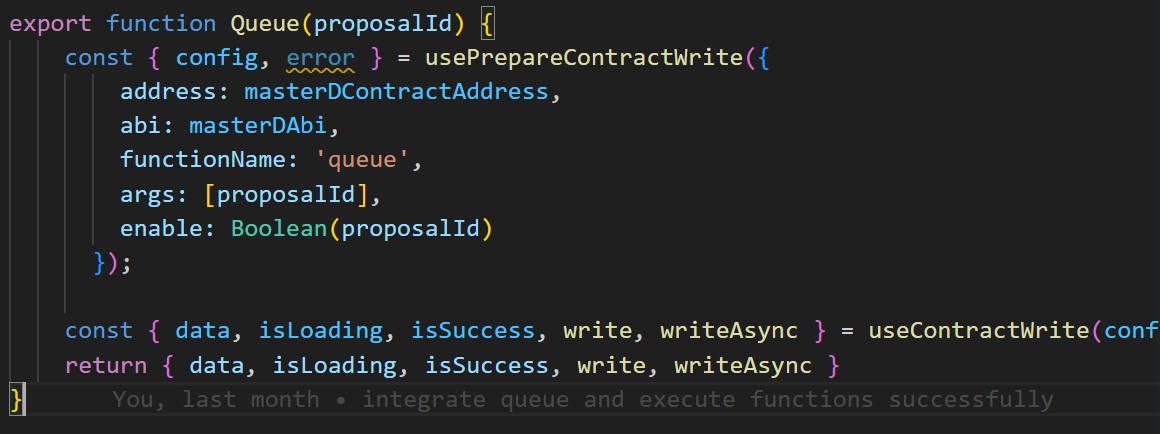
* **GetReceipt.jsx** returns of the proposal.

****

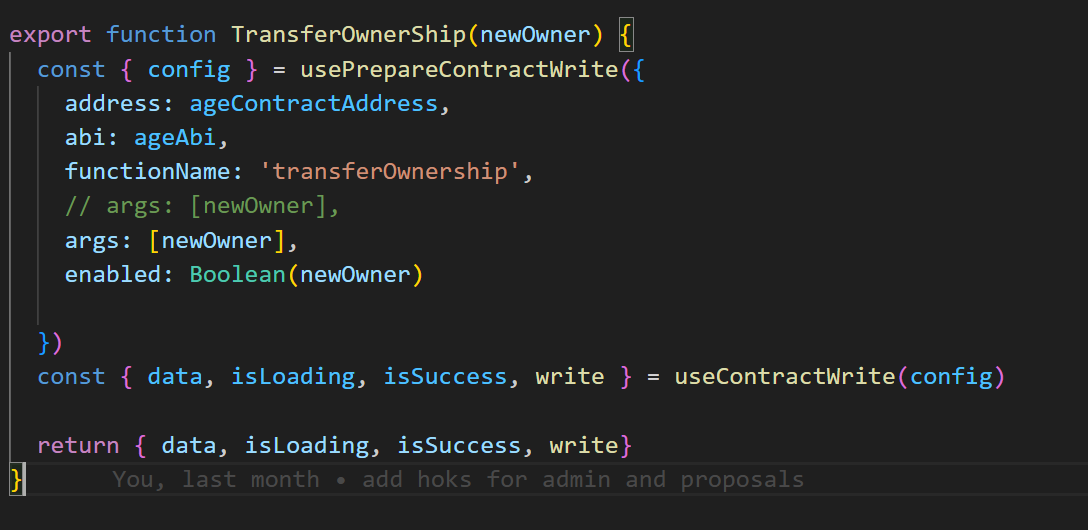
* **MintNFT.jsx** is responsible for minting NFT token for users

****

* **Queue.jsx** is responsible for queue proposals

****

* **TransferOwnership.jsx** is for changing owner of the smart contract.

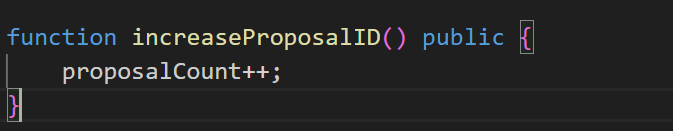
****

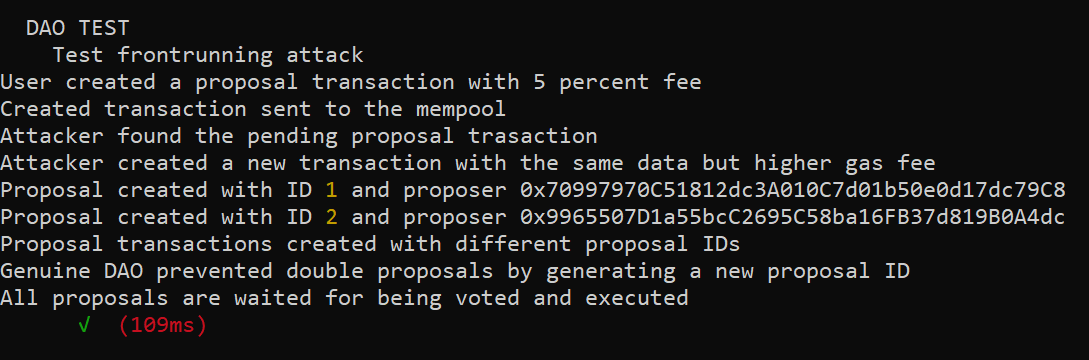
**Front Running Attack check:**

Front-running is a practice that occurs in financial markets where someone with knowledge of future transactions takes advantage of that information to make a profit. This can also happen in blockchain systems like Ethereum, where transactions are published before execution. Attackers can launch front-running attacks by inserting additional transactions before upcoming victim transactions to manipulate victim transaction executions and make profits. It is the vulnerabilities in smart contracts, which are blockchain programs invoked by transactions, that enable the front-running attack opportunities.

In our Dao smart contract we implement a solution to prevent frontrunner to create a proposal with the same data and Id.

When creating a new governance proposal, the proposalId is generated by an incremental proposal Id that happens in smart contract , To prevent duplicated proposals, the current Governor implementation checks if the proposalId exists already. If it exists, new proposal that has created by frontrunner won’t be the same with the pending proposal Id, so Frontrunner wont be able to create a proposal with the same data and Id.

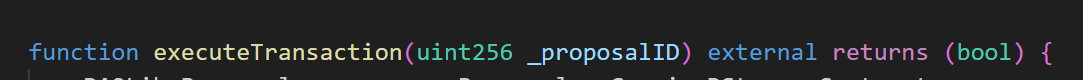




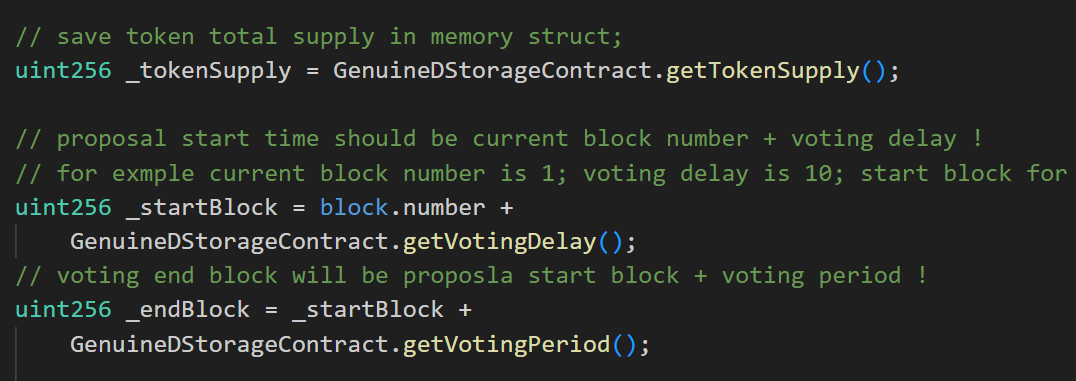
**Gas optimization:**

In This project we used some techniques to reduce the gas cost of transactions:

* Using **external** level of visibility helps us to reduce gas because **public** level is for functions who can called inside or outside of smart contract, but we need some functions just to be called from outside of smart contract so we just set visibility to **external** and reduce the cost of the transactions.



* **Defining state variables** is very efficient when we want to read smart contract storage more than one time. Because read the smart contract data has cost in a transaction. So we store them as a local variable and we don’t have to pay gas for multiple call.



* **Packing variables in a struct** is an efficient technique to reduce the storage of smart contract for saving data. So we pay lower gas for read and write data in smart contract and also deploying has lower cost.

