**CODELANDCS BLOCKCHAIN DEVELOPMENT SYLLABUS**

**WEEK 8**

**DAY 1**

**FRONTEND BLOCKCHAIN DEVELOPMENT**

Frontend development in smart contract using HTML and JavaScript can be achieved through the use of **web3.js** or even **ethers.js**, which are JavaScript libraries that allows developers to interact with the Ethereum blockchain. These libraries provides a set of APIs that allow developers to read and write data to the blockchain, as well as interact with smart contracts.

**THE BROWSER WINDOW OBJECT**

The window object which is a global object of the browser has

an **ethereum object** attached to it only if a digital wallet like

metamask is installed on the browser. To connect our

website/browser to the digitial wallet, we have to ascertain

whether the object is defined or not **(whether it is installed)**. If

defined, call the **request method** on the ethereum object. This

method equally takes in an object as an argument. A method

propety and then *“eth\_requestAccounts“* as value.

**GETTING STARTED**

* Create a file named “constants” that would be responsible for holding our Contract Address and Contract ABI.
* We would call functions manually from the ethers.js library so all we have to do is head over to <https://docs.ethers.org/v5/getting-started/>, scroll down to the web browser section and copy the ethers.js library. Create a file in our directory that would hold ethers.js library.
* We would create a basic ***index.html*** file that would be displayed on to our browser for users to interact with.
* Next is the ***index.js*** which would be responsible for interacting with our contract defined by the Contract Address and Contract ABI.
* Import the ethers.js library into the index.js file and start communicating with the blockchain.

**COMMUNICATING WITH THE ETHEREUM**

**BLOCKCHAIN FROM THE FRONTEND**

To effectively interact with the ethereum blockchain, we have to

use the ethersJS library which is the complete and compact

library for communicating with an EVM based blockchain.

To send a transaction to a blockchain, we always need three

things.

- **A provider: -** This is an endpoint to a node that is connected

to a blockchain.

- **Signer: -** This is a wallet with some ETH for paying gas fee

(Transaction Fees).

- **A Contract: -** A byte code that is already deployed to the

EVM. As aforementioned, to interact with a contract, we need

the **Contract Address, the ABI** and **a provider.** The provider

would get an instance of a web3 provider together with our signer,

for sigining transactions. When all these are combined together, a

new contract is created using the ***ethers.Contract*** method. And

now, we can access all the functions contained in the contract

(interact with the contract).

**LISTENING FOR A TRANSACTION MINE**

A function which takes in the **transaction response** and a

**provider** as an argument and equally returns a promise would be

used to call the ***once*** event on the provider. It accepts the

**txResponse hash** and a **callback function** to furnish us with the

tx receipt/confirmations.

**INTRODUCTION TO REACT MORALIS**

React Moralis is a development framework that provides developers with a quick and easy way to create decentralized applications (dApps) that interact with smart contracts. It is built on top of React, one of the most popular frontend development frameworks, and Moralis, a backend-as-a-service provider that abstracts away many of the complexities of blockchain development.

So, why should we use React Moralis for smart contract development? Let's explore some of the benefits.

Firstly, React Moralis provides developers with a simple and intuitive API that allows them to interact with smart contracts **without needing to write complex code.** The framework handles many of the low-level details, such as **ABI parsing** and **contract address management**, so developers can focus on building their dApps.

Secondly, React Moralis **provides a high level of security** for smart contract development. It leverages Moralis' secure backend infrastructure to ensure that all interactions with the blockchain network are authenticated and encrypted. This ensures that your smart contracts and dApps are secure and protected from potential attacks.

Thirdly, React Moralis **provides developers with a range of tools** and **libraries** that make it easy to build complex dApps. For example, the framework includes a range of pre-built components, such as authentication and payment systems, that can be easily integrated into your dApp. This saves time and effort in building your own components from scratch.

In terms of practical applications, React Moralis can be used for a variety of use cases, such as building decentralized finance (DeFi) applications, non-fungible token (NFT) marketplaces, and gaming platforms. The framework also supports multiple blockchain networks, such as Ethereum and Binance Smart Chain, making it a versatile tool for smart contract development.

**USING THE MORALIS PACKAGE**

First, we have to install the **moralis-v1** and **react-moralis** packages. Install an additional package called **web3uikit.**

Next, import the **MoralisProvider** in the Parent Component (App.js) from the react-moralis package that would be used to wrap our entire application.

Our frontend application would be wrapped with the **MoralisProvider**and the ***initializeOnMount*** prop set to false.

We import the ***useMoralis*** hook from the react-moralis package and this hook would present to us all that is needed to work with moralis in our desired component.

**The enableWeb3** - This is an asynchronous function gotten from the useMoralis hook. It is the equivalent of calling the request method on the window ethereum object back in plain **HTML.**

**The account -** it maybe that web3 is enabled but not connected to an account. The account object is used to display the account connected to.

**USEEFFECT**

Anytime we run **enableWeb3,** a boolean called ***isWeb3Enabled*** would be triggered to become true. It is equally passed into the useEffect dependency array to be checking when it becomes false.

**LOCALSTORAGE**

After calling the **enableWeb3** function, we need to store in the local storage to enable our application remember that we recently connected. So if i***sWeb3Enabled*** returned true, then the function would return. But if i***sWeb3Enabled*** returned false, then the useEffect hook would check if a window exists and equally checks the local storage for the particular string that is stored there. And if it is stored there, we go ahead and call the ***enableWeb3*** to actually connect.

**DISCONNECTING**

Another ***useEffect*** hook would be used to disconnect our digital wallet. We need to import the ***Moralis*** object and call the ***onAccountChanged*** function on it. This function takes in a ***callback function*** as an argument and this callback function takes in an account object as a parameter.

If the account object is null, we can assume that the user has disconnected his wallet, so we can now remove the string from local storage and called the ***deactivateWeb3*** function from the ***useMoralis*** hook which would set ***isWeb3Enabled*** to false.

Tip: Disable the connect button using the ***isWeb3EnabledLoading*** boolean.

**CALLING FUNCTIONS FROM THE FRONTEND**

The ***useWeb3Contract*** hook gotten from the react-moralis package would enable us to do basically anything with respect to interacting with a smart contract.

**THE CONSTANTS FOLDER APPROACH**

This approach is making the frontend application **“network agnostic”** so that it would work the same way the backend code would irrespective of the network we deploy to or the modification we might have made to the contract **that leads to recompilation**.

* Spin up a localhost node to act as a blockchain we would be connecting to.
* Run the update frontend script in the deploy folder that would be responsible to **update contract address** and **ABI** of a contract.
* Check if update frontend environment variable is set to true and run the deploy script.
* In updating the ABI, we need to first get hold of the contract and then write it (using the file system module) to the desired path. We can get the ABI of a contract by accessing the ***interface property*** and then calling the ***format*** method on the interface property. The method would take in the ***ethers.utils.FormatTypes.json.***
* Coming to updating the addresses, we equally need to get hold of the contract, the chainId and lastly, the currentAddresses (which is an JSON object parsed to a Javascript object)
* We check if the chainId of the network we are currently on is in the **currentAddresses object**. If so, we go ahead and check if the chainId (which has a value an array) includes the current contract address; if not, we add the current contract address into the array.
* If the chainId **does not exist inside the currentAddresses object**, we create one,  and assign an array as a value which would contain the specific contract address and finally, we write to the relative path to update contract addresses.

**USEWEB3CONTRACT HOOK**

The hook is gotten from the **react-moralis package**. It is used to perform on-chain functions. It **takes in an object as an argument** which includes the following properties

* contractAddress
* Abi
* The function name
* Message value
* Any parameters

**ChainId is used chiefly for identifying networks**. It is assessed easily when working with moralis because of the MoralisProvider (when an external wallet is connected, the **provider access the network and obtains the chainId of the network)**. A chainId string is to be imported from the **useMoralis** hook. Convert from the hex format to a plain string with the javascript **parseInt** function.

Note when to use react hooks like the useEffect and useState hook and also to format the result gotten from the call (ethers.utils.formatUnits).

**CALLING ON-FUNCTIONS**

The **useWeb3Contract** is still useful for calling on-chain functions. Equally, we need a way to notify users when their function call is a success or an error. For this, we need the **web3UIkit**. We would import the **NotificationProvider** and wrap the entire application with it.

Next, import the **useNotification hook** which when invoked returns a dispatch or notification function.

It is **pertinent** to note that the **on-chain functions come with some parameters (an object parameter with the onSuccess, onError properties e.t.c).** When a function call is successful, we go ahead and call a ***handleSuccess*** function (wait for block confirmation and call the next function that would trigger a notification to the user).

**READING AND DISPLAYING CONTRACT DATA**

* Make the on-chain function calls a stand alone function
* Re-call the function as soon as the notification function finishes running.