Fullstack dApp Integration

Learn how to write a full stack decentralized React application utilizing a Secret smart contract and Secret.js

Millionaire Problem Decentralized Application

Yao's Millionaires' problem is a secure multi-party computation problem introduced in 1982 by computer scientist and computational theorist Andrew Yao. The problem discusses two millionaires, Alice and Bob, who are interested in knowing which of them is richer without revealing their actual wealth.

The Secret smart contract we will be working with demonstrates an example implementation that allows two millionaires to submit their net worth and determine who is richer, without revealing their actual net worth.

This is the source code for the full stack application and you can use the live dApp at this web address on Secret testnet.

To interact with the dApp, you will need to have the Secret Testnet (pulsar-3) configured with your Keplr wallet and also fund it with testnet tokens. Learn how to configure and fund your keplr wallet here! In this demo you will learn how tointegrate the Secret Millionaire contract with a front end designed in React using Secret.Js. Let's get started!

Completed React.js application

Frontend library overview

This tutorial assumes that you've already learned how tocompile, upload, and instantiate a Secret smart contract to the Secret testnet and now you will learn how to connect your instantiated smart contract to a frontend library, namely, React.js.

Excluding the instantiation message, the Secret Millionaire smart contract is capable ofexecuting three messages :

1. Submit net worth
2. Reset net worth
3. Query net worth
4.

Thus, we need to design our frontend in an intuitive manner that will allow the user to execute these messages. By the end of this tutorial you will learn how to do the following:

1. Integrate Keplr wallet with React.js
2. Use secret.js to execute multiple transactions simultaneously (submit and reset net worth)
3. Use secret.js to query the updated Secret Millionaire smart contract
4.

Integrating Keplr Wallet with Secret.js

In order to execute the Secret millionaire contract, users must first be able to connect to their Keplr wallet. To see a generalized approach to connecting to Keplr wallet with Secret.js, you canreview the Secret.js docs here . However, for our application you will learn how toconnect to Keplr with React.js so that the user's wallet can control every page of your app.

Connect Wallet function

Before we proceed, review theConnectWallet() function:

```

Copy asyncfunctionconnectWallet() { try{ if(!window.:) { console.log("install keplr!"); }else{ awaitsetupKeplr(setSecretjs,setSecretAddress); localStorage.setItem("keplrAutoConnect","true"); console.log(secretAddress); } }catch(error) { alert( "An error occurred while connecting to the wallet. Please try again." ); } }

```

This function awaitsSetupKeplr() , an asynchronous function which enables Secret Network on Keplr, retrieves the user's account information from Keplr, creates a Secret Network client with this information, and then sets thesecretjs instance andsecretAddress with these details so that we can then share this data with the rest of our application. Notice that when we establish the Secret Network client with Secret.js we are also specifying theurl andchainId of the client, which in this case is theurl +chainId for Secret testnet:

```

Copy constchainID="pulsar-3", consturl="https://api.pulsar.scrttestnet.com",

constaccounts=awaitkeplrOfflineSigner.getAccounts();
```

constsecretAddress=accounts[0].address;

constsecretjs=newSecretNetworkClient({ url:url, chainId:chainId, wallet:keplrOfflineSigner, walletAddress:secretAddress, encryptionUtils:window.getEnigmaUtils(SECRET_CHAIN_ID), });

setSecretAddress(secretAddress); setSecretjs(secretjs); }

```

In fewer than 100 lines of code, you have the functionality to connect and disconnect a Keplr wallet and can now use this data in every other part of any dApp you choose to create.

See the completed file here. CreateContext() is the React.js hook that allows us to share the Secret Network client (aka the user's wallet address) throughout the entire application. Now that a user can connect to our front end, let's write some functions with Secret.js thatexecute the submit_net_worth() and reset_net_worth() functions.

Writing the Execution Messages

The Secret Millionaire contract is designed such that when a millionaire's net worth is submitted, the contract saves the first two entries and then returns an error for subsequent attempts until a reset. The reset operation resets the contract's state back to its initial condition, where no millionaires are registered.

Because the Secret smart contract requires two millionaires to be submitted at any given time, it would be an improved UI experience if the user could submit two millionaires simultaneously, rather than having to execute two separate transactions. We are able to implement this functionality with Secret.js usingMsgExecuteContract and thebroadcast method:

```

Copy letsubmit_net_worth=async(millionaire1,millionaire2)=>{ constmillionaire1_tx=newMsgExecuteContract({ sender:secretAddress, contract_address:contractAddress, msg:{ submit_net_worth:{ name:millionaire1.name, worth:parseInt(millionaire1.networth), }, }, code_hash:contractCodeHash, });

constmillionaire2_tx=newMsgExecuteContract({ sender:secretAddress, contract_address:contractAddress, msg:{ submit_net_worth:{ name:millionaire2.name, worth:parseInt(millionaire2.networth), }, }, code_hash:contractCodeHash, }); consttxs=awaitsecretjs.tx.broadcast([millionaire1_tx,millionaire2_tx], { gasLimit:300_000, }); console.log(txs); };

```

This function takes in two objects, each representing a millionaire, and sends their name and net worth to the Millionaire smart contract using thebroadcast method of thesecretjs library. This method accepts an array of transactions to send to the blockchain and an options object. In this case, the options object specifies a gas limit of 300,000, which is the maximum amount of computational work the transactions are allowed to perform before they are halted. Now let's use React.js to fire this function every time a user clicks on a button.

Creating onClick events for Secret.js transactions

ThehandleSubmit function is a handler for a form submit event in our React application. This function takes user inputs (names and net worths of two millionaires), submits this data to the blockchain, queries the richer millionaire, and updates the UI accordingly:

```

Copy consthandleSubmit=async(e)=>{ e.preventDefault(); try{ setMillionaire1({ name:name1,networth:networth1 }); setMillionaire2({ name:name2,networth:networth2 });

// Call submit_net_worth with the updated values awaitsubmit_net_worth({ name:name1,networth:networth1 }, { name:name2,networth:networth2 }); // let myQuery = []; awaitquery_net_worth(myQuery);

setRicherModalOpen(true); setShowRicherButton(false); }catch(error) { alert("Please approve the transaction in keplr."); } };

```

1. e.preventDefault();
2. : This line stops the default form submission event in a web page (which would typically reload the page).
3. setMillionaire1({ name: name1, networth: networth1 });
4. andsetMillionaire2({ name: name2, networth: networth2 });
5. : These two lines update the states ofMillionaire1
6. andMillionaire2
7. respectively.setMillionaire1
8. andsetMillionaire2
9. are state-setting functions from the useState React hook.
```

10. await submit_net_worth({ name: name1, networth: networth1 }, { name: name2, networth: networth2 });
11. : This line calls thesubmit_net_worth
12. function described above, passing two objects containing the names and net worths of two millionaires. It then waits for the function to finish.
13. await query_net_worth(myQuery);
14. : This line calls thequery_net_worth
15. function and waits for the function to finish.
16. setRicherModalOpen(true);
17. andsetShowRicherButton(false);
18. : These two lines update the states controlling whether a modal and a button are displayed in the UI. They hide a button and show a modal that presents the result of thequery_net_worth
19. function.
20. alert("Please approve the transaction in keplr.");
21. : If any error is thrown during the execution of the function, it is caught, and an alert is shown to the user instructing them to approve the transaction in Keplr.
22.

Resetting the smart contract state

resetSubmit resets the smart contract state back to 0:

```

Copy constresetSubmit=async(e)=>{ e.preventDefault(); try{ awaitreset_net_worth(); setResetModalOpen(true); setShowRicherButton(true); }catch(error) { alert("Please connect your wallet by selecting the wallet icon."); } };

```

Writing the Query Message

Lastly, we need to be able to query the result of submitted transaction, namely, which millionaire is richer. This function queries the Millionaire contract to get information about who is richer among the previously submitted millionaires, stores the result in themyQuery array, and then we can display information stored inmyQuery in our front end.

```

Copy letquery_net_worth=async(myQuery)=>{ letquery=awaitsecretjs.query.compute.queryContract({ contract_address:contractAddress, query:{ who_is_richer:{}, }, code_hash:contractCodeHash, });

myQuery.push(query); };

```

1. First, we use thequeryContract
2. method of thecompute
3. module fromsecretjs
4. library to send a query to the Millionaire contract. This method takes an object as its argument with the following properties:
5.
    - contract_address
6.
    - : The address of the smart contract to which the query is being sent.
7.
    - query
8.
    - : The query message to be sent to the contract. In this case, it's calling thewho_is_richer
9.
    - method of the contract, which returns information about the richer of the two millionaires.
10.
    - code_hash
11.
    - : The code hash of the contract to ensure that the contract code hasn't been tampered with.
12. *
13. The result of this query is stored in thequery
14. variable.
15. The result of the query is then pushed onto themyQuery
16. array. This array stores the results of all queries made so far, and we can display this on our React front end.
17.

Putting it all together

You now have all of the tools you need to create a decentralized full stack application on Secret Network. In this tutorial you learned how to write React.js functions to connect to Keplr wallet, submit simultaneous transactions and reset net worth data, and query a Secret smart contract for the wealthier party. If you have further questions as you continue to develop on Secret Network,please join the weekly Monday developer call on Discord at 12pmET.

Last updated1 month ago On this page *Millionaire Problem Decentralized Application * Frontend library overview * Integrating Keplr Wallet with Secret.js * Writing the Execution Messages * Writing the Query Message * Putting it all together

Was this helpful? Edit on GitHub Export as PDF