Best Practices for Ethereum developers on NEAR

In this example, we will create an Ethereum dApp on NEAR that functions as a portfolio manager, displaying the current balances for a list of tokens. Additionally, we will display current market value of each asset in the portfolio.

We will be using several technologies:

- NEAR Components for the user interface (UI).
- Ethers.js
- for retrieving balance data from the blockchain.
- CoinGecko API for fetching static content with information about tokens and their current prices.
- Social-DB
- for storing the list of tokens to be tracked.
- GitHub Actions for caching static content, speeding up loading, and circumventing rate limits.

Step 1: Load balances from chain

Let's start with a simple example and consider an application where we want to display a user's balances for multiple tokens.

Source code

// Load current sender address if it was not loaded yet if

(state . sender

==

undefined

&&

Ethers . provider ( ))

{ Ethers . provider ( ) . send ("eth_requestAccounts" ,

[ ]) . then ((accounts)

=>

{ if

(accounts . length)

{ // save sender address to the state State . update ({

sender : accounts [0]

}) ; } }) ; }

// Load ERC20 ABI JSON const erc20Abi =

fetch ("https://ipfs.near.social/ipfs/bafkreifgw34kutqcnusv4yyv7gjscshc5jhrzw7up7pdabsuoxfhlnckrq") ; if

(! erc20Abi . ok)

{ return

"Loading" ; }

// Create contract interface const iface =

new

ethers . utils . Interface (erc20Abi . body) ;

// specify list of tokens const tokens =

["0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599" ,

// WBTC "0x6b175474e89094c44da98b954eedeac495271d0f" ,

```
// DAI "0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984" ,
// UNI ] ;
// load receiver's balance for a giver token const
getTokenBalance
=
( receiver , tokenId )
=>
{ // encode balanceOf request const encodedData = iface . encodeFunctionData ( "balanceOf" ,
[ receiver ] ) ;
// send request to the network return
Ethers . provider ( ) . call ( { to : tokenId , data : encodedData , } ) . then ( ( rawBalance )
=>
{ // decode response const receiverBalanceHex = iface . decodeFunctionResult ( "balanceOf" , rawBalance ) ;
return
Big ( receiverBalanceHex ) . toFixed ( 0 ) ; } ) ; } ;
const
loadTokensData
=
( )
=>
{ // load balances of all tokens tokens . map ( ( tokenId )
=>
{ getTokenBalance ( state . sender , tokenId ) . then ( ( value )
=>
{ // save balance of every token to the state State . update ( {
[ tokenId ] :
{
balance : value ,
... state [ tokenId ]
}
} ) ; } ) ; } ) ; } ;
const
renderToken
=
( tokenId )
=>
( < li
```

{ tokenId } :

{ state [tokenId] . balance } < / li

) ;

if

(state . sender)

{ loadTokensData ( ) ;

return

(<

< ul

{ tokens . map ((tokenId)

=>

renderToken (tokenId)) } < / ul

< p

Your account :

{ state . sender }

< / p

< /

) ; }

else

{ // output connect button for anon user return

< Web3Connect

/

; } You can see how it works herestep_1 .

Once the web3 connection is enabled, the output will appear as follows:

0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599: 726220 0x6b175474e89094c44da98b954eedeac495271d0f: 140325040242585301886 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984: 127732731780832810 tip When developing NEAR components, it's recommended to always present some content even if the user hasn't connected their wallet yet. In this example, the component uses the button to prompt the user to connect their wallet if they haven't already.

Step 2: Load static data

To format the list, we must determine the decimal precision for each asset. While it's possible to retrieve this information from the ERC-20 contract for each token, it's important to note that the ERC-20 contract lacks certain valuable data such as the token icon and description. As a solution, we can leverage the CoinGecko API to retrieve token details, including the current market price.

Let's add a function to load token data for a given token from the Coingecko:

const

loadCoingeckData

=

(tokenId)

=>

{ let dataUrl =

https://api.coingecko.com/api/v3/coins/ethereum/contract/ { tokenId } ;

const data =

fetch (dataUrl) ; if

(data . ok)

{ return

{ name : data . body . name , icon : data . body . image . small , decimals : data . body . detail_platforms ["ethereum"] . decimal_place , price :

Number (data . body . market_data . current_price . usd) , } ; } } ; Other available API methods are listed in the Coingecko API documentation .

Now that we have the data, let's modify the loadTokensData function to save the token information in the state:

const

loadTokensData

=

( )

=>

{ // load balances of all tokens tokens . map ((tokenId)

=>

{ getTokenBalance (state . sender , tokenId) . then ((value)

=>

{ // save balance of every token to the state State . update ({

[tokenId] :

{

balance : value ,

... state [tokenId]

}

}) ; }) ; }) ;

tokens . map ((tokenId)

=>

{ const tokenData =

loadCoingeckData (tokenId) ; // save balance of every token to the state State . update ({

[tokenId] :

{

... tokenData ,

... state [tokenId]

}

}) ; }) ; } ; And lets update therenderToken function to display data we just got:

const

renderToken

=

(tokenId)

=>

{ const tokenBalance =

Big (state [tokenId] . balance

??

0) . div (new

Big (10) . pow (state [tokenId] . decimals

??

1)) . toFixed (4) ; const tokenBalanceUSD =

(tokenBalance * state [tokenId] . price) . toFixed (2) ; return

(< li

{ state [tokenId] . name } :

{ tokenBalance } { " " } < img src = { state [tokenId] . icon } width = "16" alt = { state [tokenId] . symbol }

/

{ ({ tokenBalanceUSD } USD) } < / li

) ; } ; You can see how it works herestep_2 .

Output will be like this:

Wrapped Bitcoin: 0.0073 wbtc (247.64 USD) Dai: 140.3250 dai (140.21 USD) Uniswap: 0.1277 uni (0.54 USD) info Please note that thefetch function caches data and will be executed only once during loading. tip Utilize any available web-services to provide data for your application on NEAR, ensuring that the user experience is on par with traditional web 2.0 applications.

Step 3. Save data in social-db

Now, instead of hardcoding the list of tokens directly within the application code, let's transition them to an onchain data repository named social-db. This approach allows us to adjust the list of trackable tokens without having to modify the application's code. It also offers users the flexibility to select from pre-existing token lists or formulate their own.

Learn more about howkey-value storage social-db works .

Here is an example of a simple application forsetting tokens list in social-db .

In this format, the data from the example will be stored in social-db.

{ "0x6b175474e89094c44da98b954eedeac495271d0f": "", "0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599": "", "0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984": "" } Viewing this data from the blockchain is accessible for every NEAR app or, for example, through anExplorer app .

Let's add a function to our application that will read the list of tokens.

// set list of tokens if

(! state . tokensLoaded)

{ // load tokens list from the Social DB const tokens =

Social . get (zavodil.near/tokens-db/* ,

"final") ;

if

(tokens)

{ State . update ({ tokensLoaded :

true , tokens , }) ; } }

const tokens =

Object . keys (state . tokens

??

{ }) ; info In this case,zavodil.near is the NEAR account of the user who created this list of tokens. Any other user can create their own list, and it will also be available insocial-db . You can see how it works here:step_3 . The output of the data in the application remains unchanged, but now it no longer contains hardcoded values.

tip Use social-db, an on-chain data storage, to decouple the data and the application.

Step 4. Caching Data Through GitHub Actions

Ethereum-based applications frequently depend on static content sources to present details about tokens or contracts. Often, frontends pull this data from platforms like CoinGecko or CoinMarketCap, leveraging API keys to enhance the data retrieval rate limit. Without these API keys, and given a significant volume of data, fetching from these platforms can be sluggish or even disrupted. We'll showcase a serverless approach utilizing GitHub Actions. This method preserves the decentralized nature of NEAR gateways (where securely storing API keys isn't feasible), all while ensuring user ease-of-use and swift data loading.

Let's create a Node.js application that will iterate through a list of tokens fromsocial-db and display the retrieved data along with a timestamp of the operation.

import

*

as nearAPI

from

"near-api-js" ; import

*

as cg

from

"coingecko-api-v3" ;

const

CONTRACT_ID

=

"social.near" ; const

ETHEREUM_NETWORK_ID

=

"ethereum" ; const

FETCH_TIMEOUT

=

7000 ;

async

function

connect ( )

{ const config =

```
{ networkId :
"mainnet" , keyStore :
new
nearAPI . keyStores . InMemoryKeyStore ( ) , nodeUrl :
"https://rpc.mainnet.near.org" , walletUrl :
"https://wallet.mainnet.near.org" , helperUrl :
"https://helper.mainnet.near.org" , explorerUrl :
"https://nearblocks.io" , } ; const near =
await nearAPI . connect ( config ) ; const account =
await near . account ( CONTRACT_ID ) ;
const contract =
new
nearAPI . Contract ( account ,
// the account object that is connecting CONTRACT_ID ,
// name of contract you're connecting to { viewMethods :
[ "get" ] ,
// view methods do not change state but usually return a value changeMethods :
[ ] ,
// change methods modify state sender : account ,
// account object to initialize and sign transactions. } ) ;
return contract ; }
// load data from the social-db const contract =
await
connect ( ) ; const data =
await contract . get ( {
keys :
[ "zavodil.near/tokens-db/*" ]
} ) ; const tokens = data [ "zavodil.near" ] [ "tokens-db" ] ;
// init coingecko client const client =
new
cg . CoinGeckoClient ( { timeout :
5000 , autoRetry :
false , } ) ;
let res =
{ } ; for
( let i =
0 ; i <
```

Object . keys (tokens) . length ; i ++)

{ const tokenId =

Object . keys (tokens) [i] ;

try

{ // load data from coingecko const data =

await client . contract ({ id :

ETHEREUM_NETWORK_ID , contract_address : tokenId , }) ; // format output const tokenData =

{ name : data ["name"] , symbol : data ["symbol"] , icon : data ["image"] ?. ["thumb"] , decimals : data ["detail_platforms"] ?. [ETHEREUM_NETWORK_ID] ?. ["decimal_place"] , price : data ["market_data"] ?. ["current_price"] ?. ["usd"] , } ; // store output res [tokenId]

= tokenData ;

// add timeout to avoid rate limits await

new

Promise ((resolve)

=>

{ setTimeout (resolve ,

FETCH_TIMEOUT) ; }) ; }

catch

(ex)

{ console . error (tokenId , ex) } }

// output results console . log (JSON . stringify ({ timestamp :

Date . now ( ) , data : res , })) ; Example of this code on a github , you can clone the repository and modify the data retrieval request as needed.

Now, we can create a GitHub worker that will execute this script and save the data to a file namedtokens-db.json . Here are the instructions for the worker:

name : Tokens Data Updater on : workflow_dispatch : schedule : -

cron :

*'/15 * * *'*

jobs : updateStats : runs-on : ubuntu - latest steps : -

uses : actions/checkout@v3 -

name : Prepare uses : actions/setup - node@v3 with : node-version :

16

run : npm i -

name : Tokens Data run : node load

tokens

db.json env : EXPORT_MODE : CS -

uses : EndBug/add - and - commit@v9 with : author_name :

"Tokens Data Updater" add :

'tokens-db.json --force' message :

"Tokens Data fetching" Don't forget to grant the worker the necessary permissions to add files to your repository (Open GitHub Repository: Settings->Actions->General)

The output of this worker will be thetokens-db.json file which will be regularly updated with current data. You can easily add any private API keys required for bypassing rate limits in the worker.

Now, let's get back to the NEAR application. We need to modify the code to read data from the cached file created by GitHub Actions instead of fetching it from CoinGecko every time.

To do this, we'll make changes to theloadTokensData function:

```
const

loadTokensData

=

( )

=>

{ let cacheTokenData =

{ } ; // load data generated by github action const cachedData =

fetch ( https://raw.githubusercontent.com/zavodil/tokens-db/main/tokens-db.json ) ; if

( cachedData . ok )

{ const cache =

JSON . parse ( cachedData . body ) ; const cacheDate =

new

Date ( cache . timestamp ) ; const timeDifference =

Date . now ( )

- cacheDate . getTime ( ) ; if

( timeDifference <=

30

*

60

*

1000 )

{

// use cached data if it is not outdated (30 min) cacheTokenData = cache . data ; }

tokens . map ( ( tokenId )

=>

{ const tokenData = cacheTokenData . hasOwnProperty ( tokenId ) ? cacheTokenData ?. [ tokenId ] :

// load data from coingecko if we don't have cached data only loadCoingeckData ( tokenId ) ; // save balance of every token to the state State . update ( {

[ tokenId ] :

{

... tokenData ,
```

... state [tokenId]

}

}) ; }) ; } } ; You can see how it works here step_4 . The output of the data in the application remains the same, but now it operates more efficiently.

tip Use GitHub Actions as a serverless backend for securing API keys, caching data etc Edit this page Last updatedonMar 25, 2024 bygagdiez Was this page helpful? Yes No

Previous NEAR for Ethereum developers Next Lido Example