Integrating Contracts

To integrate NEAR to your frontend, you will leverage two tools:

1. Wallet Selector
2. : Enables the user to select their preferred NEAR wallet in your dApp.
3. NEAR API JS
4. : A suit of tools to interact with the NEAR RPC.

Using those tools you will implement the following flow:

1. Setup
2. a wallet selector.
3. Load the wallet selectoron start-up
4. .
5. Ask the user tosign-in
6. using a NEAR wallet.
7. Call methods
8. in the contract.

Adding NEAR API JS and Wallet Selector

In order to usenear-api-js and thewallet-selector you will need to first add them to your project.

The wallet selector has multiple wallet packages to select fromsee in their website .

npm install \ near-api-js \ @near-wallet-selector/core \ @near-wallet-selector/my-near-wallet \ @near-wallet-selector/ledger \ @near-wallet-selector/modal-ui Usingnear-api-js in plain HTML You can addnear-api-js as a script tag in your html.

< script src = "https://cdn.jsdelivr.net/npm/near-api-js@0.44.2/dist/near-api-js.min.js" integrity = "sha256-W5o4c5DRZZXMKjuL41jsaoBpE/UHMkrGvIxN9HcjNSY=" crossorigin = "anonymous"

< / script

Create a Wallet Object

In our examples we implement a/near-wallet.js module, where we abstracted thewallet selector into aWallet object to simplify using it.

To create a wallet, simply import theWallet object from the module and initialize it. Thiswallet will later allows the user to call any contract in NEAR.

- JavaScript

frontend/index.js loading ... See full example on GitHub When instantiating the wallet you can choose if you want tocreate aFunctionCall Key .

If you create the key, then your dApp will be able toautomatically sign non-payable transactions for the user on the specified contract.

Setting customs RPC endpoints If you want to use a user-defined RPC endpoint with the Wallet Selector, you need to setup anetwork options object with the custom URLs. For example:

- JavaScript

index.js const

CONTRACT_ADDRESS

= process . env . CONTRACT_NAME ;

const my_network =

{ networkId :

"my-custom-network" , nodeUrl :

"https://rpc.custom-rpc.com" , helperUrl :

"https://helper.custom-helper.com" , explorerUrl :

"https://custom-explorer.com" , indexerUrl :

"https://api.custom-indexer.com" , } ;

const wallet =

new

Wallet ({

createAccessKeyFor :

CONTRACT_ADDRESS ,

network : my_network }) ; tip You can find the entire Wallet SelectoAPI reference here .

Wallet Start Up

In our examples we always implement a simple flow where we start by checking if the user logged-in and act on it. We recommend you to do the same.

For this, override thewindow.onload method with a function that calls thewallet.startUp() method. Such method returns if the user is already signed-in:

- JavaScript
- index.js
- near-wallet.js

frontend/index.js loading ... See full example on GitHub frontend/near-wallet.js loading ... See full example on GitHub Under the hood (check thenear-wallet tab) you can see that we are actually setting up the wallet selector, and asking it if the user logged-in already.

Calling View Methods

Once the wallet is up we can start calling view methods, i.e. the methods that perform read-only operations.

Because of their read-only nature, view methods arefree to call, and donot require the user to belogged in .

- JavaScript
- index.js
- near-wallet.js

frontend/index.js loading ... See full example on GitHub frontend/near-wallet.js loading ... See full example on GitHub The snippet above shows how we call view methods in our examples. Switch to thenear-wallet tab to see under the hood: we are actually making adirect call to the RPC usingnear-api-js .

tip View methods have by default 200 TGAS for execution

User Sign-In / Sign-Out

In order to interact with non-view methods it is necessary for the user to first sign in using a NEAR wallet.

Signing in is as simple as requesting thewallet object tosignIn , the same simplicity applies to signing-out.

- JavaScript
- index.js
- near-wallet.js

frontend/index.js loading ... See full example on GitHub frontend/near-wallet.js loading ... See full example on GitHub When the user clicks in the button, it will be asked to select a wallet and use it to login.

Function Call Key

If you instantiated theWallet passing an account for thecreateAccessKeyFor parameter, then the wallet will create

aFunctionCall Key and store it in the web's local storage.

- JavaScript

frontend/index.js loading ... See full example on GitHub By default, such key enables to expend a maximum of0.25Ⓝ on GAS calling methods inthe specified contractwithout prompting the user to sign them.

If, on the contrary, you do not create an access key, then the user will be asked to sign every single transaction (except calls toview methods , since those are always free).

tip Please notice that this only applies fornon-payable methods, if you attach money to any call the user willalways be redirected to the wallet to confirm the transaction.

Calling Change Methods

Once the user logs-in they can start calling change methods. Programmatically, calling change methods is similar to calling view methods, only that now you can attach money to the call, and specify how much GAS you want to use.

It is important to notice that, if you ask for money to be attached in the call, then the user will be redirected to the NEAR wallet to accept the transaction.

- JavaScript
- index.js
- near-wallet.js

frontend/index.js loading ... See full example on GitHub frontend/near-wallet.js loading ... See full example on GitHub Under the hood (seenear-wallet tab) we can see that we are actually asking thewallet tosign a FunctionCall transaction for us.

tip Remember that you can use thewallet to call methods inany contract. If you did not asked for a function key to be created, the user will simply be prompt to confirm the transaction.

Wallet Redirection

If you attach money to a change call, then the user will be redirected to their wallet to accept the transaction. After accepting, the user will be brought back to your website, with the resulting transaction hash being pass as part of the url (i.e.your-website.com/?transactionHashes=...).

If the method invoked returned a result, you can use the transaction hash to retrieve the result from the network. Assuming you created thenear object as in theexample above , then you query the result by doing:

- JavaScript
- index.js
- utils.js

frontend/index.js loading ... See full example on GitHub frontend/near-wallet.js loading ... See full example on GitHub

Handling Data Types

When calling methods in a contract, or receiving results from them, you will need to correctly encode/decode parameters. For this, it is important to know how the contracts encode timestamps (u64) and money amounts (u128).

Time

The block timestamp in a smart contract is encoded using nanoseconds (i.e. 19 digits:1655373910837593990). In contrast,Date.now() from javascript returns a timestamp in milliseconds (i.e 13 digits:1655373910837). Make sure to convert between milliseconds and nanoseconds to properly handle time variables.

Money

Smart contracts speak in yocto NEAR, where 1Ⓝ = 10^24yocto, and the values are always encoded asstrings .

- Convert from NEAR to yocto before sending it to the contract usingnear-api-js.utils.format.parseNearAmount(amount.toString())
- .
- Convert a response in yoctoNEAR to NEAR usingnear-api-js.utils.format.formatNearAmount(amount)

tip If the contract returns aBalance instead of aU128 , you will get a "scientific notation"number instead of astring (e.g.10^6 instead of"1000000"). In this case, you can convert the value to NEAR by doing:

function

formatAmount (amount)

{ let formatted = amount . toLocaleString ('fullwide' ,

{

useGrouping :

false

}) formatted = utils . format . formatNearAmount (formatted) return

Math . floor (formatted *

100)

/

100 }

Leveraging NEAR API JS

NEAR API JS does not limit itself to simply calling methods in a contract. In fact, you can use it to transform your web-app into a rich user experience. While we will not cover these topics in depth, it is important for you to know that with NEAR API JS you can also:

- Sign and verify messages
- : this is very useful to prove that a message was created by the user.
- Create batch transactions
- : this enables to link multipleactions
- (e.g. multiple function calls). If one of the transactions fails, then they are all reverted.
- Create accounts
- : deploy accounts for your users!

Check thecookbook to learn how to supercharge your webapp.Edit this page Last updatedonFeb 9, 2024 bygagdiez Was this page helpful? Yes No

Previous ⭑ Quickstart a WebAppNext Integrating Components