Tutorial: how to interact with the Astroport Assembly contract using Terra.js

September 5, 2022 ✦ Technical

The Astral Assembly is Astroport's version of a DAO and its goal is the operation, maintenance, development, and growth of Astroport as a user-governed DeFi commons. xASTRO holders will have the power to propose and make binding votes on smart contract parameter changes, smart contract upgrades, and treasury disbursements.

As a smart contract deployment, the Astral Assembly does not rely on any specific front-end UI and can instead be interacted with directly. This ensures that the protocol remains decentralized despite certain centralized gateways, such as the Astroport WebApp.

For the sixth tutorial in this series, we will go over the Assembly contract using Terra.js: how to submit, vote, manage, and query proposals. The Assembly contract code can be found here.

1. Prerequisites

Node.js and npm

This tutorial uses the latest stable versions of node.js and node package manager. For more information, visit https://nodejs.org.

Terra.js

For a step-by-step walkthrough on setting up Terra.js, refer to the third tutorial in this series. You should end up with something like this:

```
// ** SET UP ** //

// required modules
const { LCDClient, MnemonicKey,
MsgExecuteContract } = require('@terra-
money/terra.js');

// connecting to terra blockchain
const terra = new LCDClient({

// testnet
URL: 'https://pisco-lcd.terra.dev',
chainID: 'pisco-1',

// mainnet
// URL: 'https://phoenix-lcd.terra.dev',
// chainID: 'phoenix-1',
});


// wallet information
const mk = new MnemonicKey({
mnemonic: ''
});

const wallet = terra.wallet(mk);
```

xASTRO Tokens

This tutorial assumes you have xASTO tokens (from depositing ASTRO tokens into the Staking contract or from directly swapping to xASTRO). xASTRO is needed to submit new proposals and to vote on proposals.

To stake ASTRO for xASTRO, refer to our fifth tutorial in this series. To directly swap for xASTRO tokens, refer to our second tutorial in this series.

2. Set up

To complete this tutorial, you will interact with 3 Astroport contracts: the Assembly testnet and mainnet contracts along with the xASTRO contract. For convenience, set up 3 variables with the addresses of each contract. Note that this tutorial uses both testnet and mainnet addresses. For a full list of Astroport contract addresses, look here.

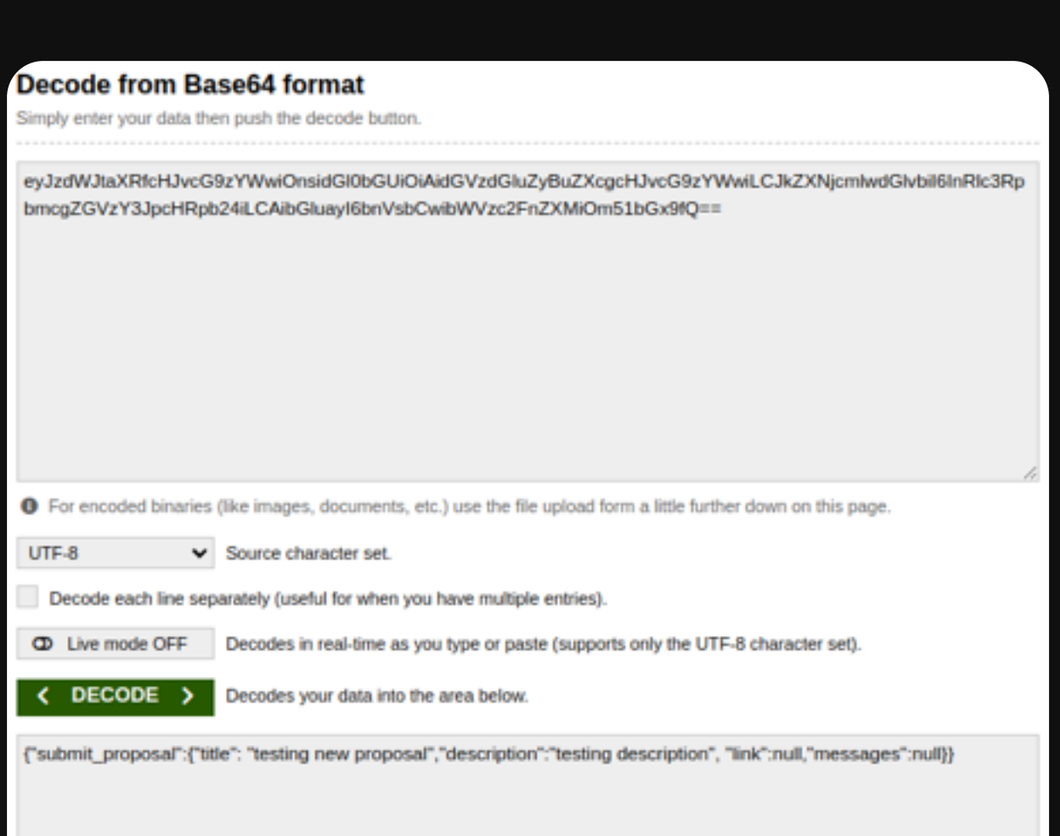
```
// xastro address (testnet)
const contract_address =
"terra1ctzthkc0nzseppqtqlwq9mjwy9gq8ht2534rtcj3y
plerm06snmqfc5ucr"

// assembly address (testnet)
const contract_address =
"terra195m6n5xq4rkjy47fn5y3s08tfmj3ryknj55jqvgq2
y55zul9myzsgy06hk"

// assembly address (mainnet)
const contract_address =
"terra1k9j8rcyk87v5jvfla2m9wp200azegjz0eshl7n2pw
v852a7ssceqsnn7pq"
```

3. Submitting Proposals

Note: Before submitting an on-chain proposal, the proposal needs to go through the Astroport forum and the Astroport Improvement Proposal (AIPs) process. For more information, visit here.

To submit an on-chain proposal, we use the `send` function in the xASTRO token contract and specify the Assembly address under the contract parameter. To clarify, we are working with the testnet xASTRO contract for our `contract_address` variable. We also need to specify the required amount of xASTRO to submit a proposal and include a Base64 encoded message (more on this next). Finally, the function is wrapped in an execute variable which contains our wallet information and the `contract_address` variable we created:

The `msg` parameter contains a `submit_proposal` message to send to the Assembly contract. You can use a Base64 encoder/decoder to complete this step:

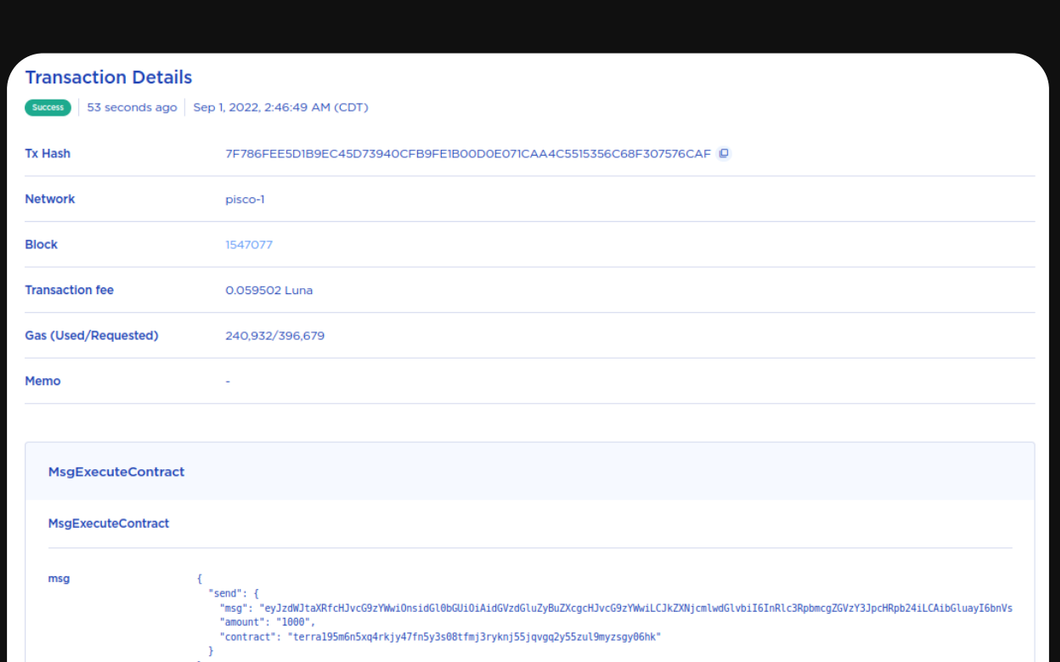


Note: the embedded `submit_proposal` itself could contain a Base64 encoding of an executable message. For example, this is useful for automatic treasury disbursements:

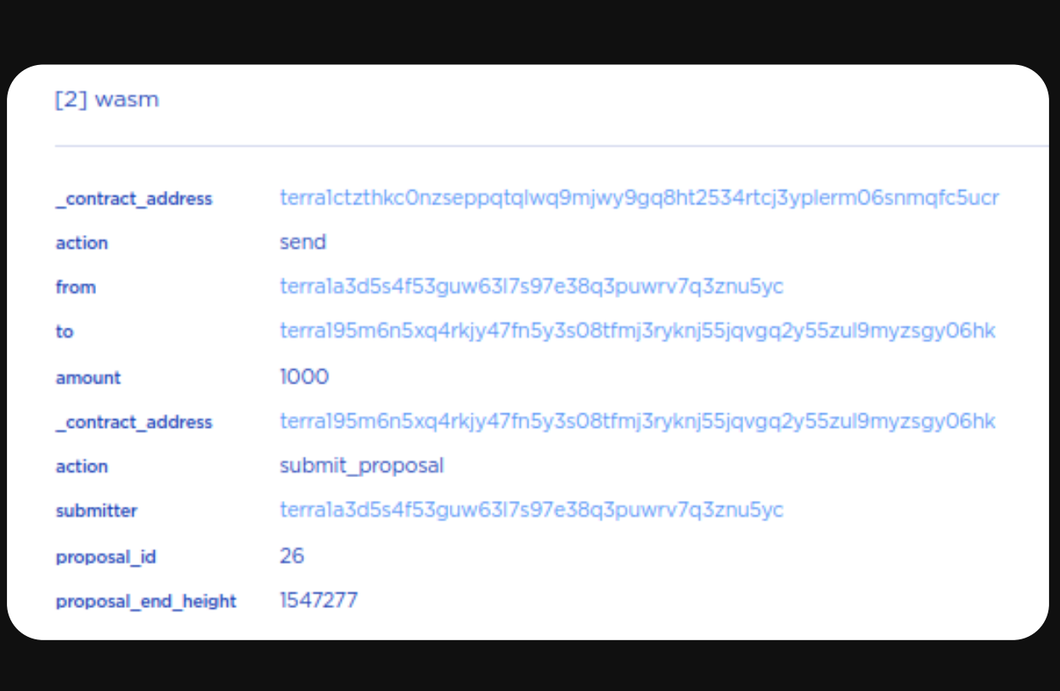
To complete the call, we need an async function that creates and signs the transaction with our wallet. The `msgs` parameter takes in the execute variable we created above which contains our wallet information, our target contract address, and the `ExecuteMsg` (`send`) to call.

We use the command line and node.js to execute the call and retrieve the transaction hash:

Congrats! You've submitted a proposal to the Astral Assembly! We can use the hash that's returned to get more information about our transaction using a Terra block explorer like terrasco.pe (for mainnet) and finder.terra.money:



We can select "Show Logs" to expand the transaction details. The "wasm" section will display the submission of our proposal to the Assembly contract and the corresponding `proposal_id`:



4. Voting on Proposals

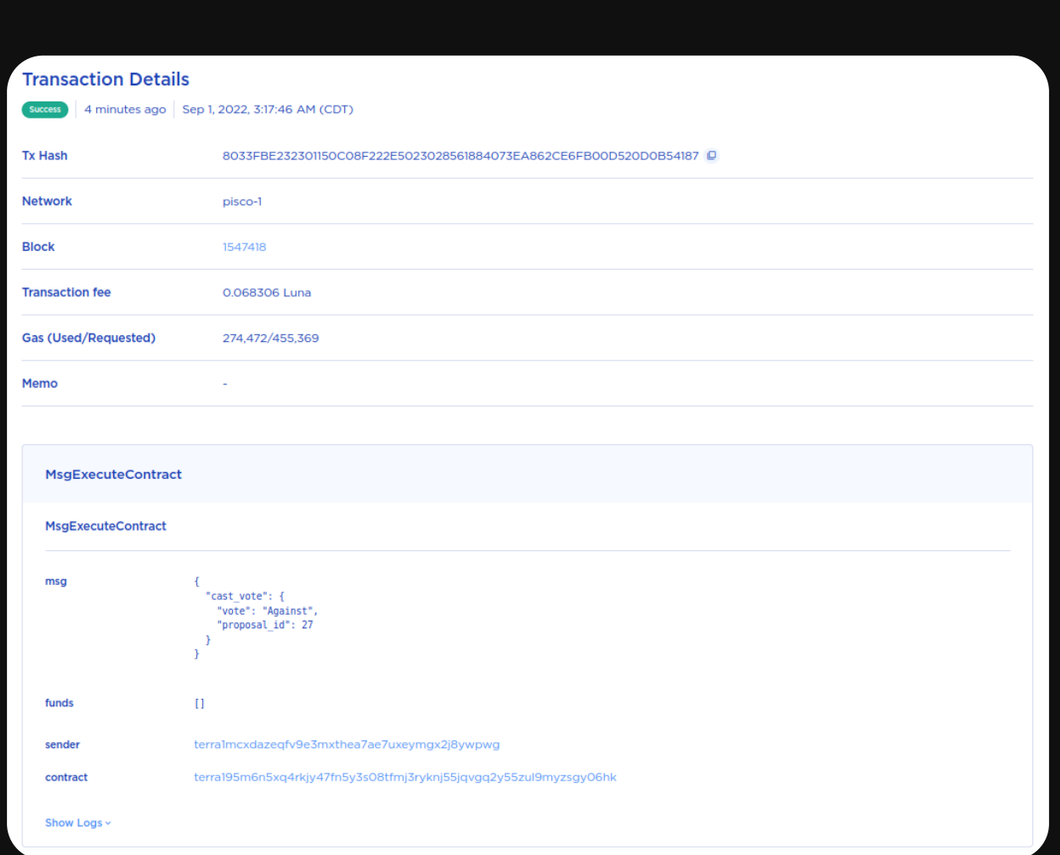
Note: For testing proposals, you will need a separate wallet / mnemonic keys since the proposal submitter cannot vote on their own proposal.

For the next two sections in this article, we will be working with the testnet Assembly address and will need to change our `contract_address` variable to correspond to this.

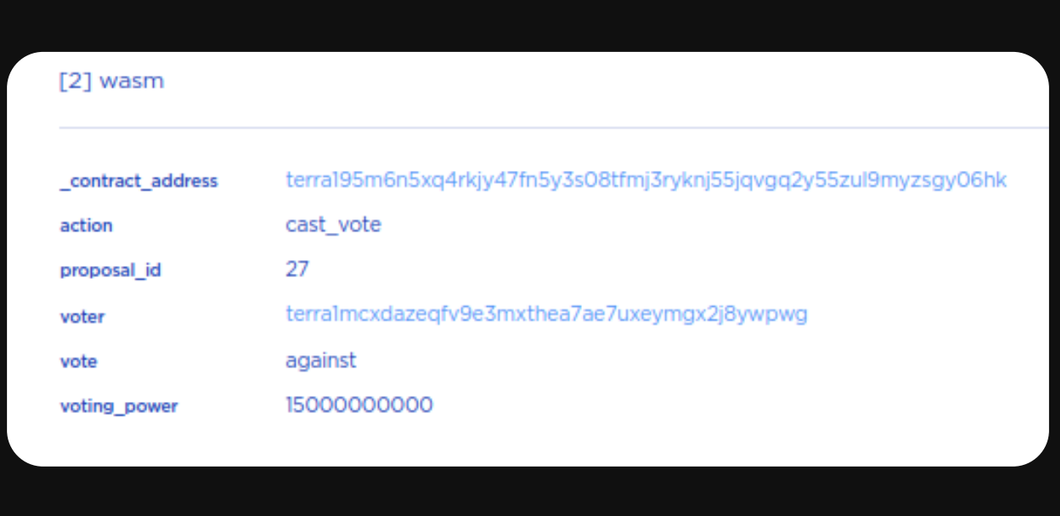
To vote on an on-chain proposal, we use the `cast_vote` function in the Assembly contract and specify the `proposal_id` along with your `vote`. Make sure that your vote is capitalized (`For` or `Against`). The function is wrapped in an execute variable which contains our wallet information and the `contract_address` variable we created. To complete the call, we reuse the above async function to create and sign the transaction with our wallet:

We use the command line and node.js to execute the call and retrieve the transaction hash:

Congrats! You've voted on a proposal and are now part of the Astral Assembly! We can use the hash that's returned to get more information about our transaction using a Terra block explorer like terrasco.pe (for mainnet) and finder.terra.money:



We can select "Show Logs" to expand the transaction details. The "wasm" section will display the submission of our `vote` and our `voting_power`:



5. Managing Proposals

Once an on-chain proposal has gone live and the voting period has ended, there are several functions that may come in handy. This section will briefly cover them, but the complete process is similar to the examples in Section 3 and 4 of this article.

end_proposal

Ends proposal voting and sets the proposal status.

execute_proposal

Executes a successful proposal after the queue state passes. This processes all executable messages in a proposal or simply marks the proposal as executed if there are no executable messages.

remove_completed_proposal

Removes an executed, expired, or rejected proposal from the general proposal list.

6. Querying Proposals

Note: we will be using the mainnet Assembly contract address to query data for the Assembly contract. Make sure you change your `contract_address` variable as well as your LCDC client to correspond with the mainnet.

Key query messages for the Assembly contract are described below. For a complete list of all queries, look here.

A custom struct is defined for each query response. Queries are executed in the command line using node.js:

config

Returns Astral Assembly parameters:

Example Response:

proposal

Returns information about a specific proposal:

Example Response:

proposal_votes

Returns information about the votes cast on a proposal:

Example Response:

user_voting_power

Returns user voting power for a specific proposal:

Example Response:

7. Congrats! You have completed the tutorial on the Astroport Assembly contract!

If you are looking for an exercise, try building your own frontend dashboard that keeps track of all proposals, votes, and voter information.

For more information regarding the Astral Assembly contract, visit the Astroport docs.

✦

Follow Astroport on Twitter and subscribe to the Astroport email newsletter to get the latest alerts from the mothership.