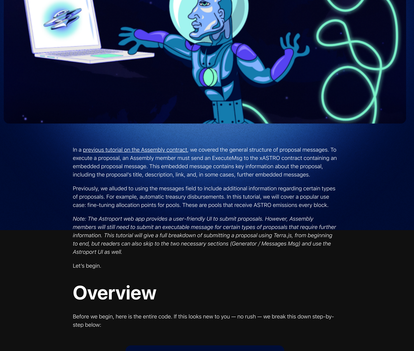
Tutorial: Structuring executable messages for Assembly proposals, Part 1: Fine-tuning allocation points

October 28, 2022 • Technical



In a previous tutorial on the Assembly contract, we covered the general structure of proposal messages. To execute a proposal, an Assembly member must send an ExecuteMsg to the xASTRO contract containing an embedded proposal message. This embedded message contains key information about the proposal, including the proposal's title, description, link, and, in some cases, further embedded messages.

Previously, we alluded to using the messages field to include additional information regarding certain types of proposals. For example, automatic treasury disbursements. In this tutorial, we will cover a popular use case: fine-tuning allocation points for pools. These are pools that receive ASTRO emissions every block.

Note: The Astroport web app provides a user-friendly UI to submit proposals. However, Assembly members will still need to submit an executable message for certain types of proposals that require further information. This tutorial will give a full breakdown of submitting a proposal using Terra.js, from beginning to end, but readers can also skip to the two necessary sections (Generator / Messages Msg) and use the Astroport UI as well.

Let's begin.

Overview

Before we begin, here is the entire code. If this looks new to you — no rush — we break this down step-by-step below:

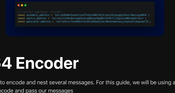
Step-by-Step

Set up

First, we use terra.js to connect to the Terra blockchain, connect a wallet, and sign and confirm transactions. Note: This guide uses the pisco-1 testnet. For a complete guide to setting up Terra.js, visit here.

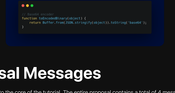
Contract Addresses

Second, it's generally a good idea to list all contract addresses that we will be using later in our code. For fine-tuning pool allocation points, a proposal will interact with 3 contracts: the Assembly, Generator, and xASTRO contracts. For a full list of mainnet and testnet addresses, visit here.



Base64 Encoder

Third, we will need to encode and nest several messages. For this guide, we will be using a custom function that will encode and pass our messages



Proposal Messages

Now we finally get to the core of the tutorial. The entire proposal contains a total of 4 messages:

1. **Generator Msg:** Contains our setup_pools message to set up allocation points. This is the innermost layer in our nested messages.
2. **Proposal Msgs:** Contains our executable message, including our Generator Msg in binary format. Can also be used with the Astroport web app to fill the messages field.
3. **Proposal Msg:** Contains our submit_proposal message (title, description, etc.), including our Proposal Msgs array.
4. **Msg:** Contains our final message to pass into our ExecuteMsg function, including our Proposal Msg in binary format. This is the outermost layer in our nested messages.

Note: Msg titles are arbitrary variable names.

Like before, we show the entire code first and break down each section below:

1. Generator Msg

The Generator contract allocates token rewards (ASTRO) for various LP tokens and distributes them pro-rata to LP stakers. The setup_pools endpoint within the Generator contract creates a new list of pools with allocation points and updates the contract's config. "Pools" is a vector that contains LP token addresses and allocation points.

Lastly, we use our toEncodedBinary function to encode and nest our message.

There are two common mistakes to look for when crafting a Generator Msg:

1. The message expects LP token addresses, not pair addresses.
2. The message expects a vector of LP contract addresses. Specifying a single address rewrites all active pools with the pools specified in the message. You can query the generator contract to include previous LP token addresses and allocation points.

Note: In some cases, the community may want to decrease allocation points for particular pools, thus proposal submitters have to mutate config.active_pools list.

2. Proposal Msgs

Our proposal_msgs variable contains a proposal message array with our generator_address and generator_binary. This message does not need to be encoded and will be nested in a further message as is.

Note: This array will serve as the input to the messages field in our proposal message. This also means if you are submitting your proposal through the Astroport web app, you can stop here. Simply use the proposal_msgs array below to fill out the messages field when submitting your proposal using the Astroport governance UI.

3. Proposal Msg

The proposal_message object contains general information regarding our proposal, such as our proposal's title, description, link, and the proposal_msgs array we defined above. If this proposal message seems familiar to you, it's because it probably is! These fields are used in the Astroport web app to submit a proposal using a user-friendly UI.

We will also need to encode this message.

4. Final Msg

Lastly, our final message will call the send function in the xASTRO contract. This message requires the address of the contract that we are sending xASTRO to (assembly_address), the amount of ASTRO to send (30,000 ASTRO required for a proposal), and a msg field with our proposal binary from above.

No need to encode this message.

Execute Proposal

Now that we have our proposal messages, we can finally execute our proposal! We use MsgExecuteContract and pass in our wallet, the contract address we are sending our message to (xASTRO address), and our msg variable. We store this in an execute variable that will be use when we broadcast our transaction.

Broadcast Transaction

Lastly, we sign and broadcast our transaction using our wallet and pass in the execute variable we created above.

That's it! Simply use the command line and node.js to execute the script and retrieve the transaction hash along with other information such as the proposal id.

Stay tuned for further tutorials and docs updates regarding executable messages and proposal types.

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

DISCLAIMER

Remember, Terra 2.0 and Astroport are experimental technologies. This article does not constitute investment advice and is subject to and limited by the Astroport disclaimers, which you should review before interacting with the protocol.

Previous post
Tutorial: Structuring executable messages for Assembly proposals, Part 2: Adding proxy contracts

Next post
Tutorial: How to integrate Astroport swaps into your web-app