XION MARKETS

Smart Contracts Documentation

Last updated on 17th January, 2025.

Introduction

This documentation encompasses the smart contracts for the XIONMarkets protocol, aiming to comprehensively outline and explain its underlying features and intricacies (method arguments, intended behavior and purpose, and/or expected return types).

Note that though comprehensive, it may not fully outline all the smart contract features in detail.

The Testnet smart contract addresses (deployed on **xion-testnet-1**) in question for this case study are:

Contract	Testnet Address
Factory contract (Proxy)	xion1d43fwefsjsqglzpqs2qvgql0zxdd8338m57hp8suk72a2wjns55qjknxnl
Market contract (An iteration)	xion1wul3d327kfm2r0kc6sz5jgmj2lsfrkt7kk4sghkz5npdclwmqd5q8nyzxe

At the core, the Factory contract serves as a proxy and enumerator to forward calls (and create markets) and enumerate all the deployed Market contracts through it, while the Market contract serves as the AMM trading market for each deployed prediction on the protocol. The mock coin contract is an implementation of **cw20** from CosmWasm.

As a result, the demonstrations in this documentation will depict accessing the markets through proxy from the Factory (only the factory can make execution calls to the deployed market contracts).

Ensure to read through this in detail as it will give insights into the functionality of the XIONMarkets protocol.

Function specifications

The XIONMarkets platform consists of methods (functions) that allow for the following:

Public-facing methods

- Buy/sell shares
- Add/remove liquidity
- Resolve markets

Admin-only methods

- Create market
- Initialize market
- Resolve market

These methods are callable only via proxy through the Factory contract.

It also consists of query methods that allow for enumeration of data (fetching markets, getting user information, getting market information etc.) as well as for price quotes.

Execute Message Enumeration

All execute methods, supplied arguments and descriptions are thus:

- **create_market (title: String, description: String, end_date: u64, categories: Vec<String>, media: [String; 2]):**

  This method (callable only by admin) is used to create markets on the platform. It takes the details of the market (market description, title, logo and backdrop image URLs, etc and ensures the **end_date** argument is a date in the future).

- **record_stats (amount: Uint128, account: Addr, stat_type: String, data: Vec<Uint128>):**

  This method (callable only by a known market to the factory) is used to record statistics like trading volume, orders, unique user accounts, etc.

- **add_admin (account: Addr):**

  This method (callable only by an already existing admin) is used to add a new admin with privileges to resolve markets and create markets.

- **initialize liquidity (market: Addr, yes_price: Uint18, liquidity: Uint128):**

  This method is used to initialize a market's initial trading price and liquidity. The **yes_shares** represents the price of 'Yes'. On the contrary, the price of 'No' is set against the price of 'Yes' so that it sums to 1. Note that the units are represented with exponents for precision and UI formatting as **e8 (8 decimals)**. This means the price of 'Yes' if intended to be 0.5 will be represented as **50_000_000**. 'No' in turn becomes 0.5 (**50_000_000**), which sums up to **e8 (100_000_000)**. The **liquidity** argument represents the amount that will be deducted as liquidity to be added to the market in exchange for liquidity shares.

- **add liquidity (market: Addr, amount: Uint18):**

  This method allows a user to add liquidity to a market to facilitate trading. They are issued shares in exchange for the tokens supplied as liquidity into the market. Liquidity can only be added to a market that has not yet been resolved to either 'Yes' or 'No' and the market must have already been initialized.

- **resolve_market (market: Addr, variant: Uint128, market_index: Uint128):**

  This method (callable only by an admin) is used to resolve a market to ‘Yes’ - 1 or ‘No’ - 0. The **variant** argument specifies this, while the **market** argument specifies the contract address of the market in question. The **market_index** argument specifies the index of the market in the active markets enumeration map to assist in transitioning it to the completed markets enumeration map.

- **remove_liquidity (market: Addr, shares: Uint128):**

  This method is used to remove liquidity from a market. The **shares** argument specifies the amount of liquidity shares to be removed in exchange for tokens, while the **market** argument specifies the contract address of the market in question.

- **place_order (market: Addr, variant: Uint128, buy_or_sell: Uint128, amount: Uint128):**

  This method is used to buy and sell ‘Yes’ and ‘No’ shares associated with a market. The **market** argument specifies the contract address of the market in question. The **variant** argument specifies if it is ‘Yes’ or ‘No’ in view, the **buy_or_sell** argument specifies if it is a buy or sell order in question, while the **amount** argument specified the amount of either tokens or shares to be traded. Users are also granted incentive points per order they execute within the protocol.

- **claim (market: Addr, variant: Uint128):**

  This method is used to claim rewards at the end of the resolution phase of the market provided the market resolves to the users’ prediction (or stake(s)). The variant represents ‘Yes’ - 1 or ‘No’ - 0.

Query Message Enumeration

All query methods, supplied arguments and descriptions are thus:

- **get_market_info (contract_address: Addr, account: Addr):**

  This method is used to get details of a market as well as that of the account specified with the argument, **account**. Information outputted include shares, prices of 'Yes' and 'No' shares, as well as liquidity holdings.

- **fetch_markets (page: Uint128, items_per_page: Uint128, account: Addr, market_type: Uint128):**

  This method returns a paginated list of markets created on the protocol filtered by the **market_type** argument (specifies, all, active, or completed markets). The **items_per_page** specifies how many entries should be returned per page. The pages should be supplied in that they do not exceed the count of the items in each category. The **account** argument supplied should be ideally be the address of the caller of the method.

- **fes_address ():**

  This method returns the address in which all fees are sent to.

- **is_admin (account: Addr):**

  This method returns a boolean that specifies if the supplied **account** is an admin or not.

- **get_incentives (account: Addr):**

  Returns the total points a user has for trading on the platform.

- **quote (market: Addr, variant: Uint128, buy_or_sell: Uint128, amount: Uint128):**

  This method is used get quotes for buy and sell orders for ‘Yes’ and ‘No’ shares associated with a market. The **market** argument specifies the contract address of the market in question. The **variant** argument specifies if it is ‘Yes’ or ‘No’ in view, the **buy_or_sell** argument specifies if it is a buy or sell order in question, while the **amount** argument specified the amount of either tokens or shares to be traded.

  The return value is the total amount of shares or tokens that they will receive in return as the case may be, the price impact, updated price of resulting variant, and fees.

- **details ():**

  This method is used to get important informational values in the factory (platform) like token address, fees address, and the market code ID for deployments.

- **get_statistics ():**

  This method returns the statistics of the platform like total markets, active markets, completed markets, unique wallets, and volume traded.

The execute and query methods serve as the operational methods for the trading, liquidity management and administration of the protocol.

Conclusion

The documentation focuses on the XIONMarkets protocol's smart contracts, including its features, procedures, and intended behaviors. It emphasizes that, while extensive, it may not cover every facet of smart contracts in depth.

The Factory contract is important to the protocol, serving as a proxy for constructing and administering Market contracts, which function as Automated Market Makers (AMMs) for forecasts. The mock coin contract is constructed using CosmWasm's cw20 standard. Users will only be able to engage with Market contracts through the Factory, which is in charge of executing calls to these markets.

Overall, the purpose of this documentation is to provide insight into the functioning and structure of the XIONMarkets protocol, with a focus on understanding its smart contracts for effective use.