@melonproject/protocol documentation

Table of Contents

This documentation is at the stage of commit `7e698f6` from the @melonproject/documentation repository on Github.

Version: v0.1

Version Date: 29.05.2019

Version Notes: asciidoc

Melon Protocol Reference

The Melon Protocol intends to present a decentralized, public, permissionless, robust infrastructure for the secure management of crypographic token assets on the Ethereum blockchain. It aims to be a viable, low-cost alternative to the current fund management ecosystem, which has evolved similarly across most legal jurisdictions.

This Melon Protocol is a collection of smart contracts written in the Solidity programming language and deployed to the Ethereum blockchain. Supporting functionality allowing web browsers to freely interact with the protocol is provided by the javascript library, Melon.js.

Motivation

Today, starting and running an investment fund is an arduous and capital intensive endeavor. To a large degree, the applicable laws and requirements - formidable hurdles that have evolved over time - have originated in the singular ideal: to protect investor capital.

The dizzying array of laws, regulations and requirements present obstacles to the practice of investment management. An individual or organization wishing to engage in investment management activities must be able to bear the costs to achieve compliance with their governing authorities. Initially, there are direct costs, such as legal and regulatory fees, set-up costs and service provider costs. Operating a fund incurs other indirect costs, such as maintaining regulatory reporting with back-office support staff, IT systems and integrations intended to satisfy requirements. The implication is that managing a fund is financially viable only after a certain scale of Assets Under Management. This scale is, in practice, quite significant. This necessarily reduces the number of managers by creating barriers to entry, resulting in an exclusive set of established players already operating at scale.

We believe that choice is good and that investment management talent has no compelling reason to be subject to hurdles when certain trust-, legal- and regulatory touchpoints are provably and reliably provided by the Melon Protocol.

Ultimately, the cost of what was initially intended as investor protection, is indeed borne by the investor, in that a portion of investment performance is consumed by these costs. We believe the Melon Protocol can achieve something the legacy investment management industry simply cannot: increasing investor protection, dramatically reducing the costs of fund operations, leveling the playing field for new and innovative managers, and making fund investment viable at any scale.

Overview

The Melon protocol can be seen as having two layers. One is of all the funds in existence, controlled by their respective managers, and participated in by investors. The second is of infrastructure-level contracts, managed by our governance system, and critical to maintaining a healthy ecosystem for funds.

Each of the subsystems mentioned below are discussed in great detail elsewhere in the documentation. This article is meant to provide a high-level view of how the components are arranged, and a brief description of what each one does.

The fund

Each fund is a small constellation of smart contracts, each customized with parameters that the manager desires at creation time.

At the core of the fund is a contract called the Hub. This contract tracks the rest of the components, which can be thought of as Spokes. The Hub also provides the methods necessary for setup of the fund, and maintains an access control list of which components can call which methods.

The Vault component functions to simply store tokens on behalf of the fund, and segregate them from the rest of the components as a way to reduce surface area.

One of these other components is the Shares component, which provides a unit of account for ownership in a fund. Shares are not tradeable, but are created and destroyed as the fund is entered and exited by investors.

This entrypoint is provided by the Participation component. The functions on that contract allow investors to buy shares with one of some set of allowed assets, or to redeem them for a proportional ``slice'' of the fund's underlying assets. Share price determines the amount of shares created for a new investor, and the quantities of underlying assets given at redemption.

The Accounting component provides this overall share price for the fund. It also tracks amounts of assets owned by the fund, and computes other metrics related to the share price (e.g. gross asset value). Furtermore, the Accounting component provides a method to trigger fee payment on the fund.

Fees are rewarded to the manager at certain points during the lifetime of a fund, such as on redemption of shares, and at the end fo a designated reward period. There are currently two types of fees: management and performance fees, and they are tracked by the FeeManager component. Management fees are calculated based on time only, while the amount of performance fees given is determined by share price evolution.

A manager tries to maximize their fund's performance by making favourable trades, using methods on the Trading component. This contract provides a transparent way to interact with multiple exchange types, through its use of exchange adapters.

Managers are restricted from making bad trades beyond a certain threshold however, through rules implemented as part of the fund's risk management subsystem. These rules are also use to mitigate other malicious or negligent behaviour on behalf of the manager. The risk management rules are tracked in the Policy Manager component. Incidentally, investment in the fund via the Participation contract is also parameterized by rules called Compliance policies, which are also tracked by the Policy Manager.

While a fund's behaviour is defined in its component contracts, much of its functionality is dependent on having working infrastructure surrounding it.

Infrastructure contracts

These contracts are deployed for the benefit of the entire network. They function to keep the entire system running, but are not controlled by individual fund managers or investors. Instead, they are

managed by the governance system, or some component thereof.

The fund factory is deployed for managers to create their funds on a particular protocol version. Factories for each of the types of fund component are also deployed independently, and are leveraged by the fund factory when producing new funds.

Each exchange type also has a corresponding infrastructure *adapter* contract, which is shared by all funds. The adapter simply converts ``generic'' exchange methods on a fund's Trading component to methods that particular exchanges expect.

The Registry contract provides a high-level interface for what other infrastructure contracts know about.

One of the addresses tracked is that of the Engine contract. The Engine uses a buy-and-burn model discussed elsewhere in the documentation, taking MLN out of circulation by buying it for ETH. The ETH used is accumulated by the Engine as funds pay for and perform certain actions.

Another contract which is tracked by the Registry is the Price Source for the entire system. The Price Source is critical to the proper operation of funds, since it yields information used in many of a fund's actions, such as investment, redemption and fee payment.

As mentioned above, the infrastructure layer is managed by Governance. More information on how governance is structured and operated is described in the Governance documentation.

NOTE	The whole documentation was put together under `./melonprotocol-documentation` as pdf and adoc. To convert adoc to pdf you just need `asciidoctor-pdf`.

Assets

Token assets are the underlying constituent members of a Melon Fund. These token assets are exclusively Ethereum blockchain tokens implementing the ERC20 standard token interface.

Asset Eligibility

In its governance capacity, the Technical Council determines the Melon Fund asset universe, in that it periodically analyzes the broader Ethereum token universe, seeking eligible tokens given their exchange membership, liquidity and underlying use in the context of asset management.

Technically, a Melon Fund can hold any ERC20 compliant token. However, not all ERC20 compliant tokens are appropriate for Melon Funds to hold due to lack of a market, current pricing, exchange listing, liquidity, etc.

ETH is not ERC20

It should be noted the the Ethereum native asset, ETH, does *not* comply with the ERC20 token standard. For this reason, market observers may occasionally see `Wrapped ETH'' or` WETH'', where ETH is itself held in an ERC20 compliant wrapper contract, so as to have the same behavior when participating in smart contract infrastructure that expects the ERC20 interface. As such, the Melon Fund cannot hold native ETH, but can only hold WETH. This has no impact on the valuation of ETH and will function and be valued like-for-like in the Melon Fund.

Note on Gas

It should be understood that non-view function interaction with an Ethereum smart contract requires the payment of gas for processing capacity. Gas is priced (dynamically according to network load and capacity) in ETH, so some amount of ETH in a calling address is required. Depending on the complexity and computational intensity of a smart contract, it is possible that non-trivial amounts of ETH may be required.

Please refer to the Asset Registrar for more details.

Definitions

Melon Fund

A Melon Fund is an Ethereum smart contract deployed to the Ethereum blockchain. It is the core smart contract of the Melon Protocol. The Melon Fund has one and only one owner. A specific Melon Fund can be canonically referenced by its contract address. The Melon Fund provides the essential functionality for minimalistic investment management: fund accounting, fee calculation, share issuance, custody, security, asset segregation and the division of responsibilities.

Stakeholders

Owner

`owner` in the Solidity context refers to the address retaining special privileges regarding the operation of the smart contract.

Investment Manager

Investment Manager refers to the individual entitled to exercise discretionary portfolio management authority over the assets within the Melon Fund. As defined by the Melon Protocol, the Investment Manager is necessarily also the creator and `owner` of the Melon Fund.

Investor

Investor refers to an individual(s) who have successfully transferred eligible crypto assets, i.e. subscribed, to the Melon Fund through the requestInvestment() mechanism and holding Melon Fund share tokens in their address. Investors' addresses will have a quantity of Melon Fund shares commensurate with their subscription(s).

Actions

Redemption

Redemption refers to an action taken by an Investor invoking the `requestRedemption()` function of the Melon Fund. A successful redemption results in the Investor receiving the Melon Fund NAV proportional to their share quantity at the time of redemption and Investor share tokens will be destroyed. Redemption requests function analogously to a limit sell order, in that the Investor specifies a redemption price, but must wait for two price feed update cycles before the redemption is actually executed. The redemption execution will fail if the portfolio holds insufficient amounts of the redemption asset token or if the price of the redemption asset token has moved below the Investor's redemption request price, i.e. to the detriment of the Investor. Investors can also elect to redeem their pro-rata share of the individual underlying assets in the Melon Fund directly as a ``slice'' of the fund, i.e. a redemption-in-kind.

Investment/Subscription

Subscriptions refers to an action taken by an Investor invoking the `requestInvestment()` function of the Melon Fund. Subscription requests function analogously to a limit buy order, in that the Investor specifies subscription price, but must wait for two price feed update cycles before the investment is actually executed and accepted by the Melon Fund. The subscription execution will fail if the price of the subscription asset token has moved above the Investor's subscription request price, i.e. to the detriment of the Investor. A successful subscription results in the transfer of Investor subscription asset tokens to the Melon Fund's custody and the Investor receiving the commensurate quantity of newly created Melon Fund share tokens based on the subscribing asset's price and Melon Fund NAV at the time of subscription.

Emergency Redeem

The Emergency Redeem functionality truly embodies the idea that Investors have full, absolute control and custody of their asset tokens at all times.

Trade

Trade is an action that can be undertaken by the Investment Manager where a specific quantity of a given asset held by the Melon Fund is exchanged for another asset via an exchange as specified by the Investment Manager at fund set up.

Version Shutdown

A Version Shutdown is an action which allows anyone to shutdown any individual Melon Fund. All trading, investment/subscription and redemption in the specified redemption asset is disabled. Redemptions are fulfilled only by slice, with Investors receiving their pro-rata share of all underlying token assets in the Melon Fund. A redemption by slice on a shutdown Melon Fund remains the responsibility of the Investor. A Version Shutdown is a high-severity action which is only callable by the Melonport governance multi-signature wallet/Technical Council.

Fund Shutdown

A Fund Shutdown is an action which can be called by the Investment Manager to disable trading, investment/subscription and redemption in the specified redemption asset. Redemptions are fulfilled only by slice, with Investors receiving their pro-rata share of all underlying token assets in the Melon Fund. A redemption by slice on a shutdown Melon Fund remains the responsibility of the Investor.

Terms

Address

Address refers to any valid Ethereum address as defined by the Ethereum Protocol. Addresses consist of 40 hexadecimal characters prefaced with ``0x''. External addresses are controlled by a user's private key. Smart contract addresses have no knowable private key and are controlled by the functionality defined by the smart contract which generated the address. External addresses are usually generated by a user's wallet.

Exchange

Please refer to the LINK: Exchange section of the documentation.

Decentralized Exchange (DEX)

Please refer to the LINK: Exchange section of the documentation.

Price Feed Operator

Please refer to the LINK: Price Feed section of the documentation.

Technical Council

Technical Council refers to the Melon Protocol governance body which actions governance functions related to critical operational areas of Melon Protocol, such as Price Feed operations, Melon Universe/Asset Registrar maintenance and future protocol development direction. The Technical Council consists of individuals elected by MLN token holders. Please refer to the LINK: Governance section of the documentation.

Slice

A Slice represents a pro-rata share of all individual tokens held in the Melon Fund at the time of redemption request. The Investor will not receive the NAV value of their investment in the investment token, but will receive all individual tokens held by the Melon Fund at the time of redemption transferred to their Investor address.

Shares

An abstraction to facilitate the equitable distribution of ownership of assets in a co-mingled, collective investment fund. Ownership of fund assets is represented by shares in the Melon Fund. The Melon Fund's share price will fluctuate with the market price of the underlying asset tokens held. An Investor buying into the Melon Fund at a specific point in time buys shares at that share price, ensuring that an equitable amount of shares in the Melon Fund are granted. See also: Open-Ended Fund structure.

Melon Price Feed

Please refer to the LINK: Price Feed section of the documentation.

Melon Asset Registrar

Please refer to the LINK: Asset Registrar section of the documentation.

Melon Governance

Please refer to the LINK: Governance section of the documentation.

Melon Risk Engineering

Risk Engineering is a Melon Fund modular smart contract which facilitates the customization of

Investment Manager interaction with the Melon Fund. Currently, the Risk Engineering module can verify if a Make Order or Take Order are permitted given the Order price and the Reference Price from the Price Feed. The Risk Engineering functionality will gain a much richer toolset in forthcoming releases, enabling the Investment Manager to demonstrably constrain Investment Manager actions, embedding an ex ante trade discipline and rigor within the Melon Fund's strategy. Please refer to the LINK: Risk Engineering section of the documentation.

Melon Compliance

Compliance is a Melon Fund modular smart contract which facilitates the customization of Investor interaction with the Melon Fund. Currently, the Compliance module can make specific Investor addresses eligible or not eligible for investment, that is an address is either allowed or disallowed to issue an investment request. Redemptions are currently always allowed. For further information on the Compliance module, please refer to the LINK: Compliance section of the documentation.

Custody/Custodian

Custody is a service normally provided to funds in the traditional investment management industry. It is the practice of actually holding and safeguarding assets on behalf of a representative owner, in this case, a fund. A Custodian is a regulated, third party operator of such a service, who is entrusted with holding these assets. Contrary to a legacy fund where the fund is the legal owner of underlying assets held by a third party custodian, the Melon Fund smart contract *is* the Custodian of the fund's token assets, with the Investor having ultimate power of control of Investor assets held within the Melon Fund. At the same time, the Investment Manager has discretionary control over the asset allocation within the Melon Fund.

Administration/Administrator

Administration is a service normally provided to funds in the traditional investment management industry and is usually a legal requirement which an independent, regulated third party must fulfill. There are many facets to fund administration:

- NAV calculation
- Securities pricing
- Preparation of financial statement, reports, filings, prospectus, etc.
- Fund accountant
- Preparation of filings
- Reconciliation with Custodian and Broker
- Daily trade settlement
- Calculation and payment of fund expenses
- Performance calculation
- Monitoring investment Compliance
- Supervision of liquidation

In the Melon Fund context, these duties are carried out by transparent, immutable and

dispassionate smart contracts running on the Ethereum blockchain infrastructure.

Audit/Auditor

Auditing is a service normally provided to funds in the traditional investment management industry, and is usually a legal requirement which an independent third party must fulfill. Auditors reconcile and compare books and accounts maintained by various counterparties to a fund such as the manager, custodian and administrator, ensuring correct record-keeping between them. Auditors issue opinions with the weight of their reputation as to the correctness of a fund's state of affairs. In the traditional investment management environment today, a high level of certainty and comfort is provided by multiple independent inspections of fund operations; essentially counterparties monitoring each other. This operational overhead naturally incurs costs which are borne by fund performance. In the Melon Fund and blockchain context, much of this overhead is obviated in that calculations are deterministic and transparent, and the fact that the trade ***IS*** the transfer ***IS*** the settlement ***IS*** the record. That is to say, there is only on canonical source of truth: the very sending of a transaction/transfer ***IS*** the recording of the transaction, rendering comparison and verification superfluous.

Transfer Agent

The Transfer Agent in the traditional investment management industry is an institution which interacts closely with the Registrar in maintaining records and entries for the ownership of securities. Transfer Agents also ensure that interest- or dividend payments are made to the security owners on time. In the Melon Fund and blockchain context, this duty is inherently embedded in the smart contract/blockchain infrastructure of an asset token ledger, and is carried out as a matter of course.

Registrar

The Registrar in the traditional investment management industry is an institution which maintains the record of ownership of securities. In the Melon Fund and blockchain context, this duty is inherently embedded in the smart contract/blockchain infrastructure of an asset token ledger, and is carried out as a matter of course.

Order

An Order is a specification for a desired asset token trade. The specification of the trade enumerates the buy asset, sell asset, buy quantity and buy price. Taken together, these things derive an implicit relative price of one asset in terms of the other. Active Orders explicitly created for- and issued to a specific exchange can be seen as a live instruction to transact the trade as specified.

Make Order

Make Orders are Orders for which no matching analog order exists. That is, Make Orders provide liquidity to the market for the asset token specified. Make Orders will not be executed immediately, but must wait for Investor to take the opposite side of the trade.

Take Orders

Take Orders are Orders which take the opposite side of a currently existing Make Order, agreeing to the specified asset tokens in the specified quantities. Take Orders remove liquidity to the market for the asset token specified.

Order Book

An Order Book is a list of buy- and sell orders maintained by a specific exchange for a specific asset token pair. Once orders are filled, the exchange removes the order from the Order Book. The Order Book essentially describes the demand- and supply curves for a specific asset token in relative terms of another asset token.

Best Price Execution

When in the position of managing the capital of Investors in a fiduciary context, Best Price Execution refers to the practice, of endeavoring to find the most favorable terms for a given trade. The practice seeks to avoid rewarding illicit behavior at the cost of the Investor. In the context of the Melon Fund, Best Price Execution prohibits exchange trades which lie below a specified tolerance threshold to the price feed, which serves as a proxy of the current market price.

Management Fee

Management Fee is the fee charged to the Investor by the Investment Manager for the service of discretionary management of the Melon Fund's assets. The Management Fee is specified as a percentage of the Investor Gross Asset Value (GAV) on a per annum basis. The Management Fee is paid in shares of the Melon Fund. Fee shares are created out of inflating the total supply of Melon Fund shares, achieving the same impact as directly paying fees out of the Melon Fund's underlying token assets, but in a more simple and elegant mechanism that further aligns Investment Manager incentives with Investors.

Performance Fee

Performance Fee is the fee charged to the Investor by the Investment Manager for achieving positive performance in the management of the Melon Fund's assets. The Performance Fee is specified as a percentage of the positive performance achieved over a given time period, i.e. the crystalization period. The Performance Fee is paid in shares of the Melon Fund. Fee shares are created out of inflating the total supply of Melon Fund shares, achieving the same impact as directly paying fees out of the Melon Fund's underlying token assets, but in a more simple and elegant mechanism that further aligns Investment Manager incentives with Investors.

Highwatermark

Highwatermark is the NAV/Share level indicating the highest performance the Melon Fund has achieved since the last fee claim by the Investment Manager. This metric is used to determine the level of performance fees due to the Investment Manager at an given point in time.

Crystalization Period

Crystalization period is the interval of time over which a performance fee, if earned, can duly be

collected. Here, earned is taken to mean that the current NAV/Share is higher than the ending NAV/Share level of the previous period for which fees were claimed.

Hurdle

A hurdle is a percentage figure indicating a positive performance amount below which performance fees will not be charged. Performance exceeding the hurdle rate will be charged as specified under Performance Fee. Hurdle functionality in Melon Fund performance measurement is not yet implemented.

Breach

A breach occurs when a specific measurement exceeds a defined threshold.

Stake

Stake is a form of security deposit held by a smart contract and used to incentivize desired behavior. When observed behavior in contrast to specified behavior is detected, the smart contract has the ability to burn all, or a portion of the staked security deposit.

Open-Ended Fund

An Open-Ended Fund is a fund structure which places no upper limit on the amount which can be invested. As capital enters the Open-Ended Fund, new shares are created commensurate with the current NAV of the fund and the amount of capital invested. As investment capital leaves the fund through redemptions, the representative shares are destroyed, commensurate with the redemption amount.

Token

A Token is a symbolic representation of a unitized measure in a cryptographic decentralized ledger system. Token ownership (or control) resides exclusively with a cryptographic address and can only be transferred (or interacted with) by the use of the private key belonging to the holding address. The private key is used to create a cryptographically-signed transaction which is submitted to the blockchain network, which in turn executes the instructions specified therein.

ERC20 Standard

The ERC20 Standard is a standardized token interface which specifies certain functionsand events, which when implemented, allow a token to interact with other tokens and infrastructure smart contracts. See ERC20 Token Standard.

Ethereum

The Ethereum blockchain is a blockchain network which implements smart contract allowing the user to specify how state may change. Ethereum is the foundation for many decentralized applications including the Melon Protocol and Melon Funds, as well as all asset tokens traded and held in Melon Funds.

Blockchain

A Blockchain is a general purpose technology which can store data in a decentralized and immutable way. The storage and maintenance of the data is incentivized by the internal issuance of a blockchain-native token of value.

Transaction

A Transaction is a transfer of tokens/value from one cryptographic account to another or the calling of a function on a smart contract to affect some behavior on the part of the called smart contract. In some cases, a smart contract function call can also include the transfer of tokens/value.

NAV

The NAV is the Net Asset Value of the Melon Fund. The price of a fund is quoted on a per-share basis, i.e. NAV/Share. The NAV will fluctuate with the market price of the underlying holdings of the fund. Market prices are represented by the Canonical Price Feed providing a periodic valuation of the fund assets. Net Asset Value is value of net long positions in the portfolio less any credit positions (e.g. leverage) less any fees due. Currently Melon Funds can only hold long exposure positions and cannot implement leverage. This means that the NAV will be the GAV less unclaimed fees at a given point in time.

GAV

The GAV is the Gross Asset Value of the Melon Fund. The GAV will fluctuate with the market price of the underlying holdings of the fund. Market prices are represented by the Canonical Price Feed providing a periodic valuation of the fund assets. The GAV represents the current market value of the Melon Fund before the consideration of any fees due at a given point in time.

Investment/Subscription Asset

The Investment/Subscription Asset is the token asset defined by the Investment Manager with which Investors may invest or subscribe to the Melon Fund.

Redemption Asset

The Redemption Asset is the token asset defined by the Investment Manager in which Investors may redeem from the Melon Fund.

Quote Asset

The Quote Asset is the asset in which the Melon Fund is valued. The Quote Asset is specified by the Investment Manager at the time of the Melon Fund's set up.

Native Asset

The Native Asset is the token asset used by a blockchain for internal accounting, incentivization and transaction fee payment. The Native Asset on the Ethereum blockchain network is Ether (ETH).

Base Asset

TBD

Reference Asset

TBD

Allocate vs Invest

Allocate: The Investment Manager chooses and trades for specific assets within the fund. Allocation is the only discretionary power with the Investment Manager regarding the Melon Fund's underlying holdings.

Invest: The Investor transfers tokens to the Melon Fund. Technically, the Melon Fund also invests into the underlying holdings; this underscores the fact that the underlying holdings' ownership is completely segregated from the Investment Manager and resided with the Melon Fund smart contract, and ultimately, with the Investor.

Exchange Adapters

General

Exchanges are each integrated into the Melon protocol through exchange adapter contracts. Exchange adapter contracts each inherit from the standard contract ExchangeAdapter.sol. Each adapter then maps this functionality to the specific exchange contract's defined functionality, overriding the base contract's public methods.

ExchangeAdapter.sol

Description

The ExchangeAdapter contract intended to generalize the implementation of any concrete exchange adapter.

The function parameter signatures for the functions `makeOrder()`, `takeOrder()` and `cancelOrder()` are identical. The parameters are summarized here:

`address targetExchange` - The address of the intended exchange contract pertaining to the order.

`address[6] orderAddresses` - An array containing address pertaining as per below to the order. `orderAddresses[0]` - The address of the order maker. `orderAddresses[1]` - The address of the order taker. `orderAddresses[2]` - The address of the maker asset token contract. `orderAddresses[3]` - The address of the taker asset token contract. `orderAddresses[4]` - The address of the fee recipient, `feeRecipientAddress`. `orderAddresses[5]` - The address of the sender, `msg.sender`.

`uint[8] orderValues` - An array containing values pertaining as per below to the order. `orderValues[0]` - The maker asset quantity. `orderValues[1]` - The taker asset quantity. `orderValues[2]` - The maker fee. `orderValues[3]` - The taker fee. `orderValues[4]` - The expiration time of the order in seconds. `orderValues[5]` - The salt/nonce used to differentiate orders of like assets and quantities. `orderValues[6]` - The order fill quantity, i.e. the quantity of the taker asset token traded. `orderValues[7]` - The signature mode specific to the Dexy exchange.

`bytes32 identifier` - The order identifier.

`bytes makerAssetData` - Encoded data specific to the maker asset.

`bytes takerAssetData` - Encoded data specific to the taker asset.

`bytes signature` - The signature of the order maker.

Inherits from

None.

On Construction

None.

Structs

None.

Enums

None.

Modifiers

`modifier onlyManager()`

This function modifier requires that `msg.sender` is the fund manager address prior to executing functionality.

`modifier notShutDown()`

This function modifier requires that the fund is not in a shut down state prior to executing functionality.

`modifier onlyCancelPermitted(address exchange, address asset)`

This function modifier requires that either `msg.sender` is the fund manager address, the fund is in a shut down state or the order has expired prior to executing functionality.

Events

None.

Public State Variables

None.

Functions

`function getTrading() internal view returns (Trading)`

This internal view function returns the current contracts address as a `Trading` contract.

`function getHub() internal view returns (Hub)`

This internal view function returns the `Hub` contract corresponding to the `Trading` contract.

`function getAccounting() internal view returns (Accounting)`

This internal view function returns the `Accounting` contract corresponding to the current `Hub` contract.

`function hubShutDown() internal view returns (bool)`

This internal view function returns a boolean indicating the current shut down state of the fund.

`function getManager() internal view returns (address)`

This internal view function returns the address of the current fund's fund manager.

`function ensureNotInOpenMakeOrder(address _asset) internal view`

This internal view function ensures that no open make order currently exists for the asset token address provided.

`function makeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature)`

This public function must ensure that:

the fund is not shut down, the asset token prices are recent, applicable risk policies are evaluated, asset tokens to be traded are approved (if required), the make order transaction is sent to the exchange contract, the order is registered on the exchange contract (if possible), the acquired asset token, if not already held, is added to `ownedAssets`.

The function creates a make order on a specific exchange specifying the asset tokens to exchange and the corresponding quantities (implicit price) required for the order. The function on the base contract remains unimplemented and reverts.

`function takeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature)`

This public function must ensure that:

the fund is not shut down, the fund is not filling its own make order, the asset token price pair exists in the price source, the asset token prices are recent, applicable risk policies are evaluated, asset tokens to be traded are approved (if required), the take order transaction is sent to the exchange contract, the order is removed from the exchange contract (if possible), the acquired asset token, if not already held, is added to `ownedAssets`.

The function takes the opposite side of an existing make order on a specific exchange, delivering the asset token demanded by the make order in exchange for the asset token supplied by the make order, in the quantities (or less for partial order fills) specified. The function on the base contract remains unimplemented and reverts.

`function cancelOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature)`

This function must ensure that:

```
the `msg.sender` is the order owner or the order has expired or the fund is shut down,
the order is removed from the fund-internal order-tracking array,
the order on the exchange is canceled.
```

The function cancels and removes all order information within the fund and on the exchange where it existed. The function on the base contract remains unimplemented and reverts.

`function getOrder( address onExchange, uint id, address makerAsset) view returns ( address, address, uint, uint)`

This public view function is meant to return the relevant order information (maker asset token address, taker asset token address, maker asset token quantity and taker asset token quantity) given the exchange contract address, the order identifier and the maker asset token contract address parameters. The function on the base contract remains unimplemented and reverts.

EngineAdapter.sol

Description

This contract is the exchange adapter to the Melon Engine and serves as the interface from a Melon fund to the Melon Engine for the express purposes of the Melon fund trading MLN to the Melon Engine in exchange for WETH.

Inherits from

DSMath, TokenUser, ExchangeAdapter (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function takeOrder ( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature) onlyManager notShutDown`

The following parameters are used by the function:

`targetExchange` - The address of the Melon Engine exchange contract. `orderAddresses[2]` - The address of the WETH token contract (maker asset token). `orderAddresses[3]` - The address of the MLN token contract (taker asset token). `orderValues[0]` - The min quantity of ETH to get back from the Engine. `orderValues[1]` - The quantity of the MLN token, expressed in 18 decimal precision. `orderValues[6]` - Same as orderValues[1].

This function requires that the `msg.sender` is the fund manager and that the fund is not shut down. The function then requires that desired MLN token trade quantity be approved by the Melon fund. The function calls the Melon Engine to get the corresponding quantity of ETH. The Melon Engine function `sellAndBurnMln` is called, as the fund transfers MLN for WETH, as the WETH is received by the Melon fund and transferred to the Melon fund's vault, `ownedAssets` is updated with the new position if required and finally, the order status is updated.

ZeroExV2Adapter.sol

Description

This contract is the exchange adapter to the 0x v2 Exchange contract and serves as the interface from a Melon fund to the 0x v2 Exchange for purposes of exchange of asset tokens listed on the 0x v2 Exchange.

Inherits from

ExchangeAdapter, DSMath (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

```
function makeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes wrappedMakerAssetData, bytes takerAssetData, bytes signature ) onlyManager notShutDown
```

Please see parameter descriptions above.

This public function requires that the `msg.sender` is the fund manager and that the fund is not shut down. The function creates a make order on the 0x v2 exchange contract. It ensures that the maker asset token is not currently listed in any other open make order the Melon fund may have on any exchange. The taker asset token is preliminarily added to the Melon fund's owned assets and an open make order is added to the Melon fund's internal order tracking. Finally, the order is pre-signed on the 0x v2 exchange contract authorizing manager's signature on behalf of the trading contract.

```
function cancelOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues,
```

`bytes32 identifier, bytes wrappedMakerAssetData, bytes takerAssetData, bytes signature ) onlyCancelPermitted(targetExchange, orderAddresses[2])`

Please see parameter descriptions above.

This public function cancels an existing order on the 0x v2 exchange contract by calling the 0x v2 exchange contract's `cancelOrder()` function. The function applies the `onlyCancelPermitted` modifier, allowing the cancel to only be submitted under one of these conditions: the fund manager cancels the order, the fund is shut down or the order has expired. The asset token is finally removed from the Melon fund's internal order tracking.

`function getOrder( address onExchange, uint id, address makerAsset) view returns ( address, address, uint, uint)`

This public view function returns the relevant order information (maker asset token address, taker asset token address, maker asset token quantity and taker asset token quantity) given the exchange contract address, the order identifier and the maker asset token contract address parameters. The function can determine whether the order has been partially or fully filled (returning the remaining maker asset token quantity and the remaining taker asset token quantity)

`function approveTakerAsset( address targetExchange, address takerAsset, bytes takerAssetData, uint fillTakerQuantity) internal`

This internal view function withdraws fillTakerQuantity amount of the taker asset from the vault and approves the same amount to the asset proxy

`function approveMakerAsset( address targetExchange, address makerAsset, bytes makerAssetData, uint makerQuantity) internal`

This internal view function withdraws makerQuantity amount of the maker asset from the vault and approves the same amount amount to the asset proxy

`function constructOrderStruct( address[6] orderAddresses, uint[8] orderValues, bytes makerAssetData, bytes takerAssetData) internal view returns (LibOrder.Order memory order)`

This internal view function returns a populated `order` struct based on the parameter values provided.

`function getAssetProxy( address targetExchange, bytes assetData) internal view returns (address assetProxy)`

This internal view function returns the address of the asset proxy given the address of the Ethfinex exchange contract and asset data provided.

`function getAssetAddress( bytes assetData) internal view returns (address assetAddress)`

This internal view function returns the address of the asset given the asset data provided.

EthfinexAdapter.sol

Description

This contract is the exchange adapter to the Ethfinex Exchange contract and serves as the interface from a Melon fund to the Ethfinex Exchange for purposes of exchange of asset tokens listed on the

Ethfinex Exchange.

Inherits from

ExchangeAdapter, DSMath, DBC (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

```
function makeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes wrappedMakerAssetData, bytes takerAssetData, bytes signature ) onlyManager notShutDown
```

Please see parameter descriptions above.

This public function requires that the `msg.sender` is the fund manager and that the fund is not shut

down. The function creates a make order on the Ethfinex exchange contract. It ensures that the maker asset token is not currently listed in any other open make order the Melon fund may have on any exchange. The function signs the order and ensures the signature is valid. The taker asset token is preliminarily added to the Melon fund's owned assets and an open make order is added to the Melon fund's internal order tracking. Finally, the order is added to the Ethfinex exchange contract.

```
function cancelOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes wrappedMakerAssetData, bytes takerAssetData, bytes signature ) onlyCancelPermitted(targetExchange, orderAddresses[2])
```

Please see parameter descriptions above.

This public function cancels an existing order on the Ethfinex exchange contract by calling the Ethfinex exchange contract's `cancelOrder()` function. The function applies the `onlyCancelPermitted` modifier, allowing the cancel to only be submitted under one of these conditions: the fund manager cancels the order, the fund is shut down or the order has expired. The asset token is finally removed from the Melon fund's internal order tracking.

```
function withdrawTokens( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature)
```

Please see parameter descriptions above.

This public function withdraws all asset tokens held by the Ethfinex exchange in open make orders, then removes the make order form the Melon fund's internal order tracking by calling the Trading contract's `removeOpenMakeOrder()`. Finally, the function returns the asset tokens to the vault and adds the asset token to `ownedAssets`.

```
function getOrder( address onExchange, uint id, address makerAsset) view returns ( address, address, uint, uint)
```

This public view function returns the relevant order information (maker asset token address, taker asset token address, maker asset token quantity and taker asset token quantity) given the exchange contract address, the order identifier and the maker asset token contract address parameters. The function can determine whether the order has been partially or fully filled (returning the remaining maker asset token quantity and the remaining taker asset token quantity)

```
function wrapMakerAsset( address targetExchange, address makerAsset, bytes wrappedMakerAssetData, uint makerQuantity, uint orderExpirationTime) internal
```

This internal function withdraws the maker asset token from the Melon fund's vault, wrapping the maker token according to Ethfinex's wrapper registry.

```
function constructOrderStruct( address[6] orderAddresses, uint[8] orderValues, bytes makerAssetData, bytes takerAssetData) internal view returns (LibOrder.Order memory order)
```

This internal view function returns a populated `order` struct based on the parameter values provided.

```
function getAssetProxy( address targetExchange, bytes assetData) internal view returns (address assetProxy)
```

This internal view function returns the address of the asset proxy given the address of the Ethfinex exchange contract and asset data provided.

`function getAssetAddress( bytes assetData) internal view returns (address assetAddress)`

This internal view function returns the address of the asset given the asset data provided.

`function getWrapperToken( address token) internal view returns (address wrapperToken)`

This internal view function returns the address of the Ethfinex-specified wrapped asset token contract given the address of asset token contract provided.

KyberAdapter.sol

Description

This contract is the exchange adapter to the Kyber Network contract and serves as the interface from a Melon fund to the Kyber Network contract for purposes of exchange of asset tokens listed on the Kyber Network.

Inherits from

ExchangeAdapter, DSMath, DBC (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

`address public constant ETH_TOKEN_ADDRESS = 0x00eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee`

This public constant state variable represents the ETH token address.

Public Functions

`function takeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature) onlyManager notShutDown`

Please see parameter descriptions above. In the context of the Kyber exchange contract's nomenclature (swapToken() function), the following parameters take the specified mappings:

`orderAddresses[2]` - The Src token address. `orderAddresses[3]` - The Dest token address. `orderValues[0]` - The Src token quantity. `orderValues[1]` - The Dest token quantity.

This public function must ensure that:

the fund is not shut down, the `msg.sender` is the fund manager. take fill quantity is equal to the taker asset quantity, the asset token prices are recent and calculate the minimum required exchange rate, applicable risk policies are evaluated, asset tokens to be traded are approved (if required), the swap order on the Kyber exchange contract is performed, the acquired asset token, if not already held, is added to `ownedAssets`. Finally, the acquired asset tokens are returned to the Melon fund's vault and the order status is updated.

Finally, the function returns acquired token assets to the fund's vault and updates the fund's internal order tracking.

`function dispatchSwap( address targetExchange, address srcToken, uint srcAmount, address destToken, uint minRate) internal returns (uint actualReceiveAmount)`

This internal function routes the call logic depending on the asset tokens involved. The three possibilities are:

Fund trades ETH for other asset token - call made to `swapNativeAssetToToken()` Fund trades other asset token for ETH - call made to `swapTokenToNativeAsset()` Fund trades non-ETH tokens - call made to `swapTokenToToken()`

The function takes the following parameters:

`targetExchange` - The address of the intended exchange contract (i.e. the Kyber Network). `srcToken` - The address of the asset token the fund will deliver. `srcAmount` - The quantity of delivering asset token. `destToken` - The address of the asset token the fund will receive. `minRate` - The minimum acceptable quantity of the receiving asset token.

The function returns the quantity of the asset token to be received by the fund.

`function swapNativeAssetToToken( address targetExchange, address nativeAsset, uint srcAmount, address destToken, uint minRate) internal returns (uint receivedAmount)`

This internal function initiates the trade where the Melon fund delivers ETH for the receiving asset

token specified by the destToken address parameter. The function withdraws the specified quantity of ETH from the fund's vault sends the ETH quantity to the Kyber Network exchange contract calling its swapEtherToToken() function.

The function takes the following parameters:

targetExchange - The address of the intended exchange contract (i.e. the Kyber Network). nativeAsset - The address of the native asset token the fund will deliver, ETH. srcAmount - The quantity of delivering asset token. destToken - The address of the asset token the fund will receive. minRate - The minimum acceptable quantity of the receiving asset token. If minRate is not defined, the expected rate form the Kyber Network is used.

The function returns the quantity of the destToken received. The Kyber Network contract function transfers the asset to this (adapter) contract, where it is then transferred back to the vault in the takeOrder function.

```
function swapTokenToNativeAsset( address targetExchange, address srcToken, uint srcAmount, address nativeAsset, uint minRate) internal returns (uint receivedAmount)
```

This internal function initiates the trade where the Melon fund delivers the asset token specified by the srcToken address parameter for receiving ETH. The function withdraws the specified quantity of the delivery asset token from the fund's vault and sends the quantity to the Kyber Network exchange contract calling its swapTokenToEther() function. The function then approves the transfer quantity for the exchange. Finally, the function converts any ETH received to WETH.

The function takes the following parameters:

targetExchange - The address of the intended exchange contract (i.e. the Kyber Network). srcToken - The address of the asset token the fund will deliver. srcAmount - The quantity of delivering asset token. nativeAsset - The address of the native asset token the fund will receive, ETH. minRate - The minimum acceptable quantity of the receiving asset token.

The function returns the quantity of the nativeAsset received, ETH. The Kyber Network contract function transfers the asset to this (adapter) contract, where it is then transferred back to the vault in the takeOrder function.

```
function swapTokenToToken( address targetExchange, address srcToken, uint srcAmount, address destToken, uint minRate) internal returns (uint receivedAmount)
```

This internal function initiates the trade where the Melon fund delivers the asset token specified by the srcToken address parameter for receiving the asset token specified by the destToken address parameter. The function withdraws the specified quantity of the delivery asset token from the fund's vault and sends the quantity to the Kyber Network exchange contract calling its swapTokenToToken() function. The function then approves the transfer quantity for the exchange. Finally, the function converts any ETH received to WETH.

The function takes the following parameters:

targetExchange - The address of the intended exchange contract (i.e. the Kyber Network). srcToken - The address of the asset token the fund will deliver. srcAmount - The quantity of delivering asset token. destToken - The address of the asset token the fund will receive. minRate - The minimum

acceptable quantity of the receiving asset token (`destToken`). If `minRate` is not defined, the expected rate form the Kyber Network is used.

The function returns the quantity of the `destToken` received. The Kyber Network contract function transfers the asset to this (adapter) contract, where it is then transferred back to the vault in the `takeOrder` function.

```
function calcMinRate( address srcToken, address destToken, uint srcAmount, uint destAmount)
internal view returns (uint minRate)
```

This internal view function returns the minimum acceptable exchange quantity for the taker asset token from the price feed given the following parameters:

`srcToken` - The address of the asset token the fund will deliver. `destToken` - The address of the asset token the fund will receive. `srcAmount` - The quantity of delivering asset token. `destAmount` - The quantity of receiving asset token.

MatchingMarketAdapter.sol

Description

This contract is the exchange adapter to the OasisDex Matching Market exchange contract and serves as the interface from a Melon fund to the OasisDex Matching Market for purposes of exchange of asset tokens listed on the OasisDex Matching Market.

Inherits from

ExchangeAdapter, DSMath (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

`event OrderCreated(uint id)`

This event is emitted when a make order is created on the exchange contract. The event logs and broadcasts the order's identifier.

Public State Variables

None.

Public Functions

`function makeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature) onlyManager notShutDown`

Please see parameter descriptions above.

This public function must ensure that:

the fund is not shut down, the `msg.sender` is the fund manager. the asset token prices are recent, applicable risk policies are evaluated, asset tokens to be traded are approved (if required), the make order transaction is sent to the exchange contract, the order is registered on the exchange contract (if possible), the acquired asset token, if not already held, is added to `ownedAssets`.

The function creates a make order on the OasisDex Matching Market exchange contract specifying the asset tokens to exchange and the corresponding quantities (implicit price) required for the order. An `orderID > 0` signifies that the order was successfully submitted to the OasisDex Matching Market exchange contract and the open make order is added to the Melon fund's internal order tracking. Finally, the function emits the `OrderCreated()` with the order's `orderID`.

`function takeOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature) onlyManager notShutDown`

Please see parameter descriptions above. Of note are the parameters:

`orderValues[6]` - The fill quantity of taker asset token. `identifier` - The active order's orderID.

This public function must ensure that:

the fund is not shut down, the `msg.sender` is the fund manager. the fund is not filling its own make order, the asset token price pair exists in the price source, the asset token prices are recent,

applicable risk policies are evaluated, asset tokens to be traded are approved (if required), the take order transaction is sent to the exchange contract, the order is removed from the exchange contract (if possible), the acquired asset token, if not already held, is added to `ownedAssets`.

The function takes the opposite side of an existing make order on the OasisDex Matching Market exchange contract, delivering the asset token demanded by the make order in exchange for the asset token supplied by the make order, in the quantities (or less for partial order fills) specified. Finally, the function returns acquired token assets to the fund's vault and updates the fund's internal order tracking.

```
function cancelOrder( address targetExchange, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes wrappedMakerAssetData, bytes takerAssetData, bytes signature ) onlyCancelPermitted(targetExchange, orderAddresses[2])
```

Please see parameter descriptions above. Of note are the parameters:

`orderAddresses[2]` - The order maker asset token. `identifier` - The active order's orderID.

This public function cancels an existing order on the OasisDex Matching Market exchange contract by calling the OasisDex Matching Market contract's `cancel()` function. The function applies the `onlyCancelPermitted` modifier, allowing the cancel to only be submitted under one of these conditions: the fund manager cancels the order, the fund is shut down or the order has expired. The asset token is finally removed from the Melon fund's internal order tracking and the maker asset token is returned to the Melon fund's vault.

```
function getOrder( address targetExchange, uint id, address makerAsset) view returns ( address, address, uint, uint)
```

This public view function returns the current relevant order information (maker asset token address, taker asset token address, maker asset token quantity and taker asset token quantity) given the exchange contract address, the order identifier and the maker asset token contract address parameters. Orders which have been partially filled will return the remaining, respective token quantities.

Factory

General

Factory contracts are utilities for the creation of the many component contracts and infrastructure providing services to Melon funds.

Factory.sol

Description

The contract is a base contract which defines base elements below. It is inherited by all factory contracts.

Inherits from

None.

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

```
event NewInstance()
```

```
`address indexed hub` - The address of the `hub` corresponding to the created
contract.
`address indexed instance) - The address of the created contract.
```

This event is triggered when a the Factory creates a new contract. The parameters listed above are logged with the event.

Public State Variables

`mapping (address ⇒ bool) public childExists`

This public mapping state variable maps an address to boolean. It is used to determine whether an address is that of a contract created by the Factory.

Public Functions

`function isInstance(address _child) public view returns (bool)`

This public view function returns a boolean indicating whether the address provided corresponds to a contract created by the `Factory` contract.

FundFactory.sol

Description

This contract manages and executes the orderly creation of a new Melon fund, linking all aspects of components, settings, routings and permissions.

Inherits from

Factory, AmguConsumer (link)

On Construction

The contract constructor requires the following address parameters and sets the corresponding address state variables:

`address _accountingFactory` - The address of the Version's `AccountingFactory` contract. `address _feeManagerFactory` - The address of the Version's `FeeManagerFactory` contract. `address _participationFactory` - The address of the Version's `ParticipationFactory` contract. `address _sharesFactory` - The address of the Version's `SharesFactory` contract. `address _tradingFactory` - The address of the Version's `TradingFactory` contract. `address _vaultFactory` - The address of the Version's `VaultFactory` contract. `address _policyManagerFactory` - The address of the Version's `PolicyManagerFactory` contract. `address _registry` - The address of the Version's `Registry` contract. `address _version` - The address of the `Version` contract.

Structs

`Settings`

Member variables:

`string name` - The name of the Melon fund. `address[] exchanges` - An array of exchange contract addresses registered for the Melon fund. `address[] adapters` - An array of exchange adapter contract addresses registered for the Melon fund. `address denominationAsset` - The address of the

asset token in which the Melon fund is denominated. `address nativeAsset` - The address of the asset token designated as the Melon fund's native asset, i.e. the Melon fund's native network token. `address[] defaultAssets` - An array of addresses designating the Melon fund's accepted asset tokens for subscription. `bool[] takesCustody` - An array of booleans mirroring an exchange contract's `takesCustody` state variable. `address priceSource` - The address of the Melon fund's price source. `address[] fees` - An array of addresses of the Melon fund's registered `fee` contracts. `uint[] feeRates` - An array of integers representing the fee rates registered for the Melon fund. `uint[] feePeriods` - An array of integer representing the the duration of a fee performance measurement period in seconds.

Enums

None.

Modifiers

`modifier componentNotSet(address _component)`

This modifier enables a function to enforce the single execution of the implementing function and is evaluated prior to executing the implementing function's functionality.

`modifier componentSet(address _component)`

This modifier requires that the provided `_component` is set, i.e. exists. The condition is evaluated prior to executing the implementing function's functionality.

Events

NewFund()

```
`address indexed manager` - The address of the Melon fund's manager.
`address indexed hub` - The address of the Melon fund's `hub`.
`address[12] routes` - An array of 12 address representing relevant Melon fund
component addresses.
```

This event is emitted when the FundFactory creates a new fund. The event logs the parameters listed above.

Public State Variables

`VersionInterface public version`

This public state variable represents the Version contract.

`address public registry`

This public state variable represents the Version's Registry contract.

`AccountingFactory public accountingFactory`

This public state variable represents the Version's AccountingFactory contract.

`FeeManagerFactory public feeManagerFactory`

This public state variable represents the Version's FeeManagerFactory contract.

`ParticipationFactory public participationFactory`

This public state variable represents the Version's ParticipationFactory contract.

`PolicyManagerFactory public policyManagerFactory`

This public state variable represents the Version's PolicyManagerFactory contract.

`SharesFactory public sharesFactory`

This public state variable represents the Version's SharesFactory contract.

`TradingFactory public tradingFactory`

This public state variable represents the Version's TradingFactory contract.

`VaultFactory public vaultFactory`

This public state variable represents the Version's VaultFactory contract.

`address[] public funds`

This public state variable address array represents all fund contract addresses created by the FundFactory.

`mapping (address ⇒ address) public managersToHubs`

This public state variable mapping maps an address to an address, creating a relationship of manager to `hub`.

`mapping (address ⇒ Hub.Routes) public managersToRoutes`

This public state variable mapping maps an address to a `Routes` struct, creating a relationship of manager to `Routes`.

`mapping (address ⇒ Settings) public managersToSettings`

This public state variable mapping maps an address to `Settings` struct, creating a relationship of manager to `Settings`. This mapping is only used internally.

Public Functions

```
function beginSetup( string _name, address[] _fees, uint[] _feeRates, uint[] _feePeriods, address[] _exchanges, address[] _adapters, address _denominationAsset, address _nativeAsset, address[] _defaultAssets, bool[] _takesCustody ) public componentNotSet(managersToHubs[msg.sender])
```

This public function takes the following parameters and initiates the set up process of a Melon fund. The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function requires that the Version is not shut down. The function

then creates a new `Hub` owned by `msg.sender` with the `name` provided. The `Hub` is then added to the `managersToHubs` mapping, mapped to the `msg.sender` (manager). A `Settings` struct is constructed from the requisite parameter values and is added to the `managersToSettings` mapping, mapped to the `msg.sender` (manager). Finally, the function the proceeds to populate the `managersToRoutes` mapping by mapping the `msg.sender` (manager) address to the following individual `Routes` elements: `priceSource`, `registry`, `version`, `engine` and `mlnToken`.

```
function createAccounting() external componentSet(managersToHubs[msg.sender])
componentNotSet(managersToRoutes[msg.sender].accounting) amguPayable(false) payable
```

This external function creates the Accounting spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

```
function createFeeManager() external componentSet(managersToHubs[msg.sender])
componentNotSet(managersToRoutes[msg.sender].feeManager) amguPayable(false) payable
```

This external function creates the FeeManager spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

```
function createParticipation() external componentSet(managersToHubs[msg.sender])
componentNotSet(managersToRoutes[msg.sender].participation) amguPayable(false) payable
```

This external function creates the Participation spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

```
function createPolicyManager() external componentSet(managersToHubs[msg.sender])
componentNotSet(managersToRoutes[msg.sender].policyManager) amguPayable(false) payable
```

This external function creates the PolicyManager spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

```
function createShares() external componentSet(managersToHubs[msg.sender])
componentNotSet(managersToRoutes[msg.sender].shares) amguPayable(false) payable
```

This external function creates the Shares spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

```
function createTrading() external componentSet(managersToHubs[msg.sender])
```

`componentNotSet(managersToRoutes[msg.sender].trading) amguPayable(false) payable`

This external function creates the Trading spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

`function createVault() external componentSet(managersToHubs[msg.sender]) componentNotSet(managersToRoutes[msg.sender].vault) amguPayable(false) payable`

This external function creates the Vault spoke of a Melon fund and adds the resulting contract address to the `managersToRoutes` mapping, mapped to the `msg.sender` (manager). The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

`function completeSetup() external amguPayable(false) payable`

This external function completes the set up of a Melon fund. The function ensure the `childExists` mapping does not contain the `hub` address, then adds the `hub` address. The function then sets all of the `hub`'s spokes, routing, permissions and then adds the `hub` address to the `funds` state variable address array. The new Melon fund is then registered with the Version Registry. The function finally emits the `NewFund()` event, logging all parameter values as specified above. The function implements the `componentNotSet()` modifier ensuring its one-time execution in the Melon fund setup process. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductIncentive` parameter is set to `false`.

`function getFundById(uint withId) external view returns (address)`

This external view function returns the address of the Melon fund given the `withId` value provided, which is the index of of the `funds` address array state variable.

`function getLastFundId() external view returns (uint)`

This external view function returns the index position of the most recently added Melon fund to the `funds` array state variable.

`function engine() public view returns (address)`

This public view function returns the address of the Melon Engine contract.

`function mlnToken() public view returns (address)`

This public view function returns the address of the MLN token contract.

`function priceSource() public view returns (address)`

This public view function returns the address of the PriceSource contract.

`function version() public view returns (address)`

This public view function returns the address of the Version contract.

`function registry() public view returns (address)`

This public view function returns the address of the Registry contract.

FundRanking.sol

Description

The contract defines a function which returns specified information on each Melon fund created under a specific Version.

Inherits from

None.

On Construction

None.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function getFundDetails( address _factory) external view returns( address[], uint[], uint[], string[], address[])`

This public view function gathers and returns the following Melon platform details for a specific Version, given the FundFactory address provided:

`address[]` - An exhaustive array of Hub (i.e. Melon fund) addresses. `uint[]` - An exhaustive array of each Melon fund's current share price. `uint[]` - An exhaustive array of each Melon fund's creation time. `string[]` - An exhaustive array of each Melon funds' name. `address[]` - An exhaustive array of each Melon fund's denomination asset token address.

Fund

Hub & Spoke

General

The Hub and Spoke contracts make up the core contract framework of the Melon fund. Hub and Spoke is an architecture design to help reason about all the moving parts of fund as constructed by an interconnected system of linked smart contracts. The Hub will contain all relevant information for all Spokes registered with the Hub. Each Spoke will independently contain all relevant information about the Hub. Individual Spokes will have programmatic access to information in specific other Spokes.

In general, there is one Hub contract and separate, distinct Spoke contracts for each fund. (CHECK: Implementation for shared components as spoke) The Hub forms the core of the individual Melon fund with each Spoke contributing specific services to the fund. The Hub contract stores the manager's address and the fund name permanently in state, and also contains the functionality to permanently and irreversibly shut the Melon fund down.

Hub.sol

Description

The Hub maintains the specific Spoke components of the fund in terms of their routing and their permissioning. Permissioning is of utmost importance as it ensures only design-intended access to specific components of the Melon fund smart contract system. The permissioning has a very low-level granularity, exclusively permitting only specific Spokes to invoke specific functions on specific target Spokes. This surgical permissioning system ensures that sensitive function execution can occur as intended and not by external actors.

Inherits from

DSGuard (link)

On Construction

The constructor requires and sets the manager address and the fund name as well as the `creator` and `creationTime.

Structs

`Routes`

Member variables:

`address accounting` `address feeManager` `address participation` `address policyManager` `address shares` `address trading` `address vault` `address priceSource` `address registry` `address version` `address engine` `address mlnToken`

This struct stores the contract addresses of the listed components and spokes.

Enums

None.

Modifiers

`modifier onlyCreator()`

A modifier requiring that the `msg.sender` is the `creator`.

Events

None.

Public State Variables

`Routes public routes`

A `Routes` struct containing Spoke and component addresses.

`address public manager`

The address of the fund manager.

`string public name`

The name of the fund as defined by the manager at fund set up.

`bool public isShutDown`

A boolean variable defining the operational status of the fund.

`bool public spokesSet`

A boolean variable defining the completeness status of all spoke settings.

`bool public routingSet`

A boolean variable defining the completeness status of all routing settings.

`bool public permissionsSet`

A boolean variable defining the completeness status of all permissions settings.

`address public creator`

An address variable defining the creator of the Melon fund, i.e. `msg.sender`.

`uint public creationTime`

An integer variable representing the Melon fund's creation time.

mapping (address ⇒ bool) public isSpoke

`A public mapping associating an address to a boolean indicating that the address is a Spoke of the Melon fund.`

Public Functions

`function shutDownFund() public`

This function sets the `isShutDown` state variable to `true`, effectively disabling the fund for all activities except investor redemptions. This function can only be successfully called by the manager address. The function's actions are permanent and irreversible.

`function setSpokes(address[12] _spokes) onlyCreator`

This function takes an array of addresses and sets all member variables of the `routes` state variable struct if they have not already been set as part of the fund initialization sequence. The function then sets the `spokesSet` state variable to `true`. This function implements the `onlyCreator` modifier.

`function setRouting() onlyCreator`

This function requires that the Spokes and Routings have been set. It then initializes (see Spoke `initialize()` below) all registered Spokes and finally sets the `routingSet` state variable to `true`. This function implements the `onlyCreator` modifier.

`function setPermissions() onlyCreator`

This function requires that the Spokes and Routings have been set, but that permissions have not been set. Next, the functions *explicitly* sets specific, design-intended permissions between the calling contracts and called contracts' functions. The function then finally sets the `permissionsSet` state variable to `true`. This function implements the `onlyCreator` modifier.

`function vault() view returns (address)`

This view function returns the vault Spoke address.

`function accounting() view returns (address)`

This view function returns the accounting Spoke address.

`function priceSource() view returns (address)`

This view function returns the priceSource contract address.

`function participation() view returns (address)`

This view function returns the participation Spoke address.

`function trading() view returns (address)`

This view function returns the trading Spoke address.

`function shares() view returns (address)`

This view function returns the shares Spoke address.

`function policyManager() view returns (address)`

This view function returns the policyManager Spoke address.

Spoke.sol

Description

All Spoke contracts are registered with only one Hub and are initialized, storing the routes to all other Spokes registered with the Hub. Spokes serve as an abstraction of functionality and each Spoke has distinct and separate business logic domains, but may pragmatically access other Spokes registered with their Hub.

Inherits from

DSAuth (link)

On Construction

Sets the `hub` state variable and sets the the `hub` as the `authority` and `owner` as defined for permissioning within DSAuth.

Structs

`Routes`

Member variables:

`address accounting address feeManager address participation address policyManager address shares address trading address vault address priceSource address canonicalRegistrar address version address engine address mlnAddress`

This struct stores the contract addresses of the listed components and spokes.

Enums

None.

Modifiers

`modifier notShutDown()`

This modifier requires that the fund is not shut down and is evaluated prior to the execution of the functionality of the implementing function.

`modifier onlyInitialized()`

This modifier requires that the `initialized` state variable is set to `true`.

Events

None.

Public State Variables

`Hub public hub`

A variable of type `Hub`, defining the specific Hub contract to which the Spoke is connected.

`Routes public routes`

A `Routes` struct storing all initialized Spoke and component addresses.

`bool public initialized`

A boolean variable defining the initialization status of the Spoke.

Public Functions

`function initialize(address[12] _spokes) public auth`

This function requires firstly that the Spoke not be initialized, then takes an array of addresses of all Spoke- and component contracts, sets all members of the `routes` struct state variable and finally sets `initialized` = `true` and the `owner` to address ``0''.

`function engine() view returns (address)`

This view function returns the engine component contract address.

`function mlnToken() view returns (address)`

This view function returns the MLN token contract address.

`function priceSource() view returns (address)`

This view function returns the priceSource component contract address.

`function version() view returns (address)`

This view function returns the version component contract address.

Fund Manager

Overview

The Manager (Investment Manager) is the central actor in the Melon Fund. The Investment Manager is the creator of the Melon Fund and as such, the `owner` of the Melon Fund smart contract. In this capacity, the Investment Manager designs the initial set up of the Melon Fund, including:

- Compliance rules regarding Investor participation
- Risk Engineering rules regarding Investments guidelines
- Management and Performance fees
- Exchanges to be used
- Asset Tokens eligible for Subscription and Redemption
- Fund name

Once the Melon Fund is set up, it will be open for investment from Investors. When Investors transfer investment capital to the Melon Fund, the Investment Manager has full discretionary investment allocation authority over the assets, although Investors retain ultimate control over their respective shares in the Melon Fund. The discretionary investment allocation authority is, however, constrained by those Risk Engineering rules put in place by the Investment Manager during fund set up.

The critical point about a Melon Fund is that Investors retain control of their investment in the fund, while delegating asset management activity to the Investment Manager, who has the fine-grained trade and asset management authority, but no ability to remove asset tokens from the segregated safety of the Melon Fund's smart contact custody.

The Investment Manager address can only be the Investment Manager for a single Melon Fund per protocol Version, i.e. one address cannot manage multiple Melon Funds on a Version.

Fund Interaction

The Investment Manager in the capacity of `owner` of the Melon Fund smart contract has the ability to interact with the following smart contract functions:

- `enableInvestment()`
- `disableInvestment)()`
- `shutDown()` - Fund shutdown
- `callOnExchange()` - Make-, Take-, and Cancel Order

Please refer to the Fund components documentation for further details on these functions.

Investors

A Melon Fund is setup to serve Investors. That is, an Investment Manager creates a Melon Fund in order to provide the service of managing capital on behalf of participating Investors.

Investors are individuals or institutions participating in the performance of Melon Funds by sending cryptographic Asset tokens to the specific Melon Fund via the fund's `requestInvestment()` function.

In the context of the platform, the Investor is the Ethereum address from which the `requestInvestment()` was called and tokens sent (i.e. `msg.sender`).

Investor Address

The individual Investor must retain the responsibility of secure custody of the Investor address's private key. The private key is required to sign valid Ethereum transactions which are ultimately responsible for calling specific functions and sending token amounts, covering the activities of investing, redeeming and redeeming by slice.

WARNING	Loss or poor security of the Investor address's private key will result in the irreversible and permanent loss of all crypto asset tokens associated with that address.

Ethereum Addresses and Keys

An Ethereum address takes the hexidecimal format of 20 bytes prefixed by ``0x". Ethereum addresses are all lowercase, but may be mixed case if a checksum is implemented. The private key is 256 random bits.

Example Ethereum Address:
`0xdd4A3d9D44A670a80bAbbd60FD300a4C8C38561f`

Example Ethereum Private Key: `d43689ae52d6a4e0e95d41bc638e95a66c4e7d0852f6db83d44c234ce9267d0d`

Participation.sol

Description

The Participation contract encompasses the entire Melon fund interface to the investor and manages or delegates all functionality pertaining to fund subscription and redemption activities including the creation/destruction of Melon fund shares, fee calculations, asset token transfers and enabling/disabling specific asset tokens for subscription.

Inherits from

ParticipationInterface, TokenUser, AmguConsumer, Spoke (links)

On Construction

The contract requires the hub address and an array of default asset addresses. These inputs set the hub and enable investment for the assets passed in to the constructor.

The Participation contract is created from the ParticipationFactory contract, which creates a new instance of `Participation` given the `hub` address, registering the address of the newly created Participation contract as a child of the ParticipationFactory.

Structs

`Request`

Member variables:

`address investmentAsset` - The address of the asset token with which the subscription is made. `uint investmentAmount` - The quantity of tokens of the `investmentAsset` `uint requestedShares` - The quantity of fund shares for the request `uint timestamp` - The timestamp of the block containing the subscription transaction `uint atUpdateId` -

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

```
uint constant public SHARES_DECIMALS = 18
```

An integer constant which specifies the decimal precision of a single share. The value is set to 18.

```
uint constant public INVEST_DELAY = 10 minutes
```

An integer constant which specifies the number of seconds a valid subscription request must be delayed before a subscription is executed.

```
uint constant public REQUEST_LIFESPAN = 1 days
```

An integer constant which specifies the time duration where an investment subscription request is live. This constant is set to 1 day in seconds.

`uint constant public REQUEST_INCENTIVE = 10 finney`

An integer constant which specifies the amount for the subscription request provided as an incentive for triggering the delayed subscription request.

`mapping (address ⇒ Request) public requests`

A public mapping associating an investor's address to a `Request` struct.

`mapping (address ⇒ bool) public investAllowed`

A public mapping which specifies all asset token addresses which have been enabled for subscription to the Melon fund.

`mapping (address ⇒ mapping (address ⇒ uint)) public lockedAssetsForInvestor`

A public (compound) mapping associating a subscription asset token address to an investor address to a quantity of the asset token. This mapping is used by the contract to lock and hold the investor's subscription token until the request can be either executed or cancelled.

Public Functions

`function() public payable`

This public function is the contract's fallback function enabling the contract to receive ETH sent directly to the contract for the `REQUEST_INCENTIVE`.

`function enableInvestment(address[] _assets) public auth`

This function requires that the caller is the `owner` or the manager or the current contract. The function iterates over an array of provided asset token addresses, ensuring each is registered with the Melon fund's PriceFeed, and ensures that registered asset token addresses are set to `true` in the `investAllowed` mapping. Finally the function emits the `EnableInvestment()` event, logging the `_assets`.

`function disableInvestment(address[] _assets) public auth`

This function requires that the caller is the `owner` or the manager or the current contract. The function iterates over an array of provided asset token addresses and ensures that asset token addresses are set to `false` in the `investAllowed` mapping. Finally the function emits the `DisableInvestment()` event, logging the `_assets`.

`function requestInvestment(uint requestedShares, uint investmentAmount, address investmentAsset) external notShutDown payable amguPayable(true) onlyInitialized`

This function ensures that the fund is not shutdown and that subscription is permitted in the provided `investmentAsset`. The function then creates and populates a `Request` struct (see details above) from the function parameters provided and adds this to the `requests` mapping corresponding to the `msg.sender`. Finally, this function emits the `InvestmentRequest` event. Execution of this functions requires payment of AMGU ETH to the Melon Engine to provide the investment request execution incentive. This function implements the `notShutDown`, `payable`, `amguPayable` and `onlyInitialized` modifiers.

`function cancelRequest() external payable amguPayable(0)`

This function removes the request from the `requests` mapping for the request corresponding to the `msg.sender` address. The function requires that a request exists and that at least one of the conditions to cancel an investment request are met: invalid investment asset price, expired request or fund is shut down. Investment asset tokens are transferred back to the investor address. Finally, the function emits the `CancelRequest` event, logging `msg.sender`. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductFromRefund` parameter is set to `0`.

`function executeRequestFor(address requestOwner) public notShutDown amguPayable(0) payable`

This function: ensures that the fund is not shutDown, ensures that `requestOwner` has a valid subscription request, ensures that the subscription asset has a valid price, triggers a management fee calculation and reward, gets the current share cost in terms of the subscription asset, ensures that the total share cost ⇐ subscription amount, transfers the subscription assets to the fund vault, calculates change (remainder value) and returns any non-zero asset quantity to the investor address, reset the `lockedAssetsForInvestor` mapping to `` `0'' for the investor address, transfers the REQUEST_INCENTIVE to `msg.sender ``, creates the new shares and allocates them to the investor address, adds the subscription asset token to the Melon fund's list of owned assets, emits the RequestExecution() event logging `requstOwner`, `msg.sender`, `investmentAsset`, `investmentAmount` and `requestedShares`. Finally, the function removes the `Request` from the `requests` mapping for the `requestOwner` address. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductFromRefund` parameter is set to `0`.

`function getOwedPerformanceFees(uint shareQuantity) view returns (uint remainingShareQuantity)`

This view function calculates and returns the quantity of shares owed for payment of accrued performance fees given the provided quantity of shares being redeemed.

`function redeem() public`

This function determines the quantity of shares owned by `msg.sender` and calls `redeemQuantity()`. A share-commensurate quantity of all token assets in the fund are transferred to `msg.sender`, i.e. the investor. This function redeems *all* shares owned by `msg.sender`.

`function redeemQuantity(uint shareQuantity) public`

This function allows the investor to redeem a specified quantity of shares held by the `msg.sender`, i.e. the investor.

`function redeemWithConstraints(uint shareQuantity, address[] requestedAssets) public`

This function: ensures that the requested redemption share quantity is less than or equal to the share quantity owned by the investor (`msg.sender`), ensures that management fee shares have been calculated and allocated immediately prior to redemption, ensures that accrued performance fee shares have been calculated, allocated to the manager, removed from the investor's balance and added to total share supply immediately prior to redemption, maintains parity between `requestedAssets[]` array parameter and the `redeemedAssets[]` internal array, ensuring that individual asset tokens are only redeemed once, calculates the proportionate quantity of each token asset owned by the investor (`msg.sender`) based on share quantity, destroys the investor's shares and reduces the fund's total share supply by the same amount, safely transfers all token assets in the

correct quantities to the `msg.sender` (investor) address using the ERC20 `transfer()` function, and finally, the function emits the `SuccessfulRedemption()` event along with the quantity of successfully redeemed shares.

`function hasRequest(address _who) view returns (bool)`

This view function returns a boolean indicating whether the provided address has a corresponding active request with a positive quantity of requested shares in the `requests` mapping.

`function hasValidRequest(address _who) public view returns (bool)`

This public view function returns a boolean indicating the `_who` address parameter has a valid request, meaning that either this is the address's first subscription or the INVEST_DELAY is respected, the subscription amount is greater than `0'' and the requested share quantity is greater than` 0".

`function hasExpiredRequest(address _who) view returns (bool)`

This view function returns a boolean indicating that the address provided has a `requests` entry which is currently older than `REQUEST_LIFESPAN`.

Shares

Shares.sol

Description

In fund management, shares assign proportionate ownership of the collective underlying assets held in the fund. Each share in a fund has equal claim to the fund's assets. Shareholders then have a claim to the underlying fund assets in proportion to the quantity of shares under their control, i.e. ownership.

In a Melon fund, fund shares are represented as ERC20 tokens. Each token has equal and proportionate claim to the underlying assets. The share tokens are fungible but are not transferrable [CHECK if this will remain so.] between owners.

As subscribed capital enters the fund, the Shares contract will create a proportionate quantity of share tokens based on the current market prices of the fund's underlying assets and the market price of the subscribed capital and assign allocate these newly created share tokens to the subscribing address.

When share tokens are redeemed. A proportionate quantity of the fund's underlying asset tokens are transferred to the redeeming address and the redeeming address's redeemed share tokens are destroyed (i.e. set to 0) and the fund's total share quantity is reduced by that same quantity.

Inherits from

Spoke, StandardToken (links)

On construction

The Shares contract requires the address of the `hub` and becomes a `spoke` of the `hub`. The Shares contract describes the Melon fund's shares as *ERC20 tokens*. The share tokens take their name as defined by the `hub`. The share token's symbol and decimals are hardcoded in the contract.

The Shares contract is created from the sharesFactory contract, which creates a new instance of `Shares` given the `hub` address, register the address of the newly created shares contract as a child of the sharesFactory.

Public State variables

```
string public symbol
```

A public string variable denoting the Melon fund's share token symbol, currently defaulting to ``MLNF''.

```
string public name
```

A public string variable denoting the Melon fund's name as specified by the manager at set up.

```
uint8 public decimals
```

A public integer variable storing the decimal precision of the Melon fund's share quantity, currently defaulting to 18, the same decimal precision as ETH and many other ERC20 tokens.

Structs

None.

Enums

None.

Public Functions

```
function createFor(address who, uint amount) auth
```

This function requires that the caller is the `owner` or the participation component contract or the fee manager component contract or the current contract. This function calls internal function `_mint()`, which increases both `totalSupply` and the `balance` of the address by the same quantity.

```
function destroyFor(address who, uint amount) auth
```

This function requires that the caller is the `owner` or the participation component contract or the current contract. This function calls internal function `_burn()`, which decreases both `totalSupply` and the `balance` of the address by the same quantity.

Reverting functions:

create `function transfer(address to, uint amount) public returns (bool)`

`function transferFrom(address from, address to, uint amount) public returns (bool)`

`function approve(address spender, uint amount) public returns (bool)`

`function increaseApproval(address spender, uint amount) public returns (bool)`

`function decreaseApproval(address spender, uint amount) public returns (bool)`

Vault

General

Vault.sol

Description

The Vault makes use of the `auth` modifier from the DSAuth dependency. The `auth` functionality ensures the precise provisioning of call permissions, and focuses on granting `owner` or the current contract call permissions.

The Vault contract is created from the vaultFactory contract, which creates a new instance of the `Vault` given the `hub` address, register the address of the newly created vault contract as a child of the vaultFactory and finally emits an event, broadcasting the creation event along with the address of the vault contract.

Inherits from

VaultInterface, TokenUser, Spoke (link)

On construction

Requires the `Hub` address and uses this to instantiate itself as a `Spoke`.

Structs

None

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function withdraw(address token, uint amount) external auth`

This external function requires that the caller is the `owner` or the current contract. This function calls the `safeTransfer()` function from `TokenUser.sol`, safely transferring ownership of the provided amount from the vault to the custody of `msg.sender`.

Accounting

General

Accounting.sol

Description

The Accounting contract defines the accounting rules implemented by the fund. All operations concerning the underlying fund positions, fund position maintenance, asset token pricing, fees, Gross- and Net Asset value calculations and per-share calculations come together in this contract's business logic.

Inherits from

AccountingInterface, AmguConsumer, Spoke (links)

On Construction

The contract requires the hub address, the denomination asset address, the native asset address and an array of default asset addresses. These inputs set the accounting spoke's denomination asset, denomination asset decimals, the default share price (1.0 in denomination asset terms) and add all default assets passed in to the `ownedAssets` array state variable.

The Accounting contract is created from the AccountingFactory contract, which creates a new instance of `Accounting` given the `hub` address, registering the address of the newly created Accounting contract as a child of the AccountingFactory.

Structs

`Calculations`

Member Variables:

`uint gav` - The Gross Asset Value of all fund positions
`uint nav` - The Net Asset Value of all fund positions
`uint allocatedFees` - Fee shares accrued since the previous fee calculation about to be allocated `uint totalSupply` - The quantity of fund shares
`uint timestamp` - The timestamp of the current transactions block

Enums

None.

Modifiers

None.

Public State Variables

```
uint constant public MAX_OWNED_ASSETS = 20
```

A constant integer determining the maximum quantity of individual asset token positions able to be held at any point in time by a Melon fund. This constant is set to ``20''.

```
address[] public ownedAssets
```

A pubic array containing the addresses of tokens currently held by the fund.

```
mapping (address ⇒ bool) public isInAssetList
```

A mapping defining the status of an asset's membership in the `ownedAssets` array.

```
uint public constant SHARES_DECIMALS = 18
```

A public constant representing the decimal precision of Melon fund share token.

```
address public NATIVE_ASSET
```

The address of the asset token native to the platform.

`address public DENOMINATION_ASSET`

The address of the token defined to be the denomination asset, or base currency of the fund. NAV, performance and all fund-level metrics will be denominated in this asset.

`uint public DENOMINATION_ASSET_DECIMALS`

An integer determining the decimal precision, or the degree of divisibility, of the denomination asset.

`uint public DEFAULT_SHARE_PRICE`

An integer determining the initial ``sizing'' of one share in the fund relative to the denomination asset. The share price at fund inception will always be one unit of the denomination asset.

`Calculations public atLastAllocation`

A `Calculations` structure holding the latest state of the member fund calculations described above.

Public functions

`function getOwnedAssetsLength() view returns (uint)`

This public view function returns the length of the `ownedAssets` array state variable.

`function getFundHoldings() returns (uint[], address[])`

This function returns the current quantities and corresponding addresses of the funds token positions as two distinct order-dependent arrays.

`function calcAssetGAV(address _queryAsset) returns (uint)`

This function calculates and returns the current fund position GAV (in denomination asset terms) of the individual asset token as specified by the address provided.

`function assetHoldings(address _asset) public returns (uint)`

This function returns the fund position quantity of the asset token as specified by the address provided.

`function calcGav() public returns (uint gav)`

This function calculates and returns the current Gross Asset Value (GAV) of all fund assets in denomination asset terms.

`function calcNav(uint gav, uint unclaimedFees) public pure returns (uint)`

This function calculates and returns the fund's Net Asset Value (NAV) given the provided fund GAV and current quantity of unclaimed fee shares.

`function valuePerShare(uint totalValue, uint numShares) view returns (uint)`

This function calculates and returns the value (in denomination asset terms) of a single share in the fund given the fund total value and the total number of shares provided.

`function performCalculations() returns (uint gav, uint unclaimedFees, uint feesInShares, uint`

`nav, uint sharePrice)`

This view function returns bundled calculations for GAV, NAV, unclaimed fees, fee share quantity and current share price (in denomination asset terms).

`function calcSharePrice() returns (uint sharePrice)`

This function calculates and returns the current price (in denomination asset terms) of a single share in the fund.

`calcGavPerShareNetManagementFee() returns (uint gavPerShareNetManagementFee)`

This function calculates and returns the GAV (in denomination asset terms) of a single share in the fund net of the Management Fee due.

`function getShareCostInAsset(uint _numShares, address _altAsset) returns (uint)`

This public function calculates and returns the quantity of the `_altAsset` asset token commensurate with the value of `_numShares` quantity of the Melon fund's shares.

`function triggerRewardAllFees() public amguPayable(0) payable`

This public function updates `ownedAssets` and rewards all fees accrued to the current point in time. The function then updates the `atLastAllocation` struct state variable. The function is `payable` and also implements the `amguPayable` modifier, requiring amgu payment. Here, the `deductFromRefund` parameter is set to `0`.

`function updateOwnedAssets() public`

This function maintains the `ownedAssets` array by removing or adding asset addresses as the fund holdings composition changes.

`function addAssetToOwnedAssets(address _asset) public auth`

This function requires that the caller is the `owner` or the trading component contract or the participation component contract or current contract. The function maintains the `ownedAssets` array and the `isInOwedAssets` mapping by adding asset addresses as the fund holdings composition changes.

`function removeFromOwnedAssets(address _asset) public auth`

This function requires that the caller is the `owner` or the trading component contract or the current contract. The function maintains the `ownedAssets` array and the `isInOwedAssets` mapping by removing asset addresses as the fund holdings composition changes.

Fees

General

Fees are charges levied against a Melon fund's assets for:

- management services rendered, in the form of Management Fees
- fund performance achieved, in the form of Performance Fees.

Legacy investment funds are typically run as legal vehicle; a business with income and expenses. As a traditional fund incurs expenses (not only those listed above, but others like administration-, custody-, directors-, audit fess), traditional funds must liquidate assets to cash or pay the expenses out of cash held by the fund. This necessarily reduces the assets of the fund and incurs further trading costs.

Melon funds employ a novel and elegant solution: Instead of using cash, or liquidating positions to pay fees, the Melon fund smart contract itself calculates and creates new shares (or share fractions), allocating them to the investment manager's own account holdings within the fund as payment.

At that point, the investment manager can continue holding an increased number of shares in their own fund, or redeem the shares to pay their own operating costs, expand their research activities or whatever else they deem appropriate. The Melon Fund smart contract autonomously and verifiably maintains the shareholder accounting impact in a truly reliable and transparent manner.

Paying fees in this manner has a few interesting side-effects: Assets do not leave the fund, as would a cash payment. There are no unnecessary trading or cash management transactions. Investors and managers have access to real-time fee accrual metrics. Finally, the incentives to the manager are reinforced beyond a cash performance fee by being paid in the currency that is their own product.

The Management Fee and Performance Fee are each represented by an individual contract. The business logic and functionality of each fee is defined within the respective contract. The contract will interact with the various components of the fund to calculate and return the quantity of shares to create. This quantity is then added to the Manager address balance and to the total supply balance.

The Fee Manager component manages the individual fee contracts which have been configured for the Melon fund at set up. The core of the Fee Manager is an array of Fee contract instances. Functions exist to prime this array at fund setup with the selected and configured fee contracts, as well as basic query functionality to aid the user interface in calculating and representing the share NAV.

Fees are calculated and allocated when fund actions such as subscribe, redeem or claim fees are executed by a participant. Calling `rewardAllFees()` will iterate over the array of registered fees in the fee manager, calling each fee's calculation logic. This returns the quantity of shares to create and allocate to the manager.

It is important to note that fee calculations take place before the fund's share quantity is impacted by subscriptions or redemptions. To this end, when a subscription or redemption action is initiated by an investor, the execution order first calculates fee amounts and creates the corresponding share quantity, as the elapsed time and share quantity at the start is known. Essentially, the Melon fund calculates and records a reconciled state immediately and in the same transaction where share quantity changes due subscription/redemption.

Management Fees

Management Fees are earned with the passage of time, irrespective of performance. The order of fee calculations is important. The Management Fee share quantity calculation is a prerequisite to the Performance Fee calculation, as fund performance must be reduced by the Management Fee expense to fairly ascertain net performance.

The Management Fee calculation business logic is fully encapsulated by the Management Fee contract. This logic can be represented as follows.

First, the time-weighted, pre-dilution share quantity is calculated:

then, this figure is scaled such that investors retain their original share holdings quantity, but newly created shares represent the commensurate fee percentage amount:

where,

= pre-dilution quantity of shares

= current shares outstanding

= number of seconds elapsed since previous conversion event

= number of seconds in a year (= 60 _ 60 _ 24 * 365)

= Management Fee rate

= number shares to create to compensate Management Fees earned during the conversion period

Performance Fees

Performance Fees accrue over time with performance, but can only be harvested after regular, pre-determined points in time. This period is referred to as the Measurement Period and is decided by the fund manager and configured at fund set up.

Performance is assessed at the end of the Measurement Period by comparing the fund's current share price net of Management Fees to the fund's current high-water mark (HWM).

The HWM represents the highest share valuation which the Melon fund has historically achieved *at a Measurement Period ending time*. More clearly, it is not a fund all-time-high, but rather the maximum share valuation of all Measurement Period-end snapshot valuations.

If the difference to the HWM is positive, performance has been achieved and a Performance Fee is due to the fund manager. This calculation is straightforward and is the aforementioned difference multiplied by the Performance Fee rate. The Performance Fee is *not* an annualized fee rate.

The calculation of the Performance Fee requires that, at that moment, no Management Fees are due. This will be true as the code structure always invokes the Management Fee calculation and allocation immediately preceding, and in the same transaction, the Performance Fee calculation.

The Performance Fee calculation business logic is fully encapsulated by the Performance Fee contract. This logic can be represented as follows.

HWM_{MP-1}\HWM_{MP-1},&S_{n}\end{cases}“/>

HWM_{MP-1}\0,&GAV_{s}\end{cases}“/>

where,

= high-water mark for the Measurement Period

= current share price net of Management Fee

= current fund Gross Asset Value

= current fund GAV per share

= performance for the Measurement Period

= pre-dilution quantity of shares

= current shares outstanding

= Performance Fee rate

= number shares to create to compensate Performance Fees earned during the conversion period

While Performance Fees are only crystalized at the end of each measurement period, there must be a mechanism whereby redeeming investors compensate for *their* current share of accrued performance fees prior to redemption.

In the case where an investor redeems prior to the current Measurement Period’s end and where the current share price exceeds the fund HWM, a Performance Fee is due. The Redemption business logic calculates the accrued Performance Fee for the entire fund at the time of redemption and weights this by the redemption share quantity’s proportion to the fund’s total share quantity. The resulting percentage is the proportion of the redemption share quantity due as the redeeming investor’s Performance Fee payment. This quantity is deducted from the redeeming share quantity

and credited to the manager's share account balance. The remaining redeeming share quantity is destroyed as the proportionate individual token assets are transferred out of the fund and the fund's total share quantity and the investor's share quantity is reduced by this net redeeming share quantity.

Note on Fee Share Allocation

The intended behavior is for the Manager to immediately redeem fee shares as they are created. This will ensure a fair and precise share allocation. The implemented code that represents the fee calculations above contain smart contract optimizations for state variable storage. By not immediately redeeming fee shares, a negligible deviation to the manager fee payout will arise due to this optimization.

FeeManager.sol

Description

The Fee Manager is a spoke which is initialized and permissioned in the same manner as all other spokes. The Fee Manager registers and administers the execution order of the individual fee contracts.

Inherits from

Spoke, DSMath (link)

On Construction

The FeeManager contract constructor requires the hub address, the denomination asset token contract address, an array of addresses representing fee contract addresses, an array of integers representing corresponding fee rates, an array of integers representing corresponding performance fee periods and the address of the registry contract.

Structs

None.

Enums

None.

Events

FeeReward(uint shareQuantity)

This event is triggered when a Fee has been successfully allocated to the fund manager. The event logs the quantity of new fee shares created.

FeeRegistration(address fee)

This event is triggered when a Fee contract is registered with the FeeManager contract. The event logs the Fee contract address.

Public State Variables

`Fee[] public fees`

An array of type fees storing the defined fees.

`mapping (address ⇒ bool) public feeIsRegistered`

A mapping storing the registration status of a fee address.

Public Functions

`function register(address feeAddress, uint feeRate, uint feePeriod, address denominationAsset) internal`

This function adds the feeAddress provided to the `fees` array and sets the `feeIsRegistered` mapping for that address to `true`. The fee contract is initialized with the `feeRate`, `feePeriod` and `denominationAsset` token address. Finally, the FeeRegistration event is emitted.

`function totalFeeAmount() public view returns (uint total)`

This function returns the total amount of fees incurred for the hub.

`function rewardAllFees() public auth`

This function requires that the caller is the `owner` or the accounting component contract or the current contract. This function creates shares commensurate with all fees stored in the `Fee[]` state variable array.

`function rewardManagementFee() public`

This public function calculates, creates and allocates the quantity of shares currently due as the management fee.

`function managementFeeAmount() public view returns (uint)`

This public view function calculates and returns the quantity of shares currently due as the management fee.

`function performanceFeeAmount() public view returns (uint)`

This public view function calculates and returns the quantity of shares currently due as the performance fee.

ManagementFee.sol

Description

The ManagementFee contract contains the complete business logic for the creation of fund shares based on assets managed over a specified time period.

Inherits from

Fee and DSMath (link)

On Construction

None.

Structs

None.

Enums

None.

Public State Variables

```
uint public DIVISOR = 10 ** 18
```

An integer defining the standard divisor. The variable is set to 10^18.

```
mapping (address ⇒ uint) public managementFeeRate
```

An public mapping associating fund address with the management fee percentage rate.

```
mapping (address ⇒ uint) public lastPayoutTime
```

An public mapping associating fund address with the block time in UNIX epoch seconds when the previous fee payout was executed for this fund.

Public Functions

```
function feeAmount(address hub) public view returns (uint feeInShares)
```

This function calculates and returns the number of shares to be created given the amount of time since the previous fee payment, asset value and the defined management fee rate.

`function initializeForUser(uint feeRate, uint feePeriod, address denominationAsset) external`

This function ensures that no previous fee payout to the manager address on this Version has been affected. The manager address then receives a fee rate- and timestamp entry in the `managementFeeRate` and `lastPayoutTime` mappings, respectively.

`function updateState(address hub) external`

This function sets `lastPayoutTime` to the current block timestamp.

PerformanceFee.sol

Description

The PerformanceFee contract contains the complete business logic for the creation of fund shares based on fund performance over a specified Measurement Period and relative to the fund-internally-defined HWM.

Inherits from

Fee, DSMath (link)

On Construction

None.

Structs

None.

Enums

None.

Events

`HighWaterMarkUpdate(address indexed feeManager, uint indexed hwm)`

This event is triggered when the Melon fund's high-watermark is updated with a new value. The event logs the fee manager contract address and the new high-watermark value.

Public State Variables

`uint public constant DIVISOR = 10 ** 18`

A constant integer defining the standard divisor.

```
uint public constant INITIAL_SHARE_PRICE = 10 ** 18
```

A constant integer defining the initial share price to ``1''.

```
uint public constant REDEEM_WINDOW = 1 weeks
```

A constant integer defining a window of time after a measurement period where a performance fee due can be harvested. The constant is set to one week (in seconds).

```
mapping(address ⇒ uint) public initializeTime
```

A public mapping associating the Melon fund address address to the Melon fund's initialization block time.

```
mapping(address ⇒ uint) public highWaterMark
```

A public mapping associating the Melon fund address to the Melon fund's `highWaterMark`, which defines the asset value which must be exceeded at the measurement period's end to facilitate the determination of the performance fee due to the manager.

```
mapping(address ⇒ uint) public lastPayoutTime
```

A public mapping associating the Melon fund address to the Melon fund's last previous time a performance fee calculation and payout was executed. It is an integer defining the block time in UNIX epoch seconds when the previous fee payout was executed.

```
mapping(address ⇒ uint) public performanceFeeRate
```

A public mapping associating the Melon fund address to the Melon fund's configured performance fee rate.

```
mapping(address ⇒ uint) public performanceFeePeriod
```

A public mapping associating the Melon fund address to the Melon fund's configured performance measurement period. This integer express the performance measurement period in seconds.

Public Functions

```
function initializeForUser(uint feeRate, uint feePeriod, address denominationAsset) external
```

This external function is executed once at the initialization of the Melon fund. The function sets the Melon fund's performance fee rate, the performance measurement period, the fund denomination asset, the initial high-watermark, the `initializeTime` and the `lastPayoutTime`.

```
function feeAmount() public view returns (uint feeInShares)
```

This function calculates and returns the number of shares to be created given the fund performance since the previous measurement period payment, asset value and the defined performance fee rate.

```
function updateState() external
```

This function sets the `highwatermark` and `lastPayoutTime` if applicable. The function requires the

`canUpdate()` function to return `true`. Finally, the function emits the `HighWaterMarkUpdate()` event logging the fee manager address and the fund's new high-watermakr, i.e. the new GAV per share value.

```
function canUpdate(address _who) public view returns (bool)
```

This public view function returns a boolean indicating whether conditions are met to trigger the collection of a performance fee. Theses conditions are: the current time is within the update window and that an update has not already been performed for the current measurement period.

Trading

The Melon Protocol integrates with decentralized exchanges to facilitate the trading of Assets, one of the essential functionalities of a Melon Fund. This means that a Melon Fund must accommodate multiple different decentralized exchanges smart contracts if the fund is to draw from a wider pool of liquidity.

Trading.sol

Description

The Trading contract is created by the tradingFactory, which holds a queryable array of all created Trading contracts and emits the `instanceCreated` event along with the trading contract's address.

The trading contract:

-manages the set up of the interface infrastructure to the various selected exchanges.

-manages open make orders that the fund has submitted to the various registered exchanges.

-houses the common function for calling specific exchange functions through each exchange's respective exchange adapters. Note that each exchange will have their own unique interface and required parameters, which the adapters accommodate.

-provides a public function to transfer an array of token assets to the fund's vault.

-provides view functions to get order information regarding specific assets on specific exchanges as well as specific order detail.

The contract has a fallback function to retrieve ETH.

Inherits from Spoke, DSMath, TradingInterface, TokenUser (link)

On Construction

The contract requires the hub address, an array of exchange addresses, an array of exchange adapter addresses, an array of booleans indicating if an exchange takes custody of tokens for make orders and the address of the Registry contract. The contract becomes a spoke to the hub. The constructor of the trading contract requires that the length of the exchange array matches the

length of exchange adapter array and the length of the exchange array matches the length of boolean custody status array. Finally, the constructor builds the public variable `exchanges[]` array of `Exchange` structs.

Structs

`Exchange`

Member Variables

`address exchange` - The address of the decentralized exchange smart contact.

`address adapter` - The address of the corresponding adapter smart contract.

`bool takesCustody` - An flag specifying whether the exchange holds the asset token in its own custody for make orders.

`Order`

Member Variables

`address exchangeAddress` - The address of the decentralized exchange smart contact.

`bytes32 orderId` - A unique identifier for a specific order on the specific exchange.

`UpdateType updateType` - A struct indicating the update behavior/action of the order. Permitted values are `make`, `take` and `cancel`.

`address makerAsset` - The address of the asset token owned and provided in exchange.

`address takerAsset` - The address of the asset token to be received in exchange.

`uint makerQuantity` - An integer representing the quantity of the asset token owned and provided in exchange.

`uint takerQuantity` - An integer representing the quantity of the asset token owned and provided in exchange.

`uint timestamp` - The timestamp of the block in which the submitted order transaction was mined.

`uint fillTakerQuantity` - An integer representing the quantity of the maker asset token traded in the make order. This value is not necessarily the same as the `makerQuantity` because taker participants can choose to only partially execute the order, i.e. take a lower quantity of the `makerAsset` token than specified by the order's `makerQuantity`.

`OpenMakeOrder`

Member Variables

`uint id` - An integer originating from the exchange which uniquely identifies the order within the exchange.

`uint expiresAt` - An integer representing the time when the order expires. The timestamp is

represented by the Ethereum blockchain as a UNIX Epoch. After order expiration, the order will no longer [exist/be active] on the exchange and custody, if the exchange held custody, returns from the exchange contract to the fund contract.

`uint orderIndex` - An integer representing the index of the order in the `orders` array.

`address buyAsset` - An address representing the token contract of the buy asset in the order.

Enums

`UpdateType` - An enum which characterizes the type of update to the order.

Member Types

`make` - Indicates a make order type, where a quantity of a specific asset is offered at a specified price.

`take` - Indicates a take order type, where the offered quantity (or less) of a specific asset is accepted at the specified price by the counterparty. Taking less than the offered quantity in the order would be considered a ``partial fill'' of the order.

`cancel` - A type indicating that the update performed will cancel the order.

Public State variables

`Exchange[] public exchanges`

A public array of `Exchange` structs which stores all exchanges with which the fund has been initialized.

`Order[] public orders`

A public array of `Order` structs which stores all active orders [CHECK] on the

`mapping (address ⇒ bool) public adapterIsAdded`

A public mapping which indicates that a specific exchange adapter (as identified by the adapter address) is registered for the fund.

`mapping (address ⇒ mapping(address ⇒ OpenMakeOrder)) public exchangesToOpenMakeOrders`

A public compound mapping associating an exchange address to open make orders from the fund on the specific exchange.

`mapping (address ⇒ uint) public openMakeOrdersAgainstAsset`

A public mapping associating an asset token contract address to the quantity of open make orders for that asset token.

`mapping (address ⇒ bool) public isInOpenMakeOrder`

A public mapping indicating that the specified token asset is currently offered in an open make order on an exchange.

`mapping (bytes32 ⇒ LibOrder.Order) public orderIdToZeroExOrder`

A public mapping a ZeroEx order identifier to a ZeroEx Order struct.

`uint public constant ORDER_LIFESPAN = 1 days`

A public constant specifying the number of seconds that an order will remain active on an exchange. This number is added to the order creation date's timestamp to fully specify the order's expiration date. `1 days` is equal to 86400 (60 * 60 * 24).

Modifiers

`delegateInternal()`

A modifier which requires that the caller (`msg.sender`) is the current contract `Trading.sol` before the implementing function executes its functionality. This ensures that only the current contract can call a function implementing this modifier.

Public functions

`function() public payable`

The contracts fallback function making the contract able to receive ETH sent to the contract without a funciton call.

`function isOrderExpired(address exchange, address asset) public view returns (bool)`

This public view function returns a boolean indicating whether an order for the asset token and exchange provided is currently expired.

`function addExchange(address _exchange, address _adapter, bool _takesCustody) internal`

This is an internal function which is called for each exchange address passed to the constructor, adding the full exchange struct to the exchanges array state variable. The function ensures that an exchange has not previously been registered.

`function callOnExchange( uint exchangeIndex, string methodSignature, address[6] orderAddresses, uint[8] orderValues, bytes32 identifier, bytes makerAssetData, bytes takerAssetData, bytes signature ) public onlyInitialized`

This is the fund's general interface to each registered exchange for trading asset tokens. The client will call this function for specific exchange/trading interactions. This function first calls the policyManager to ensure that function-specific policies are pre- or post-executed to ensure that the exchange trade adheres to the policies configured for the fund. This function implements the `onlyInitialized` modifier. Finally, the function emits the `ExchangeMethodCall()` event, logging the specified parameters.

`function addOpenMakeOrder( address ofExchange, address sellAsset, address buyAsset, uint orderId, uint expirationTime ) public delegateInternal`

This public function can only be called from within the current contract. The function ensures that the sell asset token does not already have a current open sell order and that there are one or more orders in the `orders` array. If the `expirationTime` is set to `0', the order's expiration time is set to `ORDER_LIFESPAN`. The expiration time is required to be greater that the current `block.timestamp` and less than or equal to the sum of the `ORDER_LIFESPAN` and `block.timestamp`. The function then sets

the `isInOpenMakeOrder` mapping for the sell asset token address to `true`, and sets the details of the address's `openMakeOrder` struct on the contracts `exchangesToOpenMakeOrders` mapping.

```
function removeOpenMakeOrder( address exchange, address sellAsset ) public delegateInternal
```

This public function can only be called from within the current contract. The function removes the provided sell asset token address entry for the provided exchange address.

```
function orderUpdateHook( address ofExchange, bytes32 orderId, UpdateType updateType, address[2] orderAddresses, uint[3] orderValues ) public delegateInternal
```

This public function can only be called from within the current contract. The function used the input parameters and the current execution block's timestamp to push make- or take orders to the `orders` array. [Why only make or take orders??]

```
function updateAndGetQuantityBeingTraded(address _asset) public returns (uint)
```

This public function returns the sum of the quantity of the provided asset token address held by the current contract and the quantity of the provided asset token held across all registered exchanges in the fund's make orders. The sum returned excluded quantities in make orders where the exchange does not take custody of the tokens.

```
function updateAndGetQuantityHeldInExchange(address ofAsset) public returns (uint)
```

This public function sums and returns all quantities of the provided asset token address in make orders across all registered exchanges, excluding, however, quantities in make orders where the exchange does not take custody of the tokens, but uses the ERC-20 `approve'' functionality. The rationale is that token quantities in` approve" status are not actually held by the exchange. The function also maintains the `exchangesToOpenMakeOrders` and `isInOpenMakeOrder` mappings.

```
function returnAssetToVault(address _token) public
```

This public function transfers all token quantities of the provided array of token asset addresses from the current current contract's custody back to the fund's vault.

```
function addZeroExOrderData(bytes32 orderId, LibOrder.Order zeroExOrderData) delegateInternal
```

This function adds the data provided by the parameters to the orderIdToZeroExOrder mapping state variable.

```
function returnBatchToVault(address[] _tokens) public
```

This public function returns all asset tokens represented by the `_tokens` address array parameter and returns the asset tokens to the Melon fund's vault.

```
function getExchangeInfo() public view returns (address[], address[], bool[])
```

This public view function returns two address arrays and one boolean array with all corresponding registered exchange contract addresses, adapter contract addresses and the `takesCustoday` indicators.

```
function getOpenOrderInfo(address ofExchange, address ofAsset) public view returns (uint, uint, uint)
```

This public view function takes the exchange contract address and an asset token contract address and returns three integers: the order identifier, the order expiration time and the order index.

```
function getOrderDetails(uint orderIndex) public view returns (address, address, uint, uint)
```

This public view function takes the order index as a parameter and returns the maker asset token contract address, the taker asset token contract address, the maker asset token quantity and the taker asset token quantity.

```
function getZeroExOrderDetails(bytes32 orderId) public view returns (LibOrder.Order)
```

This public view function takes the order identifier as a parameter and returns the corresponding populated `Order` struct.

Exchange Adapters

Exchange Adapters are smart contracts which communicate directly and on-chain with the intended DEX smart contract. They serve as a translation bridge between the Melon Fund and the DEX.

Currently, the Melon Protocol has adapters to integrate the following DEXs:

- Oasis DEX
- 0x (enabling interaction on the orderbooks of all 0x relayers)
- Kyber Network
- Ethfinex

Each exchange is tied to a specific adapter by the canonical registrar. A fund can be setup to use multiple exchanges, provided they are registered by the registrar.

Adapter Function Details

The `Fund.sol` smart contract in the Melon Protocol (the blockchain fund instance) commonly uses the following functions in the Exchange Adapter to interact with the intended DEX for trading purposes:

- `makeOrder()` Creates a new order in the DEX's order book. The order may not be immediately executed. Note that this function will not be implemented for the 0x adapter until 0x Version 2 is released.
- `takeOrder()` Represents implicit agreement with a standing make order on the DEX's order book. The order will be immediately executed.
- `cancelOrder()` Retracts a standing make order from the DEX's order book. The cancelation will be immediately executed.

A single event is emitted by the Exchange Adapter:

- `OrderUpdated()` Event to inform other layers (e.g. web page) that the order has been updated in some way.

The following functions are public view functions:

- `getLastOrderId()` - Constant view function which returns the last order Id on a specific exchange.
- `getOrder()` - Constant view function which returns the order's sell asset address, buy asset address, sell asset quantity and buy asset quantity on a given exchange for a given order Id.

Note that fund is not limited to these functions and can call arbitrary functions on the exchange adapters using delegate calls, provided the function signature is whitelisted by the canonical registrar.

Fund Policy

Policy Manager

The policyManager contract is the core of risk management and compliance policies. Policies are individual contracts that define and enforce specific business logic codified within. Policies are registered with the Melon fund's policyManager contract for specific function call pertaining to trading fund positions or investor subscriptions.

The policyManager is embedded into specific function calls in other Spokes of the Melon fund as required through the modifiers described below.

PolicyManager.sol

Description

In many of the functions below an array of address `addresses`, an array of uint `values` and an `identifier` are passed as parameters. For the arrays, the order of the address or value in the array is semantically significant. The individual array positions, [n], are defined within the policyManager contract as follows:

`address[5] addresses`:

[0] order maker - the address initiating or offering the trade [1] order taker - the address filling or partially filling the offered trade [2] maker asset - the token address of the asset token intended by the order maker to exchange for another token asset, i.e. the taker asset [3] taker asset - the token address of the asset token to be received by the offer maker in exchange for the maker asset [4] exchange address - address of the exchange contract on which the order is to be placed

`uint[3] values`:

[0] maker token quantity - the maximum quantity of the maker asset token to be given by the order maker in exchange for the specified taker asset token [1] taker token quantity - the maximum quantity of the taker asset token to be received by the order maker in exchange for the specified maker asset token [2] Fill amount - the quantity of the taker token exchanged in the transaction. Must be less than or equal to the taker token quantity [1]

Finally, `identifier` and `sig` parameters are described below:

`bytes32 identifier` - order id for exchanges utilizing a unique, exchange-specific order identifier

`bytes4 sig` - the keccak256 hash of the plain text function signature of the function which triggers the specific policy validation.

Inherits from

Spoke (link)

On Construction

The PolicyManager contract is passed the corresponding Hub address and sets this as the hub state variable inherited from Spoke.

The PolicyManager contract is created from the PolicyManagerFactory contract, which creates a new instance of `PolicyManager` given the `hub` address, registering the address of the newly created PolicyManager contract as a child of the PolicyManagerFactory.

Structs

`Entry`

Member variables:

`Policy[] pre` - An array of Policy contract addresses which are registered to be validated as pre-conditions to defined function calls.

`Policy[] post` - An array of Policy contract addresses which are registered to be validated as post-conditions to defined function calls.

Enums

None.

Modifiers

`modifier isValidPolicyBySig(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier)`

This modifier ensures that `preValidate()` is called prior to applied function code and that `postValidate()` is called after the applied function code, sending the function signature hash `sig` as provided.

`modifier isValidPolicy(address[5] addresses, uint[3] values, bytes32 identifier)`

This modifier ensures that `preValidate()` is called prior to applied function code and that `postValidate()` is called after the applied function code, sending the calling function signature hash

`msg.sig`.

Events

None.

Public State Variables

`mapping(bytes4 ⇒ Entry) policies`

A mapping of bytes4 to an `Entry` struct.

Public Functions

`function registerBatch(bytes4[] sig, address[] _policies) public auth`

This function requires that the caller is the `owner` or the current contract. This public function requires equal length of both array parameters. The function then iterates over the `sig` array calling the register() function, providing each function signature hash and the corresponding Policy contract address.

`function register(bytes4 sig, address _policy) public auth`

This function requires that the caller is the `owner` or the current contract. This public function first ascertains whether the Policy being registered with the PolicyManager is to be executed as a pre- or post condition and then pushes the Policy with the corresponding signature hash on to the respective pre or post Policy array within the policies mapping. Once a Policy is registered, the condition defined within the Policy will be the standard against which the policy-registered function's consequential state changes will be tested. If the state changes pass the policy test, the function will continue execution unhindered and the state changes, e.g. a trade and the respective changes to token allocation) will become final as part of a transaction in a mined block. If the state changes do not pass the policy test, the transaction will revert and no state change will be affected.

`function getPoliciesBySig(bytes4 sig) public view returns (address[], address[])`

This view function returns two address arrays (pre and post, respectively) containing all Policy contract addresses registered for the provided function signature hash. This gives the ability to query registered policies for a specific function call.

`function preValidate(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) view public`

This view function calls the `validate()` function, explicitly passing the `pre` array from the `policies` mapping state variable filtered for Policies registered for the provided function signature hash `sig`.

`function postValidate(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) view public`

This view function calls the `validate()` function, explicitly passing the `post` array from the `policies` mapping state variable filtered for Policies registered for the provided function signature hash `sig`.

```
function validate(Policy[] storage aux, bytes4 sig, address[5] addresses, uint[3] values,
bytes32 identifier) view internal
```

This internal view function receives a function-specific filtered array of Policy contract addresses and iterates over the array, calling each Policy's implemented `rule()` function. If the call to a Policy's `rule()` evaluates to `true`, execution and any state transition proceeds. If the call to a Policy's `rule()` evaluates to `false`, all execution and preliminary state transitions are reverted.

Policies

Polices are individual smart contracts which define rule or set of rules to be compared to the state of the Melon fund. Policies simply assess the current state of the Melon fund and resolve to a boolean decision, whether the action may be executed or not, returning `true` for allowed actions and `false` for disallowed actions. The Melon fund-specific Policies are deployed with parameterized values which the defined Policy logic uses to assess the permissibility of the action.

Pre- and Post-Conditionality

Policies are intended to be rules; they are intended to permit or prevent specific behavior or action depending on the *state* as compared to specified criteria.

Some Policies, due to the nature of the data required, can be immediately resolved based on the *current* state. The resolution of the Policy result is trivial because all data needed is at hand and must nod be derived or calculated. Such Policies can be defined as ``pre-condition'' Policies.

Other Policies may need to assess the consequence of the behavior or action before the logic can assess its permissibility relative to the defined rule. To do this, the result of the action must be derived or calculated, essentially asking, `What will the state be if this action is executed?'' Such Policies can be defined as` post-condition'' Policies.

On the blockchain and in smart contracts, we can use a fortunate side-effect of the process of mining and block finalization to help determine the validity of post-condition Policies. With post-condition Policies, the action or behavior is executed with the smart contract logic and the changed (but not yet finalized or mined) state is assessed against the logic and defined parameters of the post-condition Policy. In the case where this new state complies with the logic and criteria of the Policy, the action is allowed, meaning the smart contract execution is allowed to run to completion, the block is eventually mined and this new compliant state is finalized in that mined block. In the case where the new state does not comply with the logic and criteria of the Policy, the action is disallowed and the revert() function is called, stopping execution and discarding (or rolling back) all state changes. In calling the revert() function, gas is consumed to arrive at the reference state, but any unused gas is returned to the caller as the reference state is discarded.

Policy.sol

Description

The Policy contract is inherited by implemented Policies. This contract will be changed to an interface when upgraded to Solidity 0.5, as enums and structs are allowed to be defined in interfaces in that version.

Inherits from

None.

On Construction

None.

Structs

None.

Enums

`Applied` - An enum which characterizes the conditionality type of the Policy.

Member Types

`pre` - Indicates that the Policy will be evaluated prior to the corresponding function's execution.

`post` - Indicates that the Policy will be evaluated after the corresponding function's execution.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external view returns (bool)`

This view function is called by the PolicyManager to ensure that specific registered function calls are validated by their respective registered Policy validation logic. The function returns `true` when conditions defined in the Policy are met and execution continues to successful completion. If conditions defined in the Policy are no met, the function returns `false`, all execution up to that point is reverted and execution cannot complete, returning all remaining gas.

The `rule()` function takes a function signature hash `sig`, an array of address `addresses`, an array of uint `values` and an `identifier` are passed as parameters. For the arrays, the order of the address or value in the array is semantically significant. The individual array positions, [n], are defined as follows:

`address[5] addresses`:

[0] order maker - the address initiating or offering the trade [1] order taker - the address filling or partially filling the offered trade [2] maker asset - the token address of the asset token intended by the order maker to exchange for another token asset, i.e. the taker asset [3] taker asset - the token address of the asset token to be received by the offer maker in exchange for the maker asset [4] exchange address - address of the exchange contract on which the order is to be placed

`uint[3] values`:

[0] maker token quantity - the maximum quantity of the maker asset token to be given by the order maker in exchange for the specified taker asset token [1] taker token quantity - the maximum quantity of the taker asset token to be received by the order maker in exchange for the specified maker asset token [2] Fill amount - the quantity of the taker token exchanged in the transaction. Must be less than or equal to the taker token quantity [1]

Finally, `identifier` and `sig` parameters are described below:

`bytes32 identifier` - order id for exchanges utilizing a unique, exchange-specific order identifier

`bytes4 sig` - the keccak256 hash of the plain text function signature of the function which triggers the specific policy validation.

`function position() external view returns (Applied)`

This view function returns the enum `Applied` which is defined for each specific Policy contract. The `Applied` enum indicates whether the Policy logic is applied prior to, or after the corresponding function's execution. This function is called when the Policy contract is registered with the PolicyManager contract.

Compliance

General

The contracts below define the business logic for the screening of investor addresses allowed to- or prevented from subscribing to a Melon fund.

In the Melon Protocol, the term `Compliance'' revolves around the specifics of investing as an Investor into Melon Fund. This is often referred to as a` Subscription". Details around which addresses, amounts and when Investors may subscribe to the Melon Fund are configured in-, and managed by the Compliance module. To clarify, in the traditional investment management industry, ``compliance" is also used to refer to trading of underlying fund assets, as in a whether a certain asset or trade is compliant with portfolio guidelines, laws and regulations. This type of intra-portfolio asset level compliance is handled by the Risk Engineering module in the Melon Protocol.

The main use case for the Melon Fund Compliance module is the ability to create and maintain a whitelist for specific addresses. That is, the Investment Manager can explicitly define specific addresses which will then have the ability to subscribe to the Melon Fund.

Note that any listing of an address on the Compliance module whitelist only impacts the Investor's *ability* to subscribe to the Melon Fund. An existing investment from a specific address will remain invested irrespective of any change in that address's whitelist status. The current implementation does not affect an address status after the fact. Whitelist removal does not affect an invested address's current invested status or ability to redeem, but will prevent future subscriptions. The whitelist can be seen as a positive filter, creating a known universe of allowed investor addresses. Investor addresses can be added to- or removed from the whitelist, individually or in batch by the fund manager (`owner`).

Future Directions

The Melon Fund Compliance module potentially has great significance for regulators and regulatory frameworks. Possible use cases are: Regulators or KYC/AML providers may create and maintain their own whitelist of Investor addresses in a Melon Compliance module that individual Melon Funds could query; the ability for the Investment Manager to `Hard Close'' or` Soft Close" subscriptions; the ability to cap subscription amounts.

Hard Close - A fund refuses all new investment subscriptions, usually due to capacity limitations for a specific strategy.

Soft Close - A fund refuses investment subscriptions new Investors, but accepts investment top-ups from existing Investors (addresses).

UserWhitelist.sol

Description

This contract defines a positive filter list, against which subscribing addresses are verified for membership. Member addresses are permitted to subscribe to the fund.

Inherits from

Policy, DSAuth (links)

On Construction

The UserWhitelist contract requires an array of addresses (`_preApproved`) which are added to the `whitelisted` mapping.

Structs

None.

Enums

None.

Modifiers

None.

Events

```
event ListAddition(address indexed who)
```

This event is triggered when an address is added to the `whitelisted` mapping. The event logs the newly added address.

```
event ListRemoval(address indexed who)
```

This event is triggered when an address is removed from the `whitelisted` mapping. The event logs the removed address.

Public State Variables

```
mapping (address ⇒ bool) whitelisted
```

Mapping which designates an investor address as being eligible to subscribe to the fund.

Public Functions

```
function addToWhitelist(address _who) public auth
```

This function requires that the caller is the `owner` or the current contract. This function sets the

`whitelisted` mapping for the provided address to `true`, then emits the `ListAddition()` event logging the newly added address.

```
function removeFromWhitelist(address _who) public auth
```

This function requires that the caller is the `owner` or the current contract. This function sets the `whitelisted` mapping for the provided address to `false`. Addresses which had previously subscribed and are invested can not subsequently subscribe further amounts. Finally, the function emits the `ListRemoval()` event logging the removed address.

```
function batchAddToWhitelist(address[] _members) public auth
```

This function requires that the caller is the `owner` or the current contract. The function, with one transaction, enables multiple addresses in the `whitelisted` mapping to be set to `true`.

```
function batchRemoveFromWhitelist(address[] _members) public auth
```

This function requires that the caller is the `owner` or the current contract. The function, with one transaction, enables multiple addresses in the `whitelisted` mapping to be set to `false`. Addresses which had previously subscribed and are invested can not subsequently subscribe further amounts.

```
function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external
view returns (bool)
```

This function is called by the Policy Manager when functions registered for this specific policy are called. See further documentation under Policy Manager (link).

```
function position() external view returns (Applied)
```

This function is called by the Policy Manager to determine whether the policy should be executed and evaluated as a pre-condition (`Applied.pre`) or as a post-condition (`Applied.post`). Compliance contracts are called as a pre-condition to subscription and therefore return the enum `Applied.pre` during the whitelist compliance check. See further documentation under Policy Manager (link).

Risk Engineering Policies

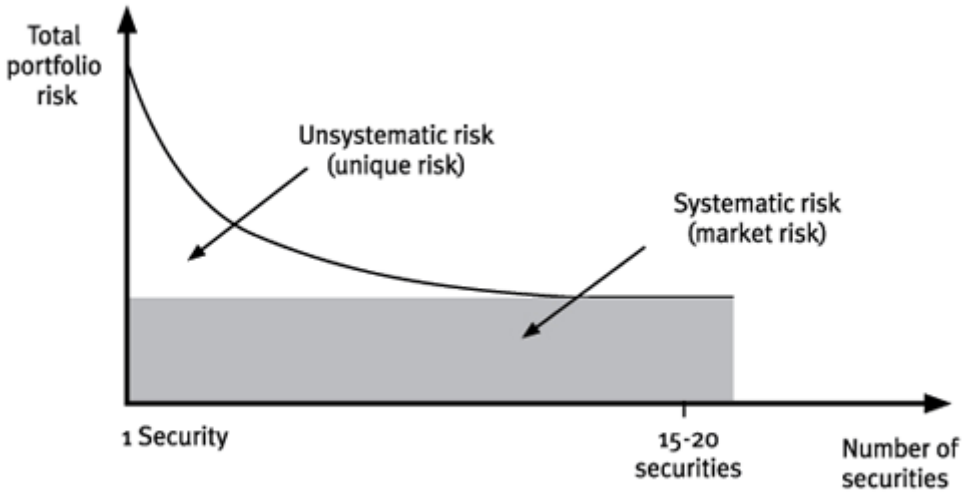
General

Risk Management vs Risk Engineering

Risk is the effect of uncertainty on objectives; the risk is the possibility that an event will occur and adversely affect the achievement of an objective.

Risk is uncertainty with implicit consequences. In general, the investor cares about two types of risk: 1) those for which she receives compensation in return for bearing, and 2) those for which no compensation is received. It is obvious that the latter must be eliminated to the extent possible.

For example, the following shows risk dissected into systematic risk and unsystematic risk:



We can observe that unsystematic risk can be *completely* eliminated from the portfolio by sufficiently diversifying or simply having a sufficient number of assets in the portfolio. Of course, these assets must be reasonably uncorrelated with each other.

Critically, an investor is NOT compensated for bearing unsystematic risk, and this would be detrimental to the portfolio's return (and the investor's wealth) over time.

Ex Post → ``Management''

Risks which have materialized should be managed to the extent possible. Examples:

- A position in the portfolio experiences positive returns to the point that the allocation breaches a concentration guideline. Managing this concentration risk means liquidating a portion of the position to return the position to a tolerable percentage of the portfolio. The proceeds are then allocated to other existing or new positions at the discretion of the investment manager.
- A position's price volatility increases to the point that it becomes too ``risky'' for the portfolio. This could be regarding the allocation or in general. In the former case, the position would be reduced so that the asset-weighted volatility of the portfolio returns to tolerable levels

Ex Ante → ``Engineering''

There are potential risks that we can explicitly prohibit with blockchain tooling. This is where we want to shine.

Portfolio guidelines are rules which are laid out in a legal document (Offering Memorandum or Prospectus) prior to fund inception. Here, the parties agree on the types of instruments to be invested, strategy, rules about concentration and other rule-based metrics.

All institutional fund managers use software to help them manage portfolios. They will say, `Our software will not allow a trade which conflicts with the guidelines'', or they will have a process in place which has a name like` pre-trade clearance'' or ``hypothetical trading''. These methods have served the industry to a satisfactory level, but non-compliant trades do find their way into a portfolio.

Melon funds approach risk differently. Melon fund currently implement *risk engineering* i.e. anticipating a specific risk and handling it in the code of the smart contract. Essentially, this is a proactive posture, as opposed to a reactive posture.

Risk engineering is the anticipation and identification of different risks, and the action of preventing their occurrence through code. Rules coded into smart contracts that are meant to ensure that something **cannot not** happen in the structure of a Melon fund.

What is risk management ?

- the **forecasting and evaluation of financial risks** together with the **identification of procedures to avoid or minimize their impact**
- **identification, evaluation and prioritization of risks** followed by coordinated and economical application of resources to minimize, monitor and control the probability or impact of unfortunate events.
- Risk management's objective is to **assure uncertainty does not deflect the endeavor from the business goals**.
- Strategies to manage threats, typically includes: **avoiding the threat, reducing the negative effect or probability of the threat, transferring all or part of the threat to another party** and even retaining some or all of the potential or actual consequences of a particular threat.
- Risk with greatest loss or impact and the greatest probability of occurring are handled first, and risks with lower probability of occurrence and lower loss are handled in descending order.

In Melon, it's different. We're not doing risk management, rather are we doing *risk engineering* ie. anticipating the risk ahead of time, and handling it in the code of the smart contract. Being proactive vs reactive.

risk engineering = the anticipation and identification of different risks ahead of time, and the action of preventing their occurrence through code. Rules coded into smart contracts that are meant to make sure that something **will not** happen in the context of a Melon fund.

What risks can we identify within the context of a Melon fund?

- Malevolent behavior of fund manager (trying to embezzle funds through orders on exchanges)
- Bad trading decisions from fund manager
- Deviation from initial strategy as laid out in the prospectus from fund manager
- Fund manager doesn't adhere to the level of risk that was communicated to the investors (volatility risk, sharpe ratio)
- Redemption risk
- Liquidity risk

What could on-chain risk engineering entail?

- Prevent investors from fund embezzlement through exchanges.
- Ensure that the fund operates accordingly to approved fund characteristics and other risk guidelines.

- Trading limits: Approve or reject trades based on: - is asset authorized? - is the asset liquid enough? - is the post trade holdings allocation acceptable? -Nominal (gross/net positions, concentration) - is the price *reasonable* (ie. best execution)? - is the volatility implications of this trade acceptable according to portfolio guidelines?
- Price feed providing additional data on assets, useful to risk management such as historical volatility, average annual rates of return and average standard deviation. Is that conceivable?

Which kind of off-chain risk engineering tooling can we provide?

- Risk engineering should also encompass monitoring of portfolio's risk in light of market risk. Simple monitoring can be done off-chain at the discretion of the fund manager. Monitoring of indicators such as downside capture [1], drawdown [2], leverage etc.
- Possible to link up your fund with risk engineering trading bot such as rebalancing bots (eg. rebalancing based on allocation or rebalancing based on risk parity). Could be simple trading bots, not necessarily enforced on-chain.
- Tools to monitor VaR: measures the dollar-loss expectation that can occur with a 5% probability.
- Tools to monitor redemption risk (based on redemption requests, redemption history and in light of current portfolio allocation)

(**Downside Capture**) In relation to hedge funds, and in particular those that claim absolute return objectives, the measure of downside capture can indicate how correlated a fund is to a market when the market declines. The lower the downside capture, the better the fund preserves wealth during market downturns. This metric is figured by calculating the cumulative return of the fund for each month that the market/benchmark was down, and dividing it by the cumulative return of the market/benchmark in the same time frame. Perfect correlation with the market will equate to a 100% downside capture and typically is only possible when comparing the benchmark to itself.

%5B**Drawdown**%5D(2 Another measure of a fund's risk is maximum drawdown. Maximum drawdown measures the percentage drop in cumulative return from a previously reached high. This metric is good for identifying funds that preserve wealth by minimizing drawdowns throughout up/down cycles, and gives an analyst a good indication of the possible losses that this fund can experience at any given point in time. Months to recover, on the other hand, gives a good indication of how quickly a fund can recuperate losses. Take the case where a hedge fund has a maximum drawdown of 4%, for example. If it took three months to reach that maximum drawdown, as investors, we would want to know if the returns could be recovered in three months or less. In some cases where the drawdown was sharp, it should take longer to recover. The key is to understand the speed and depth of a drawdown with the time it takes to recover these losses. Do they make sense given the strategy?

Rule set identification and specification

This section contains all the possible on-chain rules we have identified. Only a few of them will be implemented in v1 (see implemented policies).

Rule 1: Maximum number of positions

Definition: A fund may not exceed X positions. **Rationale**: (i) Avoid over-diversification. (ii) Avoid

excessive rebalancing transactions in the event of a new subscription (or redemption). **Implementation**: Before a trade is authorized, the system must get the number of different assets held by the fund, and make sure that adding 1 asset does not breach the X maximum number of positions. **Parameters**: X as the maximum number of underlying positions.

Rule 2: Whitelisted assets

Definition: The fund can only invest in explicitly permitted assets (included in the whitelisted asset universe defined at fund setup). **Rationale**: (i) Investors have a clear understanding of which assets can ever be invested in by the fund. (ii) Restrict the investment manager. Tighten scope of the investment universe. (iii) Fund can only invest in a niche classification (eg. ESG, utility tokens, virtual currencies, commodities only etc.) **Implementation**: Before a trade is authorized, the system must check that the asset being bought is included in the whitelisted asset universe of the fund. The asset registrar may contain an additional field *categories* that informs about the nature/type of the asset. This *category* field can then be queried by the contract before authorizing trading of the asset. Other relevant categories: type (utility, security etc.), sector, country (of jurisdiction) etc. **Parameters**: Whitelisted assets list or list of category (with specification of *AND* or *OR* relationship) **Side note**: There could be a method to remove an asset from the whitelist. Fund manager should not be able to add an asset to the whitelist.

Rule 3: Blacklisted assets

Definition: The fund is explicitly prohibited from investing in some assets. **Rationale**: (i) Investors have a guarantee that the fund will never invest in a type of asset. (ii) Restricts Investment manager (iii) Regulatory compliance (iv) Investor preferences. **Implementation**: Before a trade is authorized, the system must check that the asset being bought is not included in the blacklisted assets list. The asset registrar may contain an additional field *categories* that informs about the nature/type of the asset. This *category* field can then be queried by the contract before authorizing trading of the asset. **Parameters**: Blacklisted assets list **Side note**: There could be a method to add an asset from the backlist. Fund manager should not be able to remove an asset from the blacklist.

Rule 4: Max concentration (%)

Definition: A position (or positions in a single category) can not actively exceed X% of the NAV. **Rationale**: (i) Avoid cluster risk. (ii) Portfolio diversification (iii) Reduce unsystematic risk. **Implementation**: Before a trade is authorized, the system must check the current proportion of NAV that this asset (or category) represents and computes the post trade proportion of NAV. If the post trade proportion of NAV exceeds X%, the trade will not be authorized. **Parameters**: X as the maximum % a position's value can represent of the NAV.

Rule 5: Early redemption penalty (this may be contained in the compliance contract)

Definition: An investor can be incurred a penalty if he redeems his shares before the end of the specified period. **Rationale**: VC funds, private equity funds. Longer time horizon funds. **Implementation**: This is not part of the risk management contract per se, rather should be implemented as a compliance/participation module. **Parameters**: period (probably expressed in # of blocks before redemption is possible) and penalty.

Rule 6: Best price execution

Definition: A fund manager is not allowed to deviate more than X% away from the reference price provided by the price feed. **Rationale**: (i) Prohibit predatory investment manager behavior. (ii) Wash trading mitigation **Implementation**: Before a trade is authorized, the system must check the price of the order against the reference price and ensure the former does not deviate away more than X% from the reference price. **Parameters**: x as % of allowed deviation.

Rule 7: Turnover (expressed in # of trades)

Definition: Restrict the number of trades. The fund manager is restricted from exceeding the maximum number of trades allowed. **Rationale**: (i) Prevent predatory behavior from the investment manager. (ii) Wash trading prevention. (iii) Keep transactions fees to the minimum. **Implementation**: Before a trade is authorized, the system must check if this additional trade does not exceed the maximum # of trades allowed on the defined time period. **Parameters**: max number of trades, time period.

Consider volume turnover in terms of portfolio base currency.

Note, top-ups (reductions) due to subscriptions (redemptions) should not count as turnover. Only discretionary investment decisions

Rule 8: Turnover (expressed in volume traded)

Definition: Restrict the number of trades. The fund manager is restricted from exceeding the maximum volume in trades allowed in a given timeframe. **Rationale**: (i) Prevent predatory behavior from the investment manager. (ii) Wash trading prevention. (iii) Keep transactions fees to the minimum. **Implementation**: Before a trade is authorized, the system must check if this additional trade does not exceed the maximum volume of trades allowed on the defined time period. **Parameters**: max number volume of trades (expressed in the currency of denomination of the fund), time period.

Consider volume turnover in terms of portfolio base currency.

Note, top-ups (reductions) due to subscriptions (redemptions) should not count as turnover. Only discretionary investment decisions

Rule 9: Market cap range

Definition: A fund can only invest in assets whose market cap is contained in a certain range [x;y] (ie x < market cap < y). **Rationale**: Strategy focus enforce **Implementation**: Before trade is authorized, the system must check the market cap of the assets being traded and verify that the market cap is compliant with the market cap range defined at fund setup. The asset registrar may contain an additional field *market cap* for each asset. Open question: how to measure Market Cap. Only influences the trade permission at time of trade. That is, there is no consequence when an owned asset's market cap breaches the upper or lower specified bounds. **Parameters**: x as low boundary of the range and y as high boundary of the range.

Rule 10: Volatility threshold for a specific asset

Definition: A fund can only invest (purchase) in an asset whose volatility reference indicator is contained in a certain range [x;y] (ie x < volatility < y). **Rationale**: Investors have an understanding of the volatility level of individual assets that are allowed in the portfolio at the time of inclusion. **Implementation**: Before a trade is authorized, the system must check the volatility reference indicator of the said asset and make sure it is included in the specified [x;y] range. The volatility reference indicator could be standard deviation but also downside deviation. Possible relevant indicators: standard deviation, Sharpe ratio, Sortino indicator. **Parameters**: x as low boundary of the acceptable volatility range and y as high boundary of the acceptable volatility range.

Portfolio base currency cannot be considered for this rule.

Rule 11: Portfolio volatility threshold

Definition: A fund is prevented from performing a trade if the effect on the portfolio volatility is undesirable (ie if volatility post trade exceeds X% level of volatility) **Rationale**: Investors have an understanding of the volatility level of the portfolio maintained at the time of trading. **Implementation**: Before a trade is authorized, the system must check the post volatility of the total portfolio and make sure it is included in the specified [x;y] range. **Parameters**: x as low boundary of the acceptable portfolio volatility range and y as high boundary of the acceptable portfolio volatility range. **Side note**: This computation may be too expensive on Ethereum smart contract. We might be able to use a simplified computation. But this rule is to be envisioned in the Melon chain context.

Edge case: Could also allow trades which incrementally reduce (increase) portfolio volatility. Basically, is the trade beneficial to the investor and in the spirit of the guidelines.

Rule 12: Liquidity

Definition: A fund can only invest in assets whose liquidity is contained in a certain range [x;y] (ie x < liquidity < y).
Rationale: (i) enforce strategy focus **Implementation**: Before a trade is authorized, the system must check the liquidity available for that asset and make sure it is included in the specified [x;y] range. The liquidity data is retrieved from the exchanges the fund is allowed to trade on. Liquidity can be expressed as the % of your trade volume over the 30-days ADV (average daily volume traded). **Parameters**: x as low boundary and y as high boundary of range.

Rule 13: Correlation of adding an asset to portfolio

Definition: A fund can invest in an asset if its correlation with the current portfolio is contained between a range [x;y] **Rationale**: (i) making sure that adding an asset helps optimizing the portfolio or at least move it in the right direction. **Implementation**: Before a trade is authorized, the system must compute the correlation of that asset to the portfolio and make sure it is included in the specified [x;y] range. **Parameters**: x as low boundary and y as high boundary of the correlation range.

Rule 14: Time horizon

Definition: A fund manager needs to hold an asset for specific average period of time.

Implementation: We voluntarily decide not to encode that rule in a smart contract as it can lead to dangerous situations for the fund. Could be as a guideline for informational purposes but won't be enforced.

(not seen in practice, could be gamed by investor redemption and subsequent re-investment; must be thoughtfully aligned with lockup.)

Rule 15: Full investment

Definition: A fund manager can defined the full investment threshold ie. the proportion of assets invested vs the proportion of cash held. **Rationale**: The fund manager should not stay passive collecting management fees just by holding too large proportion of cash (taking no risk) risk. **Implementation**: We do not want to enforce that rule as the fund manager should have the flexibility to put everything in safety in case of imminent market crash or similar event. It could simply be an informational rule that can be used in monitoring/reporting tool to compare with actual investment levels of the fund. **Parameters**: X as the percentage of NAV invested. Remove, unimportant; could cause sub-optimal behavior - keep as indicator for reporting.

Rule 16: Generalizable 5/10/40 rule.

This is a current UCITS rule. Positions may not exceed 10% of NAV, and that assets which exceed 5% of NAV summed together may not exceed 40% as a group.

Rule 17: Specific Asset/Category Max (%)

This rule assigns a maximum allocation as a percentage of NAV to a specific token or category. This is slightly different from Rule 4 which is an unspecified, blanket maximum allocation for any specific asset token or category.

Rule 18: Fund Cap

This rule could be implemented by an Investment Manager to cap the overall AuM of a Melon Fund for various reasons, in particular when a specific strategy may have no further market capacity within its mandate.

Rule 19: Fund Audit

This rule could be implemented by an Investment Manager to enforce fund audits by a pre-determined auditor at defined time intervals. Funds implementing this rule would be restricted from trading (except into the denomination asset) when an audit has not been performed within the defined time interval.

Implemented Risk Policies

AssetBlacklist.sol

Description

The Policy explicitly prohibits the Melon fund from investing in or hold any asset token which is

part of the contract's blacklist. The rationale for this Policy my be: (i) investors have a guarantee that the fund will never invest in an asset. (ii) explicit restrictions on the manager (iii) regulatory compliance (iv) investor preferences. Assets token addresses can be added, but not removed from the contract's blacklist.

Inherits from

TradingSignatures, Policy, AddressList (Link)

On Construction

The contract requires an array of addresses, where each address is added to the contract's internal list.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function addToBlacklist(address _asset) external auth`

This function requires that the caller is the `owner` or the current contract. The function then requires that the token asset address is not a member of the list. If it is not a member of the list, the address

provided is then added to the contract's internal list. Once a token asset address is blacklisted, it cannot be removed.

```
function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external view returns (bool)
```

This view function returns `true` if the taker asset address (position [3] in the `addresses` array parameter) is not a member of the blacklist. The function returns `false` if the taker asset is a member of the blacklist. For a full description of the parameters, please refer to `rule()` in the general Policy contract above.

```
function position() external view returns (Applied)
```

This view function returns the enum `pre`, indicating the the Policy logic be evaluated prior to execution of the Policy's corresponding registered function call.

AssetWhitelist.sol

Description

The Policy explicitly permits the Melon fund to invest in and hold any asset token which is part of the contract's whitelist as defined at fund setup. The rationale for this Policy my be: (i) Investors have a clear understanding of which assets can ever be invested in by the fund. (ii) restricting the investment manager - tightening the scope of the investment universe. (iii) the fund can only invest in a niche classification (eg. ESG, utility tokens, virtual currencies, commodities only etc.). Assets token addresses can be removed from, but not added to the contract's whitelist.

Inherits from

TradingSignatures, Policy, AddressList (Link)

On Construction

The contract requires an array of addresses, where each address is added to the contract's internal list.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function removeFromWhitelist(address _asset) external auth`

This function requires that the caller is the `owner` or the current contract. The function then requires that the token asset address is a member of the contract's list. If it is a member of the list, the address provided is then removed from the contract's internal list. Whitelisted token asset addresses can be removed from, but not added to, the contract's list.

`function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external view returns (bool)`

This view function returns `true` if the taker asset address (position [3] in the `addresses` array parameter) is a member of the whitelist. The function returns `false` if the taker asset is not a member of the whitelist. For a full description of the parameters, please refer to `rule()` in the general Policy contract above.

`function position() external view returns (Applied)`

This view function returns the enum `pre`, indicating the the Policy logic be evaluated prior to execution of the Policy's corresponding registered function call.

MaxConcentration.sol

Description

The Policy explicitly prevents a position from actively exceeding the specified percentage of fund value. The rationale for this Policy is to: (i) avoid cluster risk, (ii) aid portfolio diversification and (iii) reduce unsystematic risk. This Policy is evaluated after execution of the registered corresponding function.

Inherits from

TradingSignatures, DSMath, Policy (Link)

On Construction

The contract requires the parameter representing the concentration upper limit as a percentage be less than 100%. The public state variable `maxConcentration` is then set to the value provided by this parameter.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

```
uint internal constant ONE_HUNDRED_PERCENT = 10 ** 18
```

This constant is used to correctly represent 100%. It is set to 10^18.

```
uint public maxConcentration
```

This public variable stores the upper limit, in percentage of fund value, which an individual asset token position can have. The value is stored using 18 decimal precision, i.e. 100000000000000000 would represent a maximum concentration for any individual token asset position of 10% of fund value.

Public Functions

```
function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external
view returns (bool)
```

This view function returns `true` if Melon fund's denomination asset is the taker asset. The rationale for this is that denomination asset should not be subject to the maximum concentration rule as the denomination asset represents non-exposure to the wider market. In cases where the taker asset is

not the denomination asset, the function evaluates the post-trade allocation percentage of the position to fund value, returning `true` if the entire fund post-trade position in the taker asset does not exceed the maximum concentration percentage. If the fund's post-trade position in the taker asset does exceed the maximum concentration percentage, the function returns `false` and the transaction is reverted. For a full description of the parameters, please refer to `rule()` in the general Policy contract above.

`function position() external view returns (Applied)`

This view function returns the enum `post`, indicating the the Policy logic be evaluated after execution of the Policy's corresponding registered function call.

MaxPositions.sol

Description

The Policy explicitly prevents a position from entering the fund if the quantity of non-denomination asset token positions exceed the specified `maxPositions` state variable. The rationale for this is to: (i) avoid over-diversification and (ii) avoid excessive rebalancing transactions in the event of a new subscription (or redemption). This Policy is evaluated after execution of the registered corresponding function.

Inherits from

TradingSignatures, Policy (Link)

On Construction

The contract requires a parameter representing the maximum number of non-denomination asset token positions permitted in the Melon fund. The public state variable `maxPositions` is then set to the value provided by this parameter. A value of 0 means that no non-denomination assets positions are permitted.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

`uint public maxPositions`

This public variable stores the upper limit of the number of non-denomination asset positions permitted in the Melon fund. For example, `maxPositions` == 5 means that a maximum of five non-denomination asset token positions are permitted at any time.

Public Functions

`function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external view returns (bool)`

This view function returns `true` if Melon fund's denomination asset is the taker asset. The rationale for this is that denomination asset should not be subject to the maximum positions rule as the denomination asset represents non-exposure to the wider market. In cases where the taker asset is not the denomination asset, the function evaluates the post-trade number of non-denomination asset positions in the fund, returning `true` if the post-trade number of non-denomination asset positions in the fund does not exceed the maximum positions configuration. If the fund's post-trade position in the taker asset causes the fund's non-denomination asset position quantity to exceed the maximum positions quantity, the function returns `false` and the transaction is reverted. For a full description of the parameters, please refer to `rule()` in the general Policy contract above.

`function position() external view returns (Applied)`

This view function returns the enum `post`, indicating the the Policy logic be evaluated after execution of the Policy's corresponding registered function call.

PriceTolerance.sol

Description

The PriceTolerance Policy explicitly prevents a make order from being submitted to an exchange, or explicitly prevents a take order trade from executing depending on a comparison between the implicit order price, adjusted for the specified tolerance, and the current reference price as provided by the price feed. The rationale is that the Policy: (i) prevents predatory investment manager behavior and (ii) mitigates wash trading. This Policy is evaluated prior to execution of the registered corresponding functions.

Inherits from

TradingSignatures, Policy, DSMath (Link)

On Construction

The contract requires a parameter representing the maximum price feed deviation of an asset trade to the detriment of fund investors. The `tolerance` state variable is then set to this value. A parameter value of 10 would represent a maximum order price deviation to the current reference price of 10%. The contract constructor requires the `_tolerancePercent` parameter to range between `0'' and` 100''.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

```
uint tolerance
```

This state variable stores the permitted deviation in percentage of the order price to the current reference price. A value of 10 would represent a maximum order price deviation to the current reference price of 10%.

```
uint constant MULTIPLIER = 10 ** 16
```

This constant is used to scale the `_tolerancePercent` parameter to correctly represent a percentage. It is set to 10^16.

```
uint constant DIVISOR = 10 ** 18
```

This constant is used to correctly represent token quantities/prices. It is set to 10^18.

Public Functions

`function takeOasisDex(address ofExchange, bytes32 identifier, uint fillTakerQuantity) view returns (bool)`

This view function gathers data from the existing Oasis Dex order and respective amounts filled, then compares the resulting price with the tolerance percentage-adjusted reference price from the price feed. If the order price does not exceed the tolerance percentage-adjusted reference price, the function returns `true` and the fund is permitted to take the order (partial- or complete fill); otherwise the function returns `false`, the trade is not permitted and the transaction is reverted.

`function takeGenericOrder(address makerAsset, address takerAsset, uint[3] values) view returns (bool)`

This view function accommodates generic exchange take order functionality, calculating the maker fill quantity given the taker quantity, maker quantity and the taker fill quantity, then compares the resulting price (fill quantity ratio) with the tolerance percentage-adjusted reference price from the price feed. If the take order price does not exceed the tolerance percentage-adjusted reference price, the function returns `true` and the fund is permitted to take the order (partial- or complete fill); otherwise the function returns `false`, the trade is not permitted and the transaction is reverted.

`function takeOrder(address[5] addresses, uint[3] values, bytes32 identifier) public view returns (bool)`

This view function is evaluated prior to executing take orders on exchanges and determines the execution routing for take orders depending on value of the `identifier` parameter. If the parameter `identifier == 0x0` (denoting a 0x-integrated exchange), the function calls `takeGenericOrder()` and `takeOasisDex()` otherwise, passing all input parameters on to the subsequent function call.

`function makeOrder(address[5] addresses, uint[3] values, bytes32 identifier) public view returns (bool)`

This view function is evaluated prior to executing make orders on exchanges. The function determines the current reference price for the asset token pair, then determines the price as determined by the make offer quantities. If the make order price does not exceed the tolerance percentage-adjusted reference price, the function returns `true` and the fund is permitted to submit the make order to the exchange; otherwise the function returns `false`, the make order is not permitted and the transaction is reverted.

`function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32 identifier) external view returns (bool)`

This view function is evaluated prior to executing make orders or take orders on exchanges and determines the execution routing for these order types depending on value of the `sig` parameter, which is the keccak256 hash of the plain text function signature of `makeOrder()` or `takeOrder()`. The function returns `true` for permitted orders, i.e. make orders can be submitted to the exchange and take orders can be traded. The function returns `false` for prohibited orders which violate the price tolerance criteria, reverting the transaction. For a full description of the parameters, please refer to `rule()` in the general Policy contract above.

`function position() external view returns (Applied)`

This view function returns the enum `pre`, indicating the the Policy logic be evaluated prior to execution of the Policy's corresponding registered function call.

Governance

Read the full post here: https://medium.com/melonport-blog/introduction-to-the-melon-governance-system-f6ff73c70eb0

Introduction to the Melon Governance System

Melon ([méllōn], μέλλων; Greek for ``future'') is an open-source protocol for on-chain asset management. It is a blockchain software system that seeks to enable participants to set up, manage and invest in technology regulated and operated funds in a way that reduces barriers to entry, whilst minimizing the requirements for trust [1]. The Melon protocol is a set of rules implementing the behavior of an investment fund as a system of smart contracts. These rules are also meant to protect the investor and fund manager from malevolent behavior of each other, even when both parties remain private.

The Melon protocol has been developed by Melonport AG. In February 2019, Melonport AG is expected to deliver v1.0 of the Melon protocol to the community, before winding down the company. This post addresses the question of future developments, maintenance and decisions relating to the Melon protocol once Melonport AG steps down as sole developer and maintainer.

Mission statement

The Melon protocol enables a new class of investment funds, referred to as Technology Regulated and Operated Funds (TROF). A TROF constitutes a first-of-its-kind on-chain and decentralized investment vehicle, made possible through leveraging decentralized technologies. Melon enables participants to perform the operational, compliance and administrative parts of asset management at a fraction of the cost, with higher transparency, security, and minimal trust requirements.

Beyond the software, Melon aims to provide an infrastructure for the asset management of the future, `μέλλων'' (Greek for` future''). The large variety of applications (eg. VC, pension funds, insurance, hedge funds, etc) in on-chain asset management need a solid infrastructure to build upon, and Melon can be seen as such; it is the backbone of the on-chain asset management, or asset management 3.0.

Melon is pioneering the way for open-source on-chain finance and, as such, is not intended to be owned by a single entity. Rather, it should be seen as an open-source public good. Melon was built in a way that is permissionless, inclusive, reliable and transparent. Users are defined as both asset managers and investors interacting with funds operating on the Melon protocol. Users do not need permission to interact with Melon.

The main goals of Melon can be summarized as:

- Providing a robust infrastructure for on-chain asset management
- Providing a reliable, secure and transparent tool for managing investment funds on-chain
- Lowering barriers to entry to what is forecast to be an $84.9 trillion asset management industry by 2025 [2]

- Nurturing innovation in the asset management industry

Governance purposes

``It’s about the processes that participants in governance use to make decisions. It’s also about how they coordinate around decisions and decision-making processes. It includes the establishment, maintenance, and revocation of the legitimacy of decisions, decision making processes, norms, and other mechanisms for coordination'' , Vlad Zamfir.

As part of the open-source finance stack, Melon is not owned by Melonport AG. In February 2019, Melonport AG will deploy v1.0 of the Melon protocol. Upon deployment, Melonport AG will no longer remain sole maintainer/developer of Melon. Therefore, as part of releasing the project in the hands of the community, Melonport AG aims at proposing a governance structure and decision-making processes that will underpin the Melon ecosystem and shall drive future development, acceptance and adoption of the Melon protocol.

To that point, it is pertinent to address the following:

- What it the scope of the decision-making processes (what)
- Principles, values and goals that shall underpin any future decision (why)
- Decision-making processes and roles (how)

What future matters require decision-making capabilities?

This section aims to address the what.

(i) Protocol upgrades

Future improvements to the Melon protocol smart contracts including bug fixes, security patches, feature additions, and third-party integrations. This also includes adding new assets to the Melon Asset Universe, as well as authorizing new exchanges to interact with the Melon protocol.

Generally, protocol upgrades refer to changes in the Melon smart contracts codebase.

(ii) Resource allocation

Inflation [4] is the only financial resource available to the Melon community and a critical source for future growth and network effects. The yearly inflation of MLN is intended to be used to fund future developments of the Melon protocol, as well as projects building in the Melon ecosystem.

Decisions need to be made as to:

The definition of priorities regarding changes or additions to the protocol Choosing adequate developers to conduct specific research or implementation Reviewing and assessing asset management projects applying for funding

(iii) Network parameters

The asset management gas price per amgu (asset management gas unit) needs to be set and adjusted according to network usage and market conditions. For more context on this, please refer

to Melonomics 2.

Which long term goals and principles should underpin Melon related decisions? ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

This section aims at addressing the why.

Melon is intended to be a robust, secure and censorship resistant asset management infrastructure.

The system aims at being drastically more efficient than any other existing solution. Melon is competitive to existing asset management alternatives and needs to remain competitive in the future. Melon constitutes an opt-in alternative to the current asset management industry.

Melon is inclusive and permissionless. It should be easily accessible without restriction, to anyone on earth, no matter where they are.

Lowering barriers to entry. Melon should always strive to lower barriers to entry for asset managers. It also gives investors access to a deeper and more accessible talent pool.

Putting users first. It should be clear that the goal is to provide a best-in-class tool to asset managers around the world. Melon aims to stay ahead of the curve and provide advanced and sophisticated infrastructure to all kinds of asset managers.

Any decisions taken in the future should be in line with the following long term goals:

- 1 Preserving the integrity of the network
- 2 Maximizing user adoption
- 3 Fostering innovation and increasing network attractiveness

Ultimately, those three goals should have a long-term positive effect on the value of the MLN token.

Melon Governance System 1.0

This section aims at addressing the how and presenting our current thinking for the Melon Governance System.

When defining this governance model, our unique goal has been to think about what is best for the Melon protocol’s longevity, integrity and success, whilst keeping in mind the experimental aspect of this type of organization.

``Governance is hard, especially for decentralized protocols. Allowing a community to govern a protocol does not make it any easier'', Dean Eigenmann [9]

The initial Melon governance system is very user-centric, and ensures the maintenance and development are driven by technically informed parties.

1. Melon Technical Council (MTC) and Melon Council (MC)

We therefore propose a technical council (see also technocratic council [10]) composed of a diverse set of known and identified entities or persons solely responsible for the following:

- Deployment of protocol upgrades (including code upgrades, feature additions and bug fixes)
- Management of the set of ENS subdomains pointing to the Melon smart contracts
- Allocation of resources to developers and application developers.
- Adjusting network parameters such as the cost per amgu (or Melon gas price)

The formation of a council aims at providing more efficiency in the decision-making system (in comparison to a token holder governance model), but also at aligning the decision-makers with the best interests of the Melon protocol and its surrounding ecosystem (through fiduciary duties towards users). It minimizes the number of decision makers and ensures those decision-makers are accordingly qualified.

The Melon Council (MC) is comprised of the MTC (Melon Technical Council) members and of the MEB (Melon Exposed Businesses) representatives.

Formation of the Melon Technical Council

There were a number of considerations to make when deciding on the makeup of this body. We evaluated the possibility of having token holders elect the council, but concluded that this is still susceptible to the eventual plutocracy outlined above. Instead, we aim at promoting a skill-based, meritocratic system.

The Melon Technical Council (MTC) will initially be appointed by the Melonport AG team, prior to the main net v1.0 deployment of Melon in February 2019. That means that Melonport AG will initially allocate an odd number of seats on the MTC, and ensure that those seats are occupied by diverse actors, with the right set of skills, expertise and incentives.

Subsequent to that, the MTC will grow organically by holding a Melon Council two-thirds vote on incoming applications. Incoming applications shall be reviewed by the existing members and approval or rejection shall be communicated with the according rationale.

The MTC is a subset of the Melon Council (MC). MTC members will be joined by MEB members (see MEB section below and MC statutes) to form the Melon Council. It is expected that the Melon Council will evolve in a consortium-like fashion of technically skilled and business exposed parties.

Melon Technical Council inclusions/exclusions

Applicants to the Technical Council shall meet the following criteria [11]:

Provable technical skills and expertise with regards to Melon (its codebase, token economics, ecosystem) and/or existing meaningful contribution to the Melon codebase A declared absence of any conflict of interest with the Melon vision and ecosystem Willingness and ability to dedicate time and resources to the Technical Council activities High evidenced ethical standards, good reputation and compliance with the MC statutes Willingness to reveal their identity The decision-making processes shall be open and transparent to the community. The Melon Council will be responsible for providing context to their decisions to the community. The Melon Council may or may not decide to consult the token holders sentiments on a specific topic.

Melon Council members can be excluded through a majority vote of the MEB (see MEB section and MC statutes)

Fiduciary duties

The Melon Council will be bound by fiduciary duties, guiding principles and MC statutes. This means the MC members will be obligated to act in the best interest of the Melon protocol. Any member violating their fiduciary duties will expose themselves to the revocation of their seat.

If a MC member has a conflict of interest on a specific question, they should inform the rest of the members immediately and abstain from voting on the matter in question.

MTC Incentivization model

The people with the right skills for the MTC are scarce so we need to provide the right incentives structure for the MTC members. We therefore propose to ring-fence x% of the MLN annual inflation pool to reward the MTC members. The MLN tokens received by MTC members will be vested.

Initially this should just cover their costs, but by providing that reward as a percentage of the annual inflation (ie. a proportion of total market cap), we also introduce an incentive to grow the market cap of Melon through adding value to the network.

This has a size self-balancing side effect: as the pools of incentives grows, the MTC can grow. If the pool of incentives decreases (ie. market cap decreases), it is likely that some members of the MTC will resign or be forced out as it will no longer be worth their time and effort. This will in turn grow the asset pool for the remaining members. At the same time, if the number of MTC members becomes too small and the pool of available assets grows, more members will be attracted in.

Note here that only MTC (not MEB representatives) members are eligible for this pool of reward.

Protocol upgrades

When a protocol upgrade is needed (feature addition, bug fixes or security vulnerability), the MTC can either implement the upgrade itself or mandate an external developer or entity to conduct the implementation.

Once the implementation is finished, the new code must be audited by an independent party. If the audit passes and the majority of the MTC agrees, the new contracts can be deployed.

The MTC owns the key of the owner address of the Melon ENS subdomains. The sole power of the MTC lies in their ability to change the ENS subdomain pointers of the Melon smart contracts to the newly deployed contracts. Note here that only the MTC (and not the whole MC!) approval is needed to update the ENS subdomain pointers.

The MTC does not have the power to force an upgrade on the user or to shut down a previous version. The responsibility to run secure and up-to-date code relies solely on the user.

Resource allocation

The Melon protocol aims at providing the right set of incentives to the people maintaining it and developing it. The inflation (whose curve has yet to be defined) will be used solely for this purpose.

The MTC will receive compensation from the inflation pool to cover costs associated with their duties as MTC members (as per MTC incentivization model above). Based on the defined roadmap

and scheduled improvements, the Melon Council will compensate developers and contributors to the Melon protocol. The MC can either post bounties for specific desired features with the associated reward, or accept proposed MIPs by external developers. Projects in the asset management 3.0 industry can also apply for funding from the inflation pool (see Melonomics 1). The MC will then conduct a thorough review on funding requests. Approval of a project requires a 50% majority + 1 vote. The MC should only accept funding requests for projects that are expected to add a net value to the Melon ecosystem as a whole. Network parameters

The Melon Engine mechanisms were recently presented in Melonomics 2. The MC will be provided with clear guidelines by the Melonport AG team with regards to adjustment of the amgu price (or Melon gas price). It will also be provided with the right tooling and framework to help make informed decisions.

It is hard to predict today how often this variable will need to be adjusted, but we see two main factors that will drive this change:

Burn rate, directly linked to usage levels of the network; as usage of the network goes up, the Melon gas price should go down to maintain a balanced and healthy burn rate. Market conditions on the MLN/ETH and ETH/USD pair More details on this part will be provided in a future blog post.

Rotating leads and responsibilities

We believe a group of people are more efficient when each person is held responsible and accountable.

The MC should elect a Chair and a Vice-Chair at the beginning of each year. The Chair and Vice-Chair will be responsible for the coordination of the MC, its meetings and preparing the agendas.

For the sake of efficiency (and to deter free-rider behavior), we propose a rotating lead system where every year, members can take a leadership role on specific topics such as: audits, features, ecosystem projects, network parameters etc. These may rotate every 6 months if necessary.

2. Developers

The establishment of a technical council should not be seen as limiting contributions to the development of Melon to the members of the technical council. Participation in development is not limited to the MTC and should always be open to the public.

The Melon protocol is entirely open-source, released under the GPL-3.0 license and, as such, any developer is invited to work on feature additions, third-party integrations, bug fixes and submit pull requests. Pull requests will be reviewed by the MTC. However, if a specific pull request is not merged by the MTC, that is not to say it can not be adopted by users.

Developers can apply for funding by submitting proposals to the MC, under the form of Melon Improvement Proposals (MIP).

Developers can also decide to implement a specific feature or improvement for which the MC shows strong interest. In that case the developer will receive the funding corresponding to the task at hand.

Last but not least, developers having already contributed to the codebase, and willing to increase their engagement with Melon should strongly consider applying for a seat at the Melon Technical Council. If you are interested and meet the MTC requirements, reach out to mtc@melonport.com.

3. Users

To be clear, the proposed Melon governance model is a user-centric model. It ensures users have permissionless access to a secure asset management protocol, and are protected from malevolent actors in the network. At the same time, users have the option to benefit from continuous innovation and improvements on top of the protocol, safeguarded by the thorough checks and analysis of the Melon Council bound by fiduciary duties. Indeed, the Melon Council is responsible for taking decisions preserving the interest of the users of the network.

The users always remain in full control, and is the sole decision maker with regards to the software they are running.

Neither the MC nor the token holders can impact the smart contract code used by a fund manager. The fund manager must take a voluntary action in order to upgrade to new versions of the code, and the fund's investors are free to instantly redeem their shares if they are not happy with the version of the code being used. The fund manager is never forced to use a new version of the code they may or may not feel comfortable with. Users take full responsibility to upgrade from code that may contain security vulnerabilities.

As a result, the convergence of users towards a specific version of the Melon protocol shall give a strong indication to the MC of their alignment with the users' sentiments and needs. Although the MTC owns and controls the ENS subdomains pointing to the latest contracts, the users are the ones truly deciding which version to base their business upon, which constitutes a strong signal to the community. This is further enabled by the unstoppable character of smart contracts (once deployed, the Melon contracts can not be taken back by the deployer).

However, users will be highly encouraged to always use the latest versions of the Melon protocol, as security vulnerabilities can be discovered and will be fixed in protocol upgrades. Users are also encouraged to conduct their own analysis, audit and review of the contracts they intend to use. The ultimate choice and responsibility relies solely on the user.

4. Melon Exposed Businesses (MEB)

In order to guarantee that the voice of the users are heard and considered, it was deemed sensible to encourage the formation of the Melon Exposed Businesses.

The intent is to unify the voices of users whose businesses heavily rely on Melon and its future development. This includes fund managers with a minimum threshold of assets managed on Melon (to be defined), or applications and projects utilising the Melon protocol.

We aim for the MTC and MEB to maintain a close relationship and preserve a healthy feedback loop. The MTC will be required to address the concerns raised by the MEB, and both entities should work together at defining the pressing needs of the users of the network.

Another important function of the MEB is to balance power by checking the decisions made by the MTC, and informing users about potential suspicious activity. The MEB will be able to elect its

delegates to represent their interest on the Melon Council (see MC statutes). It is also possible for the MEB to vote on the exclusion of a Melon Council member violating the MC statutes (through a two-third majority vote).

In the future, as more businesses build on Melon, it is expected that the MEB will grow in size. The Multichain Asset Managers Association (MAMA) as a trade body may be able to facilitate the organization of such a body.

Rather than a static model, governance is a dynamic and complex process. The above model is a reflection of our current thinking, and it is what we would like to start with for the first few years, but it is expected to change and evolve over time. The Melon Council should also stay aware of the future developments in decentralized governance, as they might decide at some point in the future to migrate towards a new model.

Melon Engine: buy-and-burn model

For context, please read Melonomics 2: https://medium.com/melonport-blog/melonomics-part-2-the-melon-engine-48bcb0dae65.

The Melon Engine is a contract that collects fees paid in ETH for asset management gas units by the users, buys MLN at a premium to market with the collected ETH and burns the commensurate quantity of MLN. The Melon Engine mechanisms enable the alignment of the token value with the usage of the network, while avoiding the pitfalls of high-velocity token models.

Perspective of a user interacting with the Melon protocol

When a user interacts with an amgu (asset management gas unit) payable function on the Melon protocol, it pays asset management gas in ETH to the Melon network. We are still in the process of defining the functions that are amgu payable. As it stands, the following functions are amgu payable: `setupFund`, `requestInvestment`, `executeRequest` and `claimFees`.

A modifier `amguPayable` is added to certain functions within the Melon protocol. That modifier will apply the logic which grants the user the ability to pay asset management gas.

The total asset management gas paid is equal to: Number of amgu consumed * amgu price

The initial amgu price will be set in MLN terms by Melonport AG before deployment in February. Thereafter, it will then be in the hands of the Melon Council to adjust the amgu price as necessary. According to our modeling, the Melon gas price (amgu price) should not need frequent adjustment. The 2 scenarios where we can expect an adjustment would be:

- Overnight spike in network usage
- Extreme volatility on MLN/USD

The Melon gas price should always be kept at levels which make the Melon protocol competitive to other asset management alternatives. At the same time, the costs incurred must be sufficient to incentivize maintainers and developers compensated in MLN tokens.

Melon Engine mechanics

The ETH from the amgu fees paid by users is sent to the Melon Engine.

The Melon Engine contract is a unidirectional liquidity contract that sells ETH and buys MLN at a premium over the market price (market price is retrieved through Kyber network).

The Melon Engine maintains consecutive 1 month (60*60*24*30 seconds) freeze periods. ETH received by the Melon Engine during a freeze period is held frozen in the Melon Engine until that freeze period ends. After 1 month, the function `thaw()` is publicly callable. Calling that function will render all frozen ETH liquid and available for external actors to buy ETH in exchange for MLN.

As soon as the Melon Engine sells ETH and buys MLN, it burns the acquired MLN.

The premium schedule provided by the Melon Engine is as follows: - Ether in the Melon Engine < 1 ETH, premium = 0% - 1≤ Ether in the Melon Engine < 5, premium = 5% - 5 ≤ Ether in the Melon Engine < 10, premium = 10% - Ether in the Melon Engine ≥ 10, premium = 15%

Perspective of an actor interacting with the Melon Engine

The Melon Engine premium is exclusive to Melon funds. Any Melon fund is eligible to call the `sellAndBurnMln` function on the Melon Engine, and may therefore benefit from the premium offered.

Use case: As soon as ETH in the Melon Engine are liquid, a Melon fund can discretionarily sell MLN to the Melon Engine for ETH. This is done through the Melon Engine exchange adapter.

Assume a Melon fund owns 10 MLN. - Through the Melon Engine exchange adapter, the Melon fund calls `sellAndBurnMln(10)`. - The function retrieves the amgu price from the Melon Engine (i.e. number of ETH the Melon Engine is willing to sell for 1 MLN, premium included). - The 10 MLN tokens the Melon fund sells are transferred to the Melon Engine contract. - The Melon Engine sends (10 * amgu price * (1+premium)) ETH to the Melon fund in exchange for the MLN tokens sold by the Melon fund. - The Melon Engine burns the MLN tokens collected in that transaction.

AmguConsumer.sol

Description

This contract defines a function modifier which is used with functions where AMG (Asset Management Gas) is designed to be charged. Placing the modifier on a function affects the calculation of the functions gas usage and resulting AMGU charge. The resulting payment and excess refund to the sender is also handled by the functionality of the modifier. The contract also defines three helper functions to retrieve the contract addresses of the Melon Engine, MLN token and the price source.

Inherits from

DSMath (link)

On Construction

None.

Structs

None.

Enums

None.

Modifiers

`modifier amguPayable(bool deductIncentive)`

This modifier determines the initial gas amount. The functionality in the function implementing this modifier is then executed. Thereafter, the AMGU (Asset Management Gas Unit) price and the MLN token market price is retrieved from the Melon Engine and the price feed, respectively. The modifier calculates the amount of ETH required for the payment of AMGU. Transactions with insufficient ETH amounts to pay the AMGU are reverted. ETH due as AMGU is paid to the Melon Engine and any excess ETH above the Ethereum gas costs and AMGU costs are refunded to the `msg.sender`.

Events

None.

Public State Variables

None.

Public Functions

`function engine() view returns (address)`

This public view function returns the address of the Melon Engine contract.

`function mlnToken() view returns (address)`

This public view function returns the address of the MLN token contract.

`function priceSource() view returns (address)`

This public view function returns the address of the price feed contract.

`function registry() view returns (address)`

This public view function returns the address of the registry contract.

Engine.sol

Description

This contract is a Melon fund exclusive, unidirectional exchange for Melon funds to exchange MLN

tokens for ETH at a premium to market prices. The Melon Engine then reduces the total supply of the MLN token by burning all received MLN tokens.

Inherits from

DSMath, DSAuth (link)

On Construction

The constructor for this contract requires the parameters `_delay` and `_postDeployOwner`. The parameter `_delay` specifies the number of delay seconds, setting the `THAWING_DELAY` state variable to this value. The constructor sets the `lastThaw` state variable to the current `block.timestamp` value. Finally, the engine contract's owner is set to the `_postDeployOwner` address provided.

Structs

None.

Enums

None.

Modifiers

None.

Events

```
event RegistryChange(address registry)
```

This event is triggered when the Version's Registry is set through the `setRegistry()` function, logging the `Registry` address.

```
event SetAmguPrice(uint amguPrice)
```

This event is triggered when the `setAmguPrice()` function is called, logging the modified `amguPrice`.

```
event Thaw(uint amount)
```

This event is triggered when the `thaw()` function is called, logging the thawed ETH `amount`.

event AmguPaid(uint amount)

This event is triggered when the `payAmguInEther()` function is called, logging the ETH `amount`.

event Burn(uint amount)

This event is triggered when the `sellAndBurnMln()` function is called, logging the burned MLN `amount`.

Public State Variables

`uint public frozenEther`

This public state variable represents the quantity of ETH held by the Melon Engine which is currently frozen.

`uint public liquidEther`

This public state variable represents the quantity of ETH held by the Melon Engine which is not currently frozen.

`uint public lastThaw`

This public state variable represents the timestamp of the last, previous frozen ETH thaw.

`uint public THAWING_DELAY`

This public state variable represents the number of seconds required before frozen ETH is thawed.

`BurnableToken public mlnToken`

This public state variable represents MLN token contract which implements token burn functionality.

`PriceSourceInterface public priceSource`

This public state variable represents the Version's PriceSource contract.

`Registry public registry`

This public state variable represents the Version's Registry contract.

`uint public MLN_DECIMALS = 18`

This public state variable defines the decimal precision of the MLN token. It is set to the constant, 18.

`uint public amguPrice`

A public state variable specifying the current setting for the Asset Management Gas Unit (AMGU) price. This setting can be changed at the discretion of the Melon Technical Council (MTC).

`uint public totalEtherConsumed`

This public state variable keeps a running total of all ETH sent to the engine contract.

`uint public totalAmguConsumed`

This public state variable keeps a running total of all amgu consumed by the engine contract.

`uint public totalMlnBurned`

This public state variable keeps a running total of all MLN burned by the engine contract.

Public Functions

`function setRegistry(address _registry) external auth`

This external function requires that the caller is the `owner` or the current contract. The function sets the `registry` state variable to the contract with the address specified by the parameter provided and further sets the `priceSource` and `mlnToken` state variables. Finally the function emits the `RegistryChange()` event, logging `registry`.

`function setAmguPrice(uint _price) external auth`

This external function requires that the caller is the `owner` or the current contract. The function sets the `amguPrice` state variable to the value specified by the parameter provided. Finally the function emits the `SetAmguPrice()` event, logging `amguPrice`.

`function getAmguPrice() public view returns (uint)`

This public view function returns the value of the `amguPrice` state variable.

`function premiumPercent() public view returns (uint)`

This public view function determines the market price premium to be offered when the Melon Engine buys MLN tokens. Please see the premium schedule above. The function returns an integer representing the premium percentage, e.g. a return value of `5'' represents` 5%".

`function payAmguInEther() external payable`

This external payable function is called while sending ETH. The function requires that the function caller must either be the fund managed by msg.sender or the fund factory used in the creation of the manager's fund. The function accepts the ETH amgu payment and increments the `frozenEther` state variable with the ETH value sent in the function call transaction. Finally, the function emits the `AmguPaid()` event, logging `amount` paid.

`function thaw() external`

This external function allocates frozen ETH (`frozenEther`) to liquid ETH (`liquidEther`) after the specified `THAWING_DELAY`. The function requires that sufficient time (minimally the `THAWING_DELAY`) has passed since the previous thaw. The function then requires that the quantity of frozen ETH is positive. The function then sets the `lastThaw` state variable to the current `block.timestamp`. The `liquidEther` state variable is incremented with the `frozenEther` state variable value, the `Thaw()` event is emitted, logging `frozenEther` value and finally, the `frozenEther` state variable is set to `` `0''. Note: this function is independently and externally called. The `lastThaw `` is only reset when this function is called, regardless of whether the previous `THAWING_DELAY` has expired.

`function enginePrice() public view returns (uint)`

This public view function returns the premium-adjusted ETH/MLN rate based on the quantity of ETH accumulated by the Melon Engine. The function first retrieves the MLN token price denominated in ETH. The calculated ETH-quantity scaled premium is then added to the market

price and returned.

`function ethPayoutForMlnAmount(uint mlnAmount) public view returns (uint)`

This public view function returns the premium-adjusted quantity of ETH to be delivered to the Melon fund in return for selling the `mlnAmount` quantity of the MLN token.

`function sellAndBurnMln(uint mlnAmount) external`

This external function requires that the fund is the msg.sender, ensuring that only Melon funds may transact with the Melon Engine and benefit from the MLN token premium. The function then requires that approved MLN tokens be transferred from the Melon fund to the Melon Engine. The function calls `ethPayoutForMlnAmount()` to get the current quantity of ETH to send to the transacting Melon fund. This quantity is required to be positive and that a sufficient quantity of liquid ETH is held by the Melon Engine. The Melon Engine transfers the ETH to the transacting Melon fund and burns the received MLN tokens. Finally, the function emits the `Burn()` event, logging `mlnAmount` burned.

Pricing

General

The Melon funds receive current pricing data from the Kyber Network, which in turn aggregates Kyber Reserve Managers providing markets for individual asset tokens. Asset prices are derived from the best price offered among participating reserve managers.

The PriceSource interface definition is provided below. KyberPriceFeed.sol implements the PriceSource.i.sol interface.

PriceSource.i.sol

Description

The PriceSource.i.sol is an interface definition intended to generalize the implementation of any concrete price source provider.

Interface Functions

`function getPriceInfo(address _asset) view returns (uint price, uint decimals)`

This public view function returns the asset token price and the asset token decimal precision given the `_asset` address parameter provided.

`function getInvertedPriceInfo(address ofAsset) view returns (uint price, uint decimals)`

This public view function returns the asset token price provided by the `ofAsset` address parameter in terms of the `ofAsset` asset and decimal precision of the asset token address provided.

`function getQuoteAsset() public view returns (address)`

This public view function returns the address of asset token contract which was configured to be the base- or quote asset of the fund. The quote asset is the asset in which the fund's value is denominated.

`function hasValidPrice(address) public view returns (bool)`

This public view function returns a boolean indicating whether the asset represented by the address parameter provided is valid. A return value of `true` indicates that the asset has a valid price. A return value of `false` indicates that the asset does not have a valid price.

`function hasValidPrices(address[]) public view returns (bool)`

This public view function returns a boolean indicating whether all of the assets represented by the `address[]` array parameter provided are valid. A return value of `true` indicates that all asset tokens have valid prices. A return value of `false` indicates that one or more of the asset tokens do not have a valid price.

`function getPrice(address _asset) public view returns (uint price, uint timestamp)`

This public view function returns the price and price timestamp of the asset token given by the asset token address provided. The timestamp represents the time of the price quote.

```
function getPrices(address[] _assets) public view returns (uint[] prices, uint[] timestamps)
```

This public view function returns an array of prices and a corresponding array of price timestamps of the asset tokens given by the array of asset token addresses provided. The timestamps represent the times of the price quote.

```
function getReferencePriceInfo(address _base, address _quote) public view returns (uint referencePrice, uint decimal)
```

This public view function takes two address parameters: the base asset (the object of the pricing information) and the quote asset (the asset in which the price is denominated). The function returns the asset price provided by the `ofBase` address parameter in terms of the quote asset, and the specified asset token's `decimals` property.

```
function getOrderPriceInfo(address sellAsset, address buyAsset, uint sellQuantity, uint buyQuantity) public view returns (uint orderPrice)
```

This public view function returns a price given the provided sell asset and corresponding quantity, and the buy asset and corresponding quantity.

```
function existsPriceOnAssetPair(address sellAsset, address buyAsset) public view returns (bool isExistent)
```

This public view function returns a boolean indicating whether a recent price exists for the asset pair provided by the `sellAsset` and `buyAsset` address parameters.

KyberPriceFeed.sol

Description

The KyberPriceFeed contract provides the interface from the version-specific Melon platform to the Kyber network for the purposes of receiving pricing data.

Inherits from

PriceSourceInterface, DSThing (link)

On Construction

The `KyberPriceFeed` contract requires the following parameters on construction:

`address ofRegistrar` - The address of the Version's Registrar contract. `address ofKyberNetworkProxy` - The address of the Kyber network proxy contract. `uint ofMaxSpread` - The the maximum spread between bid- and ask prices for the price to be considered valid. `address ofQuoteAsset` - The address of the asset token contract which is the fund's base- or quote asset. All asset prices will be denominated or based in this asset token across all funds for the purposes of pricing.

These parameters are used to set the the following public state variables on the contract:

`KYBER_NETWORK_PROXY MAX_SPREAD QUOTE_ASSET REGISTRY`

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

```
address public KYBER_NETWORK_PROXY
```

The address of the Kyber network proxy contract.

```
address public QUOTE_ASSET
```

The address of the asset token contract which is the fund's base- or quote asset.

```
Registry public REGISTRY
```

A public state variable which is the Registry contract of version under which the fund was deployed.

```
uint public MAX_SPREAD
```

A public state variable which represents the maximum spread between derived bid- and ask prices for specific asset pairs. `MAX_SPREAD` represents the maximum acceptable tolerance for a price to be valid. `MAX_SPREAD` is specified as a percentage and formatted in 10^18 terms. For example, 1.0% (0.01) would be represented 1x10^16 or 10000000000000000.

```
address public constant KYBER_ETH_TOKEN = 0x00eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
```

A constant public state variable which represents the token contract address of the ETH asset token.

`uint public constant KYBER_PRECISION = 18`

A constant public state variable which represents the divisibility precision of asset tokens...

`uint public constant VALIDITY_INTERVAL = 2 days`

A constant public state variable which represents the maximum validity of price feed prices and is set to ``2 days''.

`uint public lastUpdate`

An integer representing the time of the last price feed update.

`mapping (address ⇒ uint) public prices`

A public mapping associating an asset token contract address to the most recent price.

Public Functions

`function update()`

This public function ensures that is can only be called by the `REGISTRY` owner. The function updates the `prices` mapping and the `lastUpdate` state variable. A price of ``0'' indicates an invalid price.

`function getQuoteAsset() view returns (address)`

This public view function returns the address of asset token contract which was configured to be the base- or quote asset of the fund. The quote asset is the asset in which the fund's value is denominated.

`function getPrice(address _asset) view returns (uint price, uint timestamp)`

This public view function returns the price and price timestamp of the asset token given by the asset token address provided. The price integer returned is formatted as the product of the exchange price and ten to the power of the asset's specified decimals property. The timestamp represents the time of the price quote.

`function getPrices(address[] _assets) view returns (uint[], uint[])`

This public view function returns an array of prices and a corresponding array of price timestamps of the asset tokens given by the array of asset token addresses provided. The price integers returned are formatted as the product of the exchange price and ten to the power of the asset's specified decimals property. The timestamps represent the times of the price quote.

`function hasValidPrice(address _asset) view returns (bool)`

This public view function returns a boolean indicating whether the asset represented by the `_asset` parameter provided is recent. The function first requires that the asset is registered in the registry. Passing the address of the quote asset will return `true`. If the price provided by the Kyber reserve manager is not 0, the price is assumed to be recent and will return `true`.

`function hasValidPrices(address[] _assets) view returns (bool)` This public view function returns a boolean indicating whether the assets represented by the `_assets` address array parameter provided are all recent. The function requires that all assets passed in address array `_assets`

parameter are registered in the registry. If any single asset in the array of asset addresses is not recent, the function will return `false`.

```
function getPriceInfo(address ofAsset) view returns (bool isRecent, uint price, uint assetDecimals)
```

This public view function calls the `getReferencePriceInfo()` function and returns a boolean indicating the validity of the price, the asset price provided by the `ofAsset` address parameter in terms of the quote asset, and the specified asset token's `decimals` property.

```
function getRawReferencePriceInfo(address _baseAsset, address _quoteAsset) view returns (bool isValid, uint referencePrice, uint decimals)
```

This public view function takes the `_baseAsset` and `_quoteAsset` token contract addresses and returns a boolean indicating price validity, the reference price and the asset token's decimal precision.

```
function getInvertedPriceInfo(address ofAsset) view returns (bool isRecent, uint invertedPrice, uint assetDecimals)
```

This public view function calls the `getReferencePriceInfo()` function and returns a boolean indicating the validity of the price, the asset price provided by the `ofAsset` address parameter in terms of the `ofAsset` asset, and the specified asset token's `decimals` property.

```
function getReferencePriceInfo(address _baseAsset, address _quoteAsset) view returns (uint referencePrice, uint decimals)
```

This public view function takes two address parameters: the base asset (the object of the pricing information) and the quote asset (the asset in which the price is denominated). [CHECK: function will we re-worked...]

```
function getOrderPriceInfo(address sellAsset, address buyAsset, uint sellQuantity, uint buyQuantity) view returns (uint orderPrice)
```

This public view function returns a price given the provided sell asset and corresponding quantity, and the buy asset and corresponding quantity.

```
function existsPriceOnAssetPair(address sellAsset, address buyAsset) view returns (bool isExistent)
```

This public view function returns a boolean indicating whether a recent price exists for the asset pair provided by the `sellAsset` and `buyAsset` address parameters.

```
function getKyberMaskAsset(address _asset) returns (address)
```

This public function returns the address of the Kyber Network representation contract adddress of the asset token contract address provided. If the address provided is ETH, the native token, the function returns `KYBER_ETH_TOKEN`; otherwise `_asset` is returned.

```
function getKyberPrice(address _baseAsset, address _quoteAsset) public view returns (bool isValid, uint kyberPrice)
```

This public view function takes the `_baseAsset` and `_quoteAsset` token contract addresses and returns a boolean indicating price validity and the average expected current Kyber Network price.

```
function convertQuantity(uint fromAssetQuantity, address fromAsset, address toAsset) public view returns (uint)
```

This public view function calculates and returns the quantity of the `toAsset` token which has a current value equal to the `fromAssetQuantity` of the `fromAsset` token.

```
function getLastUpdate() public view returns (uint)
```

This public view function returns the blocktime timestamp of the last price update, `lastUpdate`.

Registry

General

The Registry contract is a singleton contract for all Version contracts. Each specific Melon protocol Version deploys funds generated of that specific version.

Registry.sol

Description

The Registry contract stores, manages and provides functionality to maintain all relevant data pertaining to individual asset tokens and individual exchanges eligible for investment and operation within the Melon Protocol. The contract provides functionality to define, add, update or remove asset tokens and exchanges.

Inherits from

DSAuth (link)

On Construction

None.

Structs

`Asset`

Member variables:

`bool exists` - A boolean to conveniently and clearly indicate existence in a mapping. `string name` - The human-readable name of the asset token. `string symbol` - The human-readable symbol of the asset token. `uint decimals` - The divisibility precision of the token. `string url` - The URL for extended information of the asset token. `uint reserveMin` - An integer representing the Kyber Network reserve minimum. `uint[] standards` - An array of integers representing EIP standards to which this asset token conforms. `bytes4[] sigs` - An array of asset token function signatures which have been whitelisted.

This struct stores all relevant information pertaining to the asset token.

`Exchange`

Member variables:

`bool exists` - A boolean to conveniently and clearly indicate existence in a mapping. `address adapter` - The address of the exchange's corresponding adapter contract registered. `bool takesCustody` - A boolean indicating the custodial nature of the exchange. `bytes4[] sigs` - An array of exchange function signatures which have been whitelisted.

This struct stores all relevant information pertaining to the exchange.

Version

Member variables:

`bool exists` - A boolean to conveniently and clearly indicate existence in a mapping. `bytes32 name` - A 32 byte value type to represent the name of the Version.

This struct stores all relevant information pertaining to the Version.

Enums

None.

Modifiers

`modifier only_version()`

A modifier which requires `msg.sender` be a Version contract.

Events

`AssetUpsert()`

`address indexed asset` - The address of the asset token contract registered. `string name` - The human-readable name of the asset token. `string symbol` - The human-readable symbol of the asset token. `uint decimals` - The divisibility precision of the token. `string url` - The URL for extended information of the asset token. `uint reserveMin` - An integer representing the Kyber Network reserve minimum. `uint[] standards` - An array of integers representing EIP standards to which this asset token conforms. `bytes4[] sigs` - An array of asset token function signatures which have been whitelisted.

This event is triggered when an asset is added or updated in `registeredAssets`. The parameters listed above are provided with the event.

`event ExchangeUpsert()`

`address indexed exchange` - The address of the exchange contract registered. `address indexed adapter` - The address of the exchange's corresponding adapter contract registered. `bool takesCustody` - A boolean indicating the custodial nature of the exchange. `bytes4[] sigs` - An array of exchange function signatures which have been whitelisted.

This event is triggered when an exchange is added or updated in `registeredExchanges`. The parameters listed above are provided with the event.

`event AssetRemoval()`

`address indexed asset` - The address of the registered asset token contract to be removed.

This event is triggered when an asset is removed from the `registeredAssets` array state variable.

The parameter listed above are provided with the event.

`event ExchangeRemoval()`

`address indexed exchange` - The address of the registered exchange contract to be removed.

This event is triggered when an exchange is removed from the `registeredExchanges` array state variable. The parameter listed above are provided with the event.

`event VersionRegistration()`

`address indexed version` - The address of the Version registered.

This event is triggered when a Version is registered. The parameter listed above are provided with the event.

`event PriceSourceChange()`

`address indexed priceSource` - The address of the new price source.

This event is triggered when a price source is changed. The parameter listed above are provided with the event.

`event MlnTokenChange()`

`address indexed mlnToken` - The address of the new MLN token contract.

This event is triggered when the MLN token is migrated to a new contract. The parameter listed above are provided with the event.

`event NativeAssetChange()`

`address indexed nativeAsset` - The address of the new native asset token contract.

This event is triggered when the native asset token is migrated to a new contract. The parameter listed above are provided with the event.

`event EngineChange()`

`address indexed engine` - The address of the new Melon Engine contract.

This event is triggered when the Melon Engine is migrated to a new contract. The parameter listed above are provided with the event.

Public State Variables

`mapping (address ⇒ Asset) public assetInformation`

This public state variable mapping maps an asset token contract `address` to an `Asset` strut containing the asset information described above.

`address[] public registeredAssets`

This public state variable is an array of addresses which stores each asset token contract `address` which is registered.

`mapping (address ⇒ Exchange) public exchangeInformation`

This public state variable mapping maps an exchange contract `address` to an `Exchange` strut containing the exchange information described above.

`address[] public registeredExchangeAdapters`

This public state variable is an array of addresses which stores each exchange adapter contract `address` which is registered.

`mapping (address ⇒ Version) public versionInformation`

This public state variable mapping maps a Version contract `address` to a `Version` strut containing the Version information described above.

`address[] public registeredVersions`

This public state variable is an array of addresses which stores each Version contract `address` which is registered.

`mapping (address ⇒ bool) public isFeeRegistered`

This public state variable mapping maps a Fee contract `address` to a boolean indicating that the Fee contract is registered.

`mapping (address ⇒ address) public fundsToVersions`

This public state variable mapping maps a Version contract `address` to a Melon fund address.

`mapping (bytes32 ⇒ bool) public versionNameExists`

This public state variable mapping maps the Version name to a boolean indicating the Version name existence.

`mapping (bytes32 ⇒ address) public fundNameHashToOwner`

This public state variable mapping maps the hash of the fund name to the fund owner. A `bytes32` type is used, as dynamic-length types cannot be used as keys in mappings and a hash of the fund name allows for names longer than 32 bytes.

`uint public constant MAX_REGISTERED_ENTITIES = 20`

This public constant state variable represents the maximum quantity of registered entities and is set to ``20''. This constant applies to Exchanges registered and Assets registered.

`uint public constant MAX_FUND_NAME_BYTES = 66`

This public constant state variable represents the maximum Melon fund name size in bytes. The maximum is set to 66 bytes.

`address public priceSource`

This public state variable is the address of the current, active price source contract.

`address public mlnToken`

This public state variable is the address of the Melon token contract.

`address public nativeAsset`

This public state variable is the address of the native asset token contract.

`address public engine`

This public state variable is the address of the Melon Engine contract.

`address public ethfinexWrapperRegistry`

This public state variable is the address of the Ethfinex Wrapper Registry contract.

`uint public incentive = 10 finney`

This public state variable defines the incentive amount in ETH. The variable is set to 10 finney.

Public Functions

`function registerAsset( address _asset, string _name, string _symbol, string _url, uint _reserveMin, uint[] _standards, bytes4[] _sigs ) external auth`

This external function requires that the caller is the `owner` or the current contract. It then requires that the asset token's information not be previously registered. The asset token's address is then appended to the `registeredAssets` array state variable. The function then registers the following information for a specific asset token within the Registry contract as per the following parameters:

`address _asset` - The address of the asset token contract to be registered. `string _name` - The human-readable name of the asset token. `string _symbol` - The human-readable symbol of the asset token. `string _url` - The URL for extended information of the asset token. `uint reserveMin` - An integer representing the Kyber Network reserve minimum. `uint[] _standards` - An array of integers representing EIP standards to which this asset token conforms. `bytes4[] _sigs` - An array of asset token function signatures which have been whitelisted.

Finally, the function ensures that the asset token's information exists in the `assetInformation` mapping state variable.

`function registerExchangeAdapter( address _exchange, address _adapter, bool _takesCustody, bytes4[] _sigs ) external auth`

This external function requires that the caller is the `owner` or the current contract. It then requires that the exchange's information not be previously registered. The exchange's address is then appended to the `registeredExchanges` array state variable. The function then registers the following information for a specific exchange within the Registry contract as per the following parameters:

`address _exchange` - The address of the exchange contract to be registered. `address _adapter` - The address of the exchange's corresponding adapter contract to be registered. `bool _takesCustody` - A boolean indicating the custodial nature of the exchange. `bytes4[] _sigs` - An array of exchange function signatures which have been whitelisted.

Finally, the function ensures that the exchange's information exists in the `exchangeInformation` mapping state variable.

`function updateAsset( address _asset, string _name, string _symbol, uint _decimals, string _url, uint _reserveMin, uint[] _standards, bytes4[] _sigs ) auth`

This function requires that the caller is the `owner` or the current contract. It then requires that the asset token's information be previously registered. The function then updates the following information for a specific asset token within the Registry contract as per the following parameters:

`address _asset` - The address of the asset token contract to be registered. `string _name` - The human-readable name of the asset token. `string _symbol` - The human-readable symbol of the asset token. `uint _decimals` - The divisibility precision of the token. `string _url` - The URL for extended information of the asset token. `uint reserveMin` - An integer representing the Kyber Network reserve minimum. `uint[] _standards` - An array of integers representing EIP standards to which this asset token conforms. `bytes4[] _sigs` - An array of asset token function signatures which have been whitelisted.

Finally, the function emits the `AssetUpsert` event along with the parameters listed above.

`function updateExchange( address _exchange, address _adapter, bool _takesCustody, bytes4[] _sigs ) auth`

This function requires that the caller is the `owner` or the current contract. It then requires that the exchange's information be previously registered. The function then updates the following information for a specific exchange within the Registry contract as per the following parameters:

`address _exchange` - The address of the exchange contract to be registered. `address _adapter` - The address of the exchange's corresponding adapter contract to be registered. `bool _takesCustody` - A boolean indicating the custodial nature of the exchange. `bytes4[] _sigs` - An array of exchange function signatures which have been whitelisted.

Finally, the function emits the `ExchangeUpsert` event along with the parameters listed above.

`function removeAsset(address _asset, uint _assetIndex) auth`

This function requires that the caller is the `owner` or the current contract. The function requires the following parameters:

`address _asset` - The address of the asset token contract to be removed. `uint _assetIndex` - The index of the asset provided in the registerAssets array state variable.

The function then requires that the asset information exists and that the asset is registered. The function then deletes the asset's entries from the `assetInformation` mapping state variable and the `registeredAssets` array state variable, while also maintaining the `registeredAssets` array. The function ensures that the asset's information entry no longer exists and finally emits the `AssetRemoval()` event with the asset's token contract address.

`function removeExchange(address _exchange, uint _exchangeIndex) auth`

This function requires that the caller is the `owner` or the current contract. The function requires the following parameters:

`address _exchange` - The address of the exchange contract to be removed. `uint _exchangeIndex` - The index of the exchange provided in the registerAssets array state variable.

The function then requires that the exchange information exists and that the exchange is registered. The function then deletes the exchange's entries from the `exchangeInformation` mapping state variable and the `registeredExchanges` array state variable, while also maintaining the `registeredExchanges` array. The function ensures that the exchange's information entry no longer exists and finally emits the `ExchangeRemoval()` event with the exchange's contract address.

`function getName(address _asset) view returns (string)`

This public view function retrieves the asset token `name` for the registered asset address provided.

`function getSymbol(address _asset) view returns (string)`

This public view function retrieves the asset token `symbol` for the registered asset address provided.

`function getDecimals(address _asset) view returns (uint)`

This public view function retrieves the asset token `decimals` for the registered asset address provided. `decimals` specifies the divisibility precision of the token.

`function assetIsRegistered(address _asset) view returns (bool)`

This public view function indicates whether the address provided is that of a registered asset token. A return value of `true` indicates that the asset is registered. A return value of `false` indicates that the asset is not registered.

`function getRegisteredAssets() view returns (address[])`

This public view function returns an array of all registered asset token addresses.

`function assetMethodIsAllowed(address _asset, bytes4 _sig) external view returns (bool)`

This external view function returns a boolean indicating whether a specific asset token function is whitelisted given the provided asset token address and the function's signature hash. A return value of `true` indicates that the function is whitelisted. A return value of `false` indicates that the function is not whitelisted.

`function exchangeIsRegistered(address _exchange) view returns (bool)`

This public view function returns a boolean indicating whether a specific exchange is registered. A return value of `true` indicates that the exchange is registered. A return value of `false` indicates that the exchange is not registered.

`function getRegisteredExchanges() view returns (address[])`

This public view function returns an array of all registered exchange addresses.

`function getExchangeInformation(address _exchange) view returns (address, bool)`

This public view function returns the corresponding exchange adapter contract address and the exchange's boolean indicator `takesCustody` given the provided exchange contract address.

`function getExchangeFunctionSignatures(address _exchange) view returns (bytes4[])`

This public view function returns an array of whitelisted exchange function signatures

corresponding to the provided exchange contract address.

`function exchangeMethodIsAllowed(address _exchange, bytes4 _sig) returns (bool)`

This public view function returns a boolean indicating whether a specific exchange function is whitelisted given the provided exchange address and the function's signature hash. A return value of `true` indicates that the function is whitelisted. A return value of `false` indicates that the function is not whitelisted.

`function registerVersion(address _version, bytes32 _name) auth`

This function requires that the caller is the `owner` or the current contract. The function sets the `versionInformation` mapping `exists` to `true`, `name` to `_name` and pushes the Version address on to the `registeredVersions` array. The `versionNameExists` mapping for this Version name is set to `true`. Finally, the `VersionRegistration()` event is emitted, logging the Version address. Versions cannot be removed from the registry.

`function setPriceSource(address _priceSource) auth`

This function requires that the caller is the `owner` or the current contract. The function sets the `priceSource` state variable to the `_priceSource` parameter value and emits the `PriceSourceChange()` event, logging `_priceSource`.

`function setMlnToken(address _mlnToken) auth`

This function requires that the caller is the `owner` or the current contract. The function sets the `mlnToken` state variable to the `_mlnToken` parameter value and emits the `MlnTokenChange()` event, logging `_mlnToken`.

`function setNativeAsset(address _nativeAsset) auth`

This function requires that the caller is the `owner` or the current contract. The function sets the `nativeAsset` state variable to the `_nativeAsset` parameter value and emits the `NativeAssetChange()` event, logging `_nativeAsset`.

`function setEngine(address _engine) auth`

This function requires that the caller is the `owner` or the current contract. The function sets the `engine` state variable to the `_engine` parameter value and emits the `EngineChange()` event, logging `_engine`.

`function getReserveMin(address _asset) view returns (uint)`

This public view function returns the `reserveMin` for the asset token contract address provided.

`function adapterForExchange(address _exchange) view returns (address)`

This public view function returns the address of the exchange adapter contract given the exchange contract address provided.

`function getRegisteredVersions() view returns (address[])`

This public view function returns an exhaustive array of addresses of all registered Version contracts.

`function isFund(address _who) view returns (bool)`

This public view function returns a boolean indicating whether the address provided is a Melon fund contract.

`function isFundFactory(address _who) view returns (bool)`

This public view function returns a boolean indicating whether the address provided is a FundFactory. The function check the existence of the `_who` address in the `versionInformation` mapping. Note that Version inherits FundFactory.

`function registerFund(address _fund, address _owner, string _name) external only_version`

This external function ensures that `msg.sender` is a Version, as only Versions can register funds. The function requires that the fund name is valid. The function then adds an entry to the `fundsToVersions` mapping, associating `_fund` to `msg.sender`, i.e. the Version address.

`function isValidFundName(string _name) public view returns (bool)`

This public function ensures that the fund name provided does not exceed `MAX_FUND_NAME_BYTES` nor contain restricted characters. Legal characters are `0-9'', a-z", A-Z'', " ,"-,''.,"_"` and ``*".

`function canUseFundName(address _user, string _name) public view returns (bool)`

This public function ensures that the fund name provided adheres to the rules set forth in `isValidFundName()`, and that the name is not already in use by a fund or is used by the provided `_user` address (to enable fund name reuse across Versions by the same `_user`).

`function reserveFundName(address _owner, string _name) external only_version`

This external function requires that that `msg.sender` is a Version and that `_name` passes all conditions of `canUseFundName()`, then adds `_name` to the `fundNameHashToOwner` mapping.

`function registerFees(address[] _fees) external auth`

This external function requires that the caller is the `owner` or the current contract. The Fee contract addresses provided are then added as entries to the `isFeeRegistered` mapping with the value of `true`.

`function deregisterFees(address[] _fees) external auth`

This external function requires that the caller is the `owner` or the current contract. The Fee contract addresses provided are then deleted from the `isFeeRegistered` mapping.

Version

General

The Version contract is a singleton contract for all funds deployed under a specific Melon protocol version. All Factory contracts are exclusive to the Version contract and cannot be changed once set.

Version.sol

Description

The Version contract inherits from FundFactory and is the single, exclusive FundFactory contract for a specific Melon Protocol version. The Version contract is the contract creator for all funds of the specific Melon Protocol version.

Inherits from

FundFactory, DSAuth, VersionInterface (link)

On Construction

The following contract addresses are provided as parameters:

`address _accountingFactory` - The address of the AccountingFactory contract. `address _feeManagerFactory` - The address of the FeeManagerFactory contract. `address _participationFactory` - The address of the ParticipationFactory contract. `address _sharesFactory` - The address of the SharesFactory contract. `address _tradingFactory` - The address of the TradingFactory contract. `address _vaultFactory` - The address of the VaultFactory contract. `address _policyManagerFactory` - The address of the PolicyManagerFactory contract. `address _registry` - The address of the Registry contract. `address _postDeployOwner` - The address of the contract owner.

These address parameters set the corresponding state variables as defined in the inherited definition of `FundFactory`. The constructor then directly sets the `registry` state variable with the address of the registry contract and calls the `setOwner()` function passing the address of the current contract.

The Version contract assumes the Governance contract is the deployer.

Structs

None.

Enums

None.

Modifiers

None.

Events

None.

Public State Variables

None.

Public Functions

`function shutDownFund(address _hub) external`

This external function first requires that `msg.sender` is the manager for the specified `_hub`. The function then calls the target fund's `shutDownFund()` function.