
Tapio Project



This project demonstrates a basic Hardhat use case. It comes with a sample contract, a test for that contract, and a script that deploys that contract.

Try running some of the following tasks:

```
1 npx hardhat help
2 npx hardhat test
3 REPORT_GAS=true npx hardhat test
4 npx hardhat node
5 npx hardhat run scripts/deploy.ts
```

Goerli Testnet Address:

```
1 constant: 0x07e70721C1737a9D410bcd038BA7e82e8BC19e2a
2 rETHRate: 0xf2dD62922B5f0cb2a72dAeda711018d6F56EEb17
3
4 tapETH: 0x0C68f684324551b4B6Ff6DFc6314655f8e7d761a
5 WTapETH: 0x31CcC35cbcd56B6e8f01E8207B1302f009ABC27c
6
7 stETHSwap: 0x79106c599A6A320DFB0686513631a92ff8343b44
8 rETHSwap: 0x9719443a2BBb5AB61744C1B3C71C2E3527101a91
9
10 application: 0x44A54f1cc211cfCFfE8b83C22f44728F3Fa5004C
11
12 wETHAddress: '0xB4FBF271143F4FBf7B91A5ded31805e42b2208d6'
13 stETHAddress: '0x1643E812aE58766192Cf7D2Cf9567dF2C37e9B7F'
14 rETHAddress: '0x178e141a0e3b34152f73ff610437a7bf9b83267a'
```

rETH staking website: <https://testnet.rocketpool.net/>

Smart Contracts OVERVIEW

The main contracts of Tapio V1.5 are the following:

TapEth : contract of rebase token tapETH

WTapETH: contract of wrapped tapETH

StableAsset: contract of stableswap pool

StableAssetApplication: user contract interface for different stableSwap pools

Contract TapETH

The contract **TapETH** is upgradable and uses the interface IERC20.

Write Methodes

- **proposeGovernance(address _governance)**

This function allows the current governance to set a new governance address.

- **acceptGovernance(address _governance)**

This function allows the pending governance to be activated: to update the governance to the pending governance.

- **addPool(address _pool)**

This function can be executed only by the governance to whitelist a stableSwap pool.

- **removePool(address _pool)**

This function can be executed only by the governance to remove a whitelisted stableSwap pool.

- **transferShares(address _recipient, uint256 _sharesAmount)**

This function allows the caller to transfer **_sharesAmount** shares of tapETH from his address to **_recipient**.

- **transferSharesFrom(address _sender, address _recipient, uint256 _sharesAmount)**

This function allows the spender to transfer **_sharesAmount** shares of tapETH from to **_sender** to **_recipient**.

- **mintShares(address _account, uint256 _tokenAmount)**

This function can be executed by a whitelisted stableSwap pool to mint **_tokenAmount** of tapETH for **_account**.

- **burnShares(uint256 _tokenAmount)**

This function allows the caller to burn **_tokenAmount** of tapETH.

- **burnSharesFrom(address _account, uint256 _tokenAmount)**

This function allows the spender to burn **_tokenAmount** of tapETH from the addresss **_account** .

View Methodes

- **getTotalPooledEther()**

This function returns the total supply of tapETH (uint256).

- **getTotalShares()**

This function returns the total shares of tapETH (uint256).

- **getSharesByPooledEth(uint256_tapETHAmount)**

This function returns the shares of tapETH (uint256) corresponding to `_tapETHAmount` of tapETH.

- ****getPooledEthByShares(uint256 _sharesAmount)****

This function returns the amount of tapETH (uint256) corresponding to 'sharesAmount' shares of tapETH.

- ****setTotalSupply(uint256 _amount)****

This function can be only called by a whitelist stableSwap pool contract to increase the total supply of tapETH by `_amount`.

Contract WTapETH

The contract **WTapETH** is upgradable and inherits from the contract ERC20Permit.

Write Methodes

- ****wrap(uint256 _tapETHAmount)****

This function allows the user to wrap `_ tapETHAmount` of tapETH that consisting in transferring `_tapETHAmount` of tapETH to the smart contract WTapETH and minting the corresponding shares amount in wtapETH.

- ****unwrap(uint256 _wtapETHAmount)****

This function allows the user to unwrap `_wtapETHAmount` of wtapETH that consisting in burning `wtapETHAmount` of wtapETH and sending from the smart contract WTapETH to the caller the corresponding amount of tapETH.

View Methodes

- ****getWtapETHByTapETH(uint256 _tapETHAmount)****

This function returns the amount of wtapETH that corresponds to `_tapETHAmount` of tapETH.

- ****getTapETHByWtapETH(uint256 _wtapETHAmount)****

This function returns the amount of tapETH that corresponds to `_wtapETHAmount` of wtapETH.

- **tapETHPerToken()**

This function returns the amount of tapETH that corresponds to 1 wtapETH.

- **tokensPerTapETH()**

This function returns the amount of wtapETH that corresponds to 1 tapETH.

Contract StableAsset

The contract **StableAsset** is upgradable and inherits from the contract ReentrancyGuard.

Write Methodes

- ****mint(uint256[] calldata _amounts, uint256 _minMintAmount)****

This function allows the user to provide liquidity in the different tokens of the pool to mint at least `_wtapETHAmount` of tapETH. The Logic of the function consists of :

- 1) update token balances
- 2) calculate the new D value
- 3) calculate delta D = new D - old D
- 4) calculate mintAmount = delta D - feeAmount = delta D * (1- mintFee)
- 5) revert if mintAmount < _minMintAmount
- 6) mint mintAmount of tapETH for the caller
- 7) increase the total supply of tapETH by feeAmount

- ****swap(uint256 _i, uint256 _j, uint256 _dx, uint256 _minDy)****

This function allows the user to swap `_dx` amount of token index `i` to at least `_minDy` amount of token index `j`. The Logic of the function consists of:

- 1) update balance of token index `i` .

-
- 2) calculate the new balance of token index j : new y
 - 3) calculate $\Delta y = \text{new } y - \text{old } y$
 - 4) calculate $\text{outputAmount} = \Delta y - \text{feeAmount} = \Delta y * (1 - \text{swapFee})$
 - 5) revert if $\text{outputAmount} < _minDy$
 - 6) send outputAmount of token index j to the caller
 - 7) increase the total supply of tapETH by feeAmount

- **redeemProportion(uint256 _amount, uint256[] calldata _minRedeemAmounts)****

This function allows the user to redeem $_amount$ of tapETH in order to receive at least $_minRedeemAmounts[i]$ of each token index i . The Logic of the function consists of:

- 1) calculate $\text{redeemAmount} = _amount - \text{feeAmount} = \text{amount} * (1 - \text{redeemFee})$.
 - 2) for each token i :
 - calculate $\text{tokenAmount} = \text{balances}[i] * \text{redeemAmount} / D$
 - revert if $\text{tokenAmount} < \text{minRedeemAmounts}[i]$
 - send tokenAmount of token index i to the caller
 - 3) update $D = D - _amount$
 - 4) burn $_amount$ of tapETH from the caller
 - 5) increase the totalSupply of tapETH by feeAmount
- **redeemSingle(uint256 _amount, uint256 _i, uint256 _minRedeemAmount)****

This function allows the user to redeem $_amount$ of tapETH in order to receive at least $_minRedeemAmount$ of token index i . The Logic of the function consists of:

- 1) calculate $\text{redeemAmount} = _amount - \text{feeAmount} = \text{amount} * (1 - \text{redeemFee})$.
 - 2) calculate the new amount of token i (new y) for $D = D - \text{redeemAmount}$
 - 3) calculate $\Delta y = \text{new } y - \text{old } y$
 - 4) revert if $\Delta y < _minRedeemAmount$
 - 5) send Δy of token index i to the caller
 - 6) increase the total supply of tapETH by feeAmount
- **redeemMulti(uint256[] calldata _amounts, uint256 _maxRedeemAmount)****

This function allows the user to redeem at most $_maxRedeemAmount$ of tapETH to receive $_amounts[i]$ of each token index i . The Logic of the function consists of:

- 1) update balance of each token index i .
- 2) calculate the new D
- 3) calculate $\Delta D = \text{new } D - \text{old } D$

-
- 4) calculate $\text{redeemAmount} = \text{delta D} + \text{feeAmount} = \text{delta D} * (1 + \text{redeemFee})$
 - 5) revert if $\text{redeemAmount} > \text{_maxRedeemAmount}$
 - 6) for each token index i , send $\text{_amounts}[i]$ to the caller
 - 7) increase the total supply of tapETH by feeAmount

functions to be executed only by the governance:

- **`**proposeGovernance(address _governance)**`**

This function allows the current governance to set a new governance address.

- **`**acceptGovernance(address _governance)**`**

This function allows the pending governance to be activated: to update the governance to the pending governance.

- **`**setMintFee(uint256 _mintFee)**`**

This function allows the governance to update the mintFee.

- **`**setSwapFee(uint256 _swapFee)**`**

This function allows the governance to update the swapFee.

- **`**setRedeemFee(uint256 _redeemFee)**`**

This function allows the governance to update the redeemFee.

- **`pause()`**

This function allows the governance to pause the mint, swap and redeem function.

- **`unpause()`**

This function allows the governance to unpause the mint, swap and redeem function.

- **`**setAdmin(address _account, bool _allowed)**`**

This function allows the governance to add an admin if `_allowed` is true or to remove an admin if `_allowed` is false.

- **`**updateA(uint256 _futureA, uint256 _futureABlock)**`**

This function allows the governance to update the value of A to `_futureA` from the block `_futureABlock`.

Write Methodes

- **getA()**

This function returns the current value of A.

- ****getMintAmount(uint256[] calldata _amounts)****

This function returns (uint256 mintAmount, uint256 fee) where mintAmount is the amount of tapETH to mint for the user, and fee is the mint fee.

- ****getSwapAmount(uint256 _i, uint256 _j, uint256 _dx)****

This function returns (uint256 amount, uint256 fee) where amount is the output amount in token of index j to send to the user, and fee is the swap fee.

- ****getRedeemProportionAmount(uint256 _amount)****

This function returns (uint256[] amounts, uint256 fee) where amounts[i] is the output amount in token of index i to send to the user, and fee is the redeem fee.

- ****getRedeemSingleAmount(uint256 _amount, uint256 _i)****

This function returns (uint256 amount, uint256 fee) where amount is the output amount in token of index i to send to the user, and fee is the redeem fee.

- ****getRedeemMultiAmount(uint256[] calldata _amounts)****

This function returns (uint256 amount, uint256 fee) where amount is the amount of tapETH to redeem and fee is the redeem fee.

Contract StableAssetApplication

The contract **StableAssetApplication** is upgradable and inherits from the contract ReentrancyGuard.

Write Methodes

- ****mint(StableAsset _pool, uint256[] calldata _amounts, uint256 _minMintAmount)****

This function allows the user to provide liquidity in the different tokens of the pool `_pool` to mint at least `_wtapETHAmount` of tapETH.

- ****swap(StableAsset _pool, uint256 _i, uint256 _j, uint256 _dx, uint256 _minDy)****

This function allows the user to swap `_dx` amount of token index `i` to at least `_minDy` amount of token index `j` using the pool `_pool`.

- `**redeemProportion(StableAsset _pool, uint256 _amount, uint256[] calldata _minRedeemAmounts)**`

This function allows the user to redeem `_amount` of tapETH from the pool `_pool` in order to receive at least `_minRedeemAmounts[i]` of each token index `i`.

- `**redeemSingle(StableAsset _pool, uint256 _amount, uint256 i, uint256 _minRedeemAmount)**`

This function allows the user to redeem `_amount` of tapETH from the pool `_pool` in order to receive at least `_minRedeemAmount` of token index `i`.

- `**swapCrossPool(StableAsset _sourcePool, StableAsset _destPool, address _sourceToken, address _destToken, uint256 _amount, uint256 _minSwapAmount)**`

This function allows the user to swap `_amount` amount of token `_sourceToken` from the pool `_sourcePool` to at least `_minSwapAmount` amount of token `_destToken` from the pool `_destPool`.