WEB3MUSIC-NETWORK

Token protocol documentation

INTRODUCTION

Token protocol is an application in solidity that provides direct interactions between users and music artists.

The system defines its own custom internal currency, *Web3MusicNativeTokens*, and manages their wallets, funds, and investments by developing the ERC20 standards of the *Ethereum blockchain*. Users, through their W3M tokens, can invest, support and interact with their favorite artists and the system will manage the recognition of the support provided through a weighted voting system based on the investments made.

The application makes contracts to oversee the controlled release of tokens, manage users and their funds, control, manage and remunerate artists and define the operation of the DAO. This ensures that creators receive fair rewards based on the influence they have on their followers and the platform ecosystem.

CONTRACTS

Application has 4 smart contracts carefully implemented to cover all the required functionality.

- Web3MusicNativeToken
- Web3MusicNativeTokenDAO
- Web3MusicNativeTokenManagement
- FanToArtistStaking

The contracts are implemented in different branches used for proper application development.
FanToArtistStaking can be read on gas-saving-v2-checkpoints & improve-changeArtistStaked branches.
Web3MusicNativeToken can be read on 14-logic-for-token-generation & 21-hardcap-a-1bln branches.

WEB3MUSICNATIVETOKEN

Web3MusicNativeToken is the smart contract dedicated to the initialization of tokens and their management. The following features are developed in branch ’14-logic-for-token-generation’.

INITIALIZATION

The **`constructor`** receives as a single parameter the address associated with the FanToArtistStaking contract, to securely connect the two contracts and coordinate them. It is needed to implement stake transfer checks and to verify that only usable tokens can be transferred.

OWNERSHIP FEATURES

The contract inherits the features implemented by the Pausable and Ownable2Step standards provided by the OpenZeppelin library. The **`transferOwnership`** and **`renounceOwnership`** functions can be used only by the owner to modify the ownership of the contract. The owner can also lock and unlock the smart contract and their functionalities by the **`pause`** and **`unpause`** functions.

ERC20 FEATURES

All the standard features of ERC20 tokens are available, plus some specific functions have been overridden to realize the required behaviors. The **`mint`**, **`burn`** e **`burnFrom`** functions provide the interface for implementing basic functionality, they can only be called by the owner.

RELEASABLE PAYMENT

The functionality of undermining/transferring tokens and blocking them at the time of transaction has been realized. The locked tokens will be released linearly throughout the entire lock-up period.
The functions enabling the gradual release of tokens are **`mint_and_lock`** and **`transfer_and_lock`** both of which receive the beneficiary's address, the amount of tokens to transfer, the start of the release period and its duration as input. Before sending the request, they map the transfer data through a specifically created data structure called **`releasablePayment`** using the beneficiary's address as key. Gradual payment can only be made once for each address. Tokens generated in this way and still blocked can be used to invest in artists' stakes.

The actual release occurs each time an address attempts to transfer tokens. This is made possible by overriding the standard function **`_beforeTokenTransfer`**, where the control logic is implemented. It ensures tracking of actual balances and implements functionalities allowing the staking of locked resources, it also updates the payment duration and the amount of blocked tokens to guarantee the release flow.

However, **`balanceOf`**, the standard function of ERC20 tokens, always considers the total quantity of tokens owned, whether locked or unlocked.

RELEASE LOGIC

The release logic is implemented by 3 interacting functions: **`release`**, **`released`** and **`releasable`**.
Underlying these functions is the **`_vestingSchedule`** function, which calculates the amount of total tokens released through an analytic function that linearly associates the elapsed time, the effective duration and the total amount of tokens locked.

`_vestingSchedule` and **`released`** are called together by **`releasable`** which makes the difference between the tokens returned by **`_vestingSchedule`** and the amount of tokens already released provided by **`released`**.

Finally, the **`release`** function, called by **`_beforeTokenTransfer`**, saves the release in the data structure and making the transaction effective.

STAKING FUNCTIONS

The functions that realize the basic mechanics to make stakes are **`lock`** and **`pay`**.

The **`lock`** function allows users to transfer their tokens to the stake address, it takes as input the user's address and the amount of tokens to be staked. This function can only be called from the fanToArtistStaking address.

The **`pay`** function allows artists to be paid by minting the amount of tokens to the artist's address. The logic by which **`pay`** is called is implemented in FanToArtistStaking. It takes as input the same parameters of the **`lock`** function and has equal access control.

HARD CAP

In the branch `21-hardcap-a-1bln`, a hard cap value is established to restrict the quantity of tokens that the owner can generate. Two variables, **`max_mint`** and **`minted`**, are employed for this restriction.

The validation occurs within the **`mint`** and **`mint_and_lock`** functions by referencing the **`max_mint`** value. After the execution of the **`_mint`** function, the **`minted`** value is updated accordingly.

There is no validation within the **`pay`** function. The artist’s reward is not added to **`minted`** value.

FANTOARTISTSTAKING

FanToArtistStaking is the smart contract that implements the necessary features for users' investment and artists' remuneration. It also takes on board the voting power granted to users in relation to their investment. The following features are developed in branch ’gas-saving-v2-checkpoints’.

DATA STRUCTURES

FanToArtistStaking contains several data types and data structures to implement its operation.

Data types:

- **`Stake: uint256 amount, uint40 start, uint40 end`**
- **`ArtistReward: uint256 rate, uint40 start, uint40 end`**
- **`ArtistCheckpoint: uint256 tokenAmount, uint40 lastRedeem, uint256 amountAcc`**

These *data types* are used in the following ***structures***:

- **`_stake`**: a nested mapping **`address`** → **`address`** → **`Stake`**, the first address is the artist, the second address is the user.
- **`_artistReward`**: a dynamic **`ArtistReward`** array that tracks changes in artists' reward rate.
- **`_artistCheckpoints`**: a mapping **`address`** → **`ArtistCheckpoint`** for the operations of the **`_calcSinceLastPosition`** function, a function required to calculate the artist's reward.

Other important structures:

- **`_verifiedArtists`**: a mapping **`address`** → **`bool`** to keep track of verified artists.
- **`_votingPower`**: a mapping **`address`** → **`uint256`** to keep track of each investor's voting power.

INITIALIZATION

The function **`initialize`** takes 6 input parameters:

- **`address Web3MusicNativeToken_`**: defines the address of the Web3NativeToken smart contract and checks that must be different from zero address.
- **`uint256 artistWeb3MusicNativeTokenRewardRate`**: defines the artist reward rate.
- **`uint40 min`**: defines the minimum stake time.
- **`uint40 max`**: defines the maximum stake time.
- **`uint256 limit_`**: defines the limit of rewards.
- **`uint256 changeRewardLimit`**: defines the time interval before the **`artistReward`** can be changed.

In this function the Web3MusicNativeToken contract interface **`_Web3MusicNativeToken`** is initialized, the first data of the **`_artistReward`** array is assigned with **`rate`** equal to reward rate parameter and the minimum and maximum stake periods are assigned to **`_minStakePeriod`** and **`_maxStakePeriod`**. The variables **`REWARD_LIMIT`** and **`CHANGE_REWARD_LIMIT`** are set to the value of the corresponding parameters.

This initializer function can only be called once.

OWNERSHIP FEATURES

The contract inherits the features implemented by the Ownable2Step standard provided by the OpenZeppelin library. The **`transferOwnership`** functions can be used only by the owner to modify the ownership of the contract.

ARTIST VERIFICATION

The **`addArtist`** function takes two addresses, artist and sender, as parameters and can only be called by the owner. It verifies that the artist's address is different from address zero, and in case that artist is not already verified it inserts the address into the **`_verifiedArtists`** mapping by assigning the boolean value to true. Finally, it creates the first **`ArtistCheckpoint`** of the artist by assigning it to the apposite mapping.

VOTING POWER

When a user invests in an artist through the functions inside this contract, the system assigns to user the voting power calculated by the quantity of tokens invested. The assignment is done by updating the dedicated **`_votingPower`** mapping. This voting system will be the basis of the DAO functionality of this application. The amount of total power issued in the **`_totalVotingPower`** variable is also updated.

_CALCSINCELASTPOSITION FUNCTION

This function is used by the three main features of the staking system. It is necessary to calculate the amount of tokens transferable to the artist, update the amount of tokens staked on him, and update the redemption timestamp.

`_calcSinceLastPosition` takes three parameters as input:

- **`address artist`**
- **`uint256 amount`**
- **`bool isAdd`**

First, it calculates the tokens earned up to that time. If the amount of tokens in stake on the artist is greater than zero, it iterates a for loop to keep track of changes in the vector **`_artistReward`**. Inside this loop it finds the interval of time elapsed since that moment and adds to **`accumulator`** variable the amount of calculated tokens. The function that calculates the number of tokens releasable is proportional to the number of tokens on the stake, the time interval and the **`_artistReward`** rate of the interval. Lastly, it is divided by a constant factor equal to 10e9.

When the for loop is done it increments the quantity of **`amountAcc`** variable of the **`ArtistCheckpoint`** inside **`_artistCheckpoints`** map.

The parameter **`isAdd`** determine the behaviour of the second operation. If true, the function adds the amount parameter to the **`tokenAmount`** variable inside **`_artistCheckpoints`** map. Else, it decrements **`tokenAmount`** by the same value. In the end, it updates the **`lastRedeem`** value inside the checkpoint for future calls.

STAKE

The investment of tokens is done through their addition to artist-dedicated stakes. The function **stake** takes as parameters the artist's address, the amount of tokens to be invested, and the expiration time. The amount of tokens invested must be greater than or equal to one unit, which is 1e-18 token.

It ensures the accuracy of the parameters and utilizes the **_Web3MusicNativeToken** interface to execute the function **lock**, validating the success of the call's outcome.

After that, it initializes a **Stake** struct with the parameters and assigns it to the **_stake** map, increases the user's **votingPower**, updates the **_totalVotingPower** and calls the **_calcSinceLastPosition** with **isAdd** parameter set to true to increase the stake.

STAKE TOOLS

Users have three tools to modify the stakes already created.

increaseAmountStaked: this function, after verifying the existence of the stake and ensuring it's not expired, executes the **lock** function through the **_Web3MusicNativeToken** interface. If the function is successful, it adds the amount parameter to the amount variable of the corresponding stake by the **_stake** mapping. It increases the variables dedicated to the voting system and finally calls **_calcSinceLastPosition** to effectively update the artist's stake.

extendStake: this function extends the duration of a stake. The requirements for its execution are that the stake exists, is not already expired, that the time extension passed as a parameter falls within the values of **_minStakePeriod** and **_maxStakePeriod**, and that the new overall duration is not greater than the current timestamp plus **_maxStakePeriod**. The only change it makes on the data is to update the end parameter of the data inside **_stake**.

changeArtistStaked: The function changes the beneficiary of an already made investment. This function checks for the existence of a stake under the old beneficiary's name, **artist**, and the non-existence of one under the new beneficiary's name, **newArtist**. Afterward, it ensures that the existing stake isn't expired and that **artist** and **newArtist** are not the same addresses. To change the beneficiary, it creates a new **Stake** data with the **start** set at the time of the call and inserts it into the **_stake** map under the new artist's name. Subsequently, it calls **_calSinceLastPosition** twice: first with **isAdd** set to true to update the **ArtistCheckpoint** of the new artist, and then with **isAdd** set to false to decrement the tokens from the **ArtistCheckpoint** of the old artist and potentially record the accrued reward up to that point. Finally, it deletes the data dedicated to the old artist within the **_stake** structure.

The latest function has been enhanced in the 'improve-changeArtistStaked' branch, allowing the transfer of invested tokens to an existing stake. This enhancement is achieved by modifying the initial checks and executing an alternate flow if the initial conditions fall into this scenario. If the function **_isStakingNow** alerts to the presence of a stake with **newArtist** as the beneficiary, it verifies that the stake is not expired and subsequently updates the token quantity in the stake. Additionally, it alters the stake duration by determining which of the two stakes has the longer duration and assigns that duration to the stake, setting the current timestamp as **start**.

REWARD

Remuneration of artists is done through two functions based on the **_calcSinceLastPosition**, **getReward**, the external wrapper, and **_getReward**, the internal function.

They take in input the artist's address and **_getReward** calls **_calcSinceLastPosition** with the amount parameter equal to 0 to get the updated amount of tokens releasable through artist's checkpoint, without changing the stake. Afterwards, it calls the function **pay** by using the **_Web3MusicNativeToken** interface and changes the value of **amountAcc** to zero.

REDEEM

The redeem function has two parameters that identify the stake, they are the artist's address and the user's address. It checks that the parameters are correct and that the stake duration has elapsed before executing itself.

It calls the **_calcSinceLastPosition** function to update the stake and redeem the user's investment. As a result, the **isAdd** parameter is set to false and the **amount** parameter is equal to the total tokens invested. After this call, the function decrements the **_votingPower** value related to the user and the **_totalVotingPower**. Finally, it deletes the related data from the **_stake** mapping.

WEB3MUSICNATIVETOKENMANAGEMENT

This contract is responsible for generating the calls that handle the contracts. Administrator can interact with it to manage the entire system, assign roles, manage artists and update the DAO whitelist.

INITIALIZATION

The constructor receives two parameters: the address of the Web3MusicNativeToken and fanToArtistStaking contracts. After checking and storing the addresses for internal calls, it grants all the defined roles to the contract owner.

ROLES

Roles are defined to determine responsibilities and functions of administrators.

- Admin: the account with this role is able to grant or revoke other roles.
- Minter: the account with this role is able to mint tokens.
- Burner: the account with this role is able to burn tokens.
- Artist verifier: the account with this role is able to add artist to F2A whitelist.
- Artist remover: the account with this role is able to remove artist from F2A whitelist.

FUNCTIONS

This contract provides all the interface functions needed to manage the system. They can be called by the owner and administrators with the appropriate role. The following is a complete list of functions with a brief description, functions are divided for each role.

Admin functions:

- **`transferWeb3MusicNativeToken`** transfers ownership of the Web3MusicNativeToken contract.
- **`transferFanToArtistStaking`** transfers ownership of the FanToArtistStaking contract.
- **`pauseWeb3MusicNativeToken`** blocks any type of transaction of the Web3MusicNativeToken contract.
- **`unpauseWeb3MusicNativeToken`** unlocks any type of transaction of the Web3MusicNativeToken contract.
- **`changeWeb3MusicNativeToken`** modify the Web3MusicNativeToken interface by assigning another instance of the Web3MusicNativeToken contract.
- **`changeFTAS`** modify the FanToArtistStaking interface by assigning another instance of the FanToArtistStaking contract.
- **`changeArtistRewardRate`**: change the inflation rate of FanToArtistStaking.

Minter functions:

- **`mint`** executes mint Web3NativeToken function.

Burner functions:

- **`burn`** executes mint Web3NativeToken function.
- **`burnFrom`** executes mint Web3NativeToken function.

Artist Verifier functions:

- **`addArtist`** adds the artist address array to FanToArtistStaking whitelist.

Artist Remover functions:

- **`removeArtist`** removes the artist address array to FanToArtistStaking whitelist.

CUSTOM

The function **`custom`** can be used to execute others function of Web3NativeToken, FanToArtistStaking and Web3NativeTokenDAO contract. This function is used to change the ownership of this contracts and to access the **manageWhitelist** and **switchWhitelist** function of the Web3NativeTokenDAO contract. It can only be used by the owner.

ADDARTIST AND REMOVEARTIST

In the branch `gas-saving-v2-checkpoints`, these functions are modified to operate through the input of address arrays instead of single address. The gas consumption of the for-loop inside this function can be elevated.

VESTING MANAGER

In the branch `14-token-for-generation`, the role of the vesting manager is added. This role is allowed to interface with the Web3NativeToken contract to use the **`mint_and_lock`** and **`transfer_and_lock`** functions. The **`transfer_and_lock`** function first transfers the caller's token to the MGMT. Therefore, to ensure the execution of **`transfer_and_lock`**, an approve function must be executed first.

WEB3MUSICNATIVETOKENDAO

The logic for managing, voting and executing proposals is realized in this contract.

DATA STRUCTURES

The contract generates the **proposal** data structure which has as internal variables:

- **uint256 maxVotingPower**:
- **uint256 votesFor**:
- **uint256 votesAgainst**:
- **uint128 timeStart**:

proposal data are stored in a mapping named **_proposals** and keep track of accepted proposals. To access this mapping data, a value called **porposalId** is used, which is computed using the **hashProposal** function. It executes the keccak256 function that operates on the encoding of the targets, calldatas and description parameters that define a proposal.

Another important structure is **_votes**. It is a double mapping **uint256** → **address** → **bool** that records who have voted on a specific proposal. The associated address is the user's address, while the first uint256 key is the result of the hashing function kekkac256 that takes as parameters the encode of the **proposalId** variables of the proposal and its **timeStart**.

Finally, there is a structure called **whitelistedAddresses** that implements the whitelist of the governance. This whitelist can be disabled via the **whitelistEnabled** boolean that is assigned during initialization. The contract provides functions that can modify this last parameter.

INITIALIZATION

The constructor of this contract acquires and assigns 5 initial variables after verifying their correctness. These variables are:

- **address ftas_**: the address for the interface to the fanToArtistStaking. The interface is contained in ftas.
- **uint128 quorum_**: assigned to **_quorum** and determines the quorum percentage to be reached to execute a proposal.
- **uint128 majority_**: assigned to **_majority** and determines the majority percentage to be reached to execute a proposal.
- **uint128 time**: assigned to the variable timeVotes and is the time interval in which the proposal can be voted on.
- **bool whitelist**: assigned to the variable whitelistEnabled.

For these parameters to be accepted, **ftas** must be different from the zero address, and the quorum and majority must be less or equal than 10e8.

PROPOSE

The function **propose** has as input an array of address targets, an array of call data to perform the operation decided on the respective targets in case the proposal is executed, and a string description of the proposal. The function checks that the arrays have the same length and that a proposal with the same **proposalId** does not already exists. If these checks are successful, the proposal, by **proposalId**, is

assigned to the **_proposals** mapping. The **timeStart** of the proposal is set to the current timestamp and the **maxVotingPower** is determined by making a call to the **totalVotingPower** function of the fanToArtistStaking. The parameters **votesFor** and **votesAgainst** are set to 0.

VOTE

The function **vote** has as input an array of address targets, an array of call data and the string description. These parameters are used to take the **proposalId** of the propose. It also takes a boolean flag to determine whether the vote is for or against the proposal. It successively checks that the sender of that vote is within the whitelist or that the whitelist is disabled before proceeding. If the verification is valid, it computes the **proposalId** and verifies that the user has not already voted using the **_votes** mapping. It acquires the user's **votingPower** quantity by **votingPowerOf** function of the fanToArtistStaking and adds this quantity to the **votesFor** or to the **votesAgainst** in relation to the flag. Finally, update the **_votes** mapping to keep track of the vote.

EXECUTE

This function can be called, with the same parameters as propose, to execute a proposal accepted by the organization. After calculating the proposalId it checks if the proposal is present and that it is finished. It checks that the proposal has been approved through appropriate internal functions and finally deletes the proposal and executes the functions associated with it.

INTERNAL FUNCTIONS

Verification of proposal approvals is done through two internal functions, **_reachedQuorum** and **_votePassed**. The first verifies that the percentage of the quorum, calculated based on **maxVotingPower** by using **_quorum**, is less than the total votes. The second verifies that the majority, calculated based on total votes by using **_majority**, is less than the votes in favour.

GETPROPOSAL

getProposal is a function that takes the same input parameters as **propose**, checks whether the proposal associated with those parameters exists, and returns it to the caller.

WHITELIST

This contract provides two functions to manage the whitelist associated with the DAO:

- **manageWhitelist**: takes a target and a flag as parameters, replaces the value pointed to by **whitelistedAddress[target]** with the flag. It is used to insert or remove a user from the whitelist.
- **switchWhitelist**: enables or disables the whitelist by changing the **whitelistEnabled** value.

They can only be used by the owner.

***Note**: The quorum verification system could be affected by the actual amount of tokens staked in the proposal approval period. A user could substantially increase the amount of tokens staked and saturate the maximum amount of votes, defined at the time of proposal creation, used to calculate quorum.*