Crest-Dapp

Contract functions

Functions described in this doc are only the ones that could be interesting for the front There is no dedicated reward wallet

The only key wallets are:

- **admin**
- **adminProxy**
- **treasury**

All contracts without exception are proxies, The whole system is based on minting and burning mechanisms. All node rewards are directly minted :

- If the token price is too high, tokens are also minted for the pool to meet the high limit price range
- If the token price is too low, tokens are burned for the pool to meet the low limie price range

When a node is created:

- A percentage of the price is burned
- The remaining token amount is sold with a single transaction
- A percentage of the token sold are sent to the treasury
- The remaining of the token sold are sent to the team splitter contract

There is a maximum number of nodes that can be created, this value can be increased over time, each node has a lifetime for it to produce rewards. When the node reached his lifetime, it is not burned, it just stops producing rewards Token and nfts share a common blacklist, stored on the token contract

Token erc20

It is a standard erc20, the pool will be a standard dex such as TraderJoe. The required call to sell is swapExactTokensForTokensSupportingFeeOnTransfer. There is no required call for the token purchase, price is updated on the sale of the token if required but not on the buy Price is also updated if required when new tokens are minted from node claims Functions:

- **Handles all standard erc20 calls** (balanceOf, totalSupply, transfer etc.)
- **decimalsLimit**():
    - This is a value to associate with price limits for the lm
    - It allows to handle decimals price value
    - Refer to limit calls
- **lowLimit**():
    - This is is the low limit price of the lm
    - low limit price = lowLimitReturnValue / (10**decimalsLimit)
- **highLimit**():
    - This is is the high limit price of the lm
    - high limit price = highLimitReturnValue / (10**decimalsLimit)

- **fee**():
  - This is the fee associated to the sale of the token
  - The fee is sent to the treasury wallet
  - 10000 represents 100%
  - 100 represents 1%
- **totalBurn**():
  - Represents the total amount of token ever burned

Node erc721

It is a standard erc721, enumerable, it can be transfered as any erc721, it also can be sold on market places. It follows ERC2981 for royalty fees on the sale on market place, receiver being the treasury. Nft internal claim datas are stored in an array allowing an easier proxy update if required

Functions:

- **Handles all standard enumerable erc721 calls** (balanceOf, totalSupply, transferFrom etc.)
- **baseURI**():
  - Return base uri for nft metadata "baseURI + nftTokenId" represents the http call to do to get those nft metadata
- **tokenURI**(uint `tokenId`):
  - Return the "baseURI + nftTokenId" from previous call for passed tokenId parameter
- **viewTokensOfOwnerByIndex**(address `_owner`,uint `cursor`,uint `size`): *This is a very important call, it allows to get all nftTokenId owned by specific address This call should follow the node.balanceOf(user)*
  - **parameters**:
    - `_owner` is the owner for which you want to get token ids
    - `cursor` is the starting index of the tokenIds
    - `size` is the amount of tokenIds you wish
  - **Returns**(uint[] memory, uint) :
    - The first uint array represents the tokenIds
    - The second uint represents the nb of tokenIds
  - **Example**: *The user has 3 nfts with ids [12, 35, 198]*
    - if cursor = 0 and size = 3 return ([12,35,198], 3)
    - if cursor = 1 and size = 2 return ([35,198], 2)
    - if cursor = 1 and size = 4 return ([35,198], 2)
- **price**(): *Represents the price of the nft*

- **rewardAmountTime**(): *Represents the time unit to calculate rewards, second based*
- **rewardAmount**(): *Represents the exact amount of rewards obtained during a single time unit (rewardAmountTime)*
- **lifespan**(): *Represents the amount of time for a node to stop producing rewards after its creation, second based*
- **max**(): *Represents the maximum number of node that can be created*
- **viewCountData**(uint `tokenId`): *Represents the number of claim data stored on the contract for this `tokenId`, this will always return 2 at first. This should be useless at first for the front but can become handy for updates if we decide to add data for each node*
- **viewData**(uint `tokenId`, uint `cursor`, uint `size`): *It allows to get the claim data of a specific tokenId*
    - **Parameters**: *Cursor and size holds the same role as in "viewTokensOfOwnerByIndex" except that it is for the claim data of the tokenId. This call shouldnt be needed from the front, however it allows to explain better the upcoming one which will be mandatory*
        - `tokenId` is the tokenId
        - `cursor` is the starting index of the datas
        - `size` is the number of data you wish
    - **Returns**(uint[] memory, uint):
        - The first uint array represents the datas with 2 element
            - datas[0] = creationTime, (epoch in second)
            - datas[1] = lastClaimTime, (epoch in second)
        - The second uint represents the nb of datas
- **viewDataBatch**(uint[] calldata `tokenIds`,uint `cursor`, uint `size`): *It is basically the same call as "viewData", however it returns the data for several tokenIds. This allows to fetch several tokenIds data with a single smartcontract call*
    - **parameters**:
        - `tokenIds` is an array of tokenIds, data will be returned in the same order cursor and size are the same as for "viewData".
            - userBalance = node.balanceOf(user)
            - userTokenIds = (await node.viewTokensOfOwnerByIndex(user,0,userBalance))[0]
            - userTokenIdsData = node.viewDataBatch(userTokenIds,0,2)
        - `cursor` is the starting index of the datas
        - `size` is the number of data you wish
    - **Returns**(uint[][] memory): *This is an array of array with the following structure [dataFromTokenIds[0], dataFromTokenIds[1], etc]*

    - dataFromTokenIds being the array of data ("viewData") of the provided array of tokenIds

NodeManager

This is the contract to interact with to create/claim nodes

Functions:

- **viewCountContracts**(): *Return number of node contracts, at first it will be 1, but with more node type arriving the value will increase*
- **viewContracts**(uint `cursor`, uint `size`):

  You start getting used to cursor and size

    - **Returns**(address[] memory, uint):

      - The address array of node contract addresses
      - The second uint represents the nb of addresses

    Looping through the returned addresses allows a fully interactive frontend fetching all new contracts as soon as they are live by refreshing the page

- **createManagedTokens**(uint `i`, address `to`, uint `amount`): *This is the call to create new nfts*

    - **Parameters**:

      - `i` is the contract address index from `viewContracts`, at first it ll always be 0
      - `to` is the address receiving the nft, the buyer is msg.sender, this allows to offer nft to friends
      - `amount` is the number of nft to create

    To create a node, the user MUST have approved the dispatchManager address Logic

```
if (token.allowance(user.address, dispatchManager.address).eq(0))
{

        token.approve(dispatchManager.address,
ethers.constants.MaxUint256)

} else { token.createManagedTokens() }
```

- **claim**(address `from`, uint[] calldata `iData`, uint[][] calldata `tokenIds`):

  This is the call to claim nfts

- **parameters**
  - `from` is the address owning the nfts, it must be equal to msg.sender, just something to allow the contract owner to claim for someone else
  - `iData` are the indexes from "viewContracts", this allows to claim several different node from different contract within a single transaction
  - `tokenIds` is an array of array with tokenids of node contract addresses to claim
    - exemple: *Lets imagine, you have 2 different contract addresses (2 badges)* `claim(user, [0,1], [[2],[4,1]])` *then it will claim token id 2 from contract 0 and the tokenids 4 and 1 from the contracrt 1*

DispatchManager

This is the guy handling the token distribution when creating nft and claiming nft. He must be approved before creating a node as it handles the transfer

Functions:

- **burnRate**(): *this is the percentage burned when creating a node. 7000 = 70% (out of 10000)*
- **treasuryRate()**: *this is the percentage sent to the treasury when creating a node 2000 = 20% (out of 10000)*

Dex

This is for simple explanation regarding the calls to swap tokens/stable

Functions:

- **When selling token**: *must check token router allowance* `(token.approve(router.address, constants.MaxUint256))`, *refer to* `createManagedTokens`. *Then call:*

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountToS
wapOfToken, amoutnOutMinToGetOutAsStable, /* just put 0 */
[token.address, stable.address], addressReceivingStable, deadline /*
blocktimestamp + time margin */)
```

- **When buying tokens**: *must check stable router allowance (stable.approve(router.address, constants.MaxUint256)), refer to "createManagedTokens", then you can call many stuff:*

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
amountToSwapOfStable, amoutnOutMinToGetOutAsToken, /* just put 0 */
[stable.address, token.address], addressReceivingToken, deadline /*
blocktimestamp + time margin */ )
```

```
router.swapExactTokensForTokens( amountToSwapOfStable,
amoutnOutMinToGetOutAsToken, /* just put 0 */ [stable.address,
token.address], addressReceivingToken, deadline /* blocktimestamp +
time margin */)
```

```
router.swapTokensForExactTokens( amountToGetOfToken,
amoutnInMaxAsStable, /* just put 0*/ [stable.address, token.address],
addressReceivingToken, deadline /* blocktimestamp + time margin */)
```

- **Calculation**:
  - **remainingNftToBuy**:
    - `node.max() - node.totalSupply()`
  - **daysBeforeRoi**:
    - `node.price() / (node.rewardAmount() * 24*3600 / node.rewardAmountTime())`

    or

    - `node.price() / node.rewardAmount()` as rewardAmountTime should return precisely 24 * 3600 = 1 day
  - **apr**
    - `(node.rewardAmount() * 24*3600 / node.rewardAmountTime()) / node.price() * 100`

    or

    - `node.rewardAmount() / node.price() * 100` as rewardAmountTime should return precisely 24 * 3600 = 1 day
  - **pendingRewardForSingleTokenId**:

- Refer to `node.viewDataBatch` to get creationTime and lastClaimTime
- Refer to `node.lifespan` to get lifespan
- Refer to `node.rewardAmount` to get rewardAmount
- Refer to `node.rewardAmountTime` to get rewardAmountTime

```
function getSingleNodeReward()
{
    if (creationTime + lifespan < currentTimestamp)
    {
        return rewardAmount * (creationTime + lifespan -
lastClaimTime) / rewardAmountTime
    }
    return rewardAmount * (currentTimestamp - lastClaimTime) /
rewardAmountTime
}
```

- **tokenPrice**:

```
function getPrice()
{
    const reserves = await pair.getReserves(token.address,
stable.address)
    let reserveStable, reserveToken

    if (token.address < stable.address)
    {
        // bignumber comparison
        reserveToken = reserves[0]
        reserveStable = reserves[1]
    } else
    {
        reserveToken = reserves[1]
        reserveStable = reserves[0]
    }

    reserveToken /= 10**(await token.decimals())
    reserveStable /= 10**(await stable.decimals())
    return reserveStable / reserveToken
}
```

- **market cap**:
  - Return `getPrice() * (token.totalSupply() / 10**(await token.decimals()))`