Autonolas Tokenomics

A brief overview of the tokenomics model **1**
- Main tokenomics objectives 2
- OLAS token 2
  - Functionalities enabled by OLAS 2
  - OLAS inflation model 2
- Main tokenomics primitive 3
- How the staking model for agents and component code is incentivized 3
- How and when the bonding mechanism is incentivized 5
  - Product function 6
  - How interest rate and discount factor are regulated 7
  - Bonding respects the OLAS inflation schedule 7

Tokenomics smart contract **8**
- High-level description 8
- Technical description 9
  - Depository.sol 9
  - GenericBondCalculator.sol 13
  - Treasury.sol 14
  - Dispenser.sol 20
  - DonatorBlacklist.sol 21
  - TokenomicsConstants.sol 22
  - TokenomicsProxy.sol 23
  - Tokenomics.sol 24

A brief overview of the tokenomics model

The goal of this tokenomics is to enable the rise of a sustainable ecosystem of Autonomous Services.

Autonomous services are capable of complex processing and can provide any level of desired centralization or decentralization. When operated by groups with open participation, these are extremely reliable and robust. In Autonolas, autonomous services are implemented as Agent Services which primarily run off-chain but can be also connected to a settlement layer (i.e. a programmable blockchain). These services are made up of *software agents* which in turn are made up of *software components.*

Main tokenomics objectives

Reaching its goal, the **main objectives of tokenomics** are the following:

- To incentivize the development and composability of software agents and software components.
- To grow the Autonolas protocol's capital in the form of protocol-owned liquidity (POL) when there is a high potential output of software development. In such a way capital will not exceed software code production and its usefulness.
- To enable the protocol to own its productive services and accrue donations from them.

OLAS token

The protocol coordinates the goals described above through a tradable utility token, OLAS, that will provide access to the core functionalities of the Autonolas project. The token follows the ERC20 standard and is deployed on the Ethereum mainnet.

The token has an inflationary model to account for the economic primitives enabled by Autonolas tokenomics, e.g., bonding mechanism and OLAS top-up to boost developers' incentives (cf. tokenomics primitive).

Functionalities enabled by OLAS

We summarize the main functionalities enabled by OLAS as follows:

1. OLAS can be locked for veOLAS to participate in the Autonolas DAO governance, thus shaping the protocol and its tokenomics
2. OLAS can be locked for veOLAS for permissionless access to a service whitelist that unlocks code owners' top-ups
3. OLAS can be used to acquire (on a third-party DEX) LP-tokens that are required for the bonding mechanism. This will enable protocol-owned liquidity and therefore support the protocol's long-term growth.

OLAS inflation model

The number of OLAS tokens is capped at 1bn for the first 10 years and the maximum token inflation per annum is capped at 2% thereafter. Upon launch, an allocation of

- 32.65% of the tokens will be sold to founding members of the DAO;
- 10% of the tokens are earmarked for future private and public sales;
- 10% of the tokens will be allocated to the Autonolas DAO treasury; and
- 47.35% may be used to incentivize developers' top-ups for useful code and bonders, autonomously provisioned by the protocol over the initial 10 years.

Main tokenomics primitive

Tokenomics primitives that are primarily leveraged to accomplish the above objectives:

1. **Staking model for agent and component code**
   Developers of agent and component code can register their code on-chain by using Autonolas registries. The code existing off-chain will be uniquely represented on-chain by means of NFTs. The developer can accrue incentives proportional to their code contribution.
2. **Bonding mechanism.** The protocol can grow its own liquidity by incentivizing liquidity providers to sell their own liquidity pairs (with one of the tokens in the pair being the protocol token, e.g. OLAS-ETH) to the protocol for OLAS at a discount
3. **Protocol-owned services (PoSes).** The PoSes are autonomous services, owned by a DAO, operated by the ecosystem, and implemented by developers around the world. This novel primitive enables the DAO that manages the PoSe to own productive autonomous services and derive donations from them.

How the staking model for agents and component code is incentivized

Before proceeding with the next two sections, it is important to give the following definition. The term epoch can either be defined as any consecutive period of *m* blocks or similarly as a certain number of consecutive seconds *len*. Note that the number of block *m* or similarly the number of seconds *len* are tunable values that can be set by the DAO governance and have a fixed minimum.

1. Developers stake agents and/or components on-chain
2. Service owners use staked agents and components to create autonomous services. Then they register their service on-chain using Autonolas registries

3. A signal of appreciation for a specific autonomous service can be shown by sending a donation to the protocol for that service.
   Specifically, to signal appreciation to a registered service s_j, an Ether donation r_j can be sent to Autonolas protocol (using the depositServiceDonationsETH method, see methods in Treasury.sol) for the registered service s_j.
4. With a share of accrued donations, the protocol will reward the staked agents and/or components that facilitated such donations.
   Specifically, assuming that during an epoch,
   1. the registered services s_j are appreciated with donations r_j
   2. the number of staked components (respectively agents) referenced in the service s_j is denoted with $Ncomp(s_j)$ (respectively $Nagents(s_j)$)
   3. the staked component c_i (agent a_i) is referenced in the registered services $s_{j,i}$, $j = 1, ..., l_i$
   4. $Fcomps$ (respectively $Fagents$) is the share of the donations aimed to reward the staked referenced components (respectively agents)

   then the protocol will reward component c_i with the following amount

$$rewards_{c_i} = Fcomps \cdot \sum_{1 \leq j \leq l_i} \frac{r_{j,i}}{\mathrm{Ncomps}(s_{j,i})} \quad (1)$$

   (respectively agent a_i

$$rewards_{a_i} = Fagents \cdot \sum_{1 \leq j \leq l_i} \frac{r_{j,i}}{\mathrm{Nagents}(s_{j,i})} \quad (2)$$

)

5. Moreover, when the service owners or donors own a certain *threshold of veOLAS* (e.g. they have locked a certain amount of OLAS for a certain period of time), they are considered whitelisted. The protocol will also distribute a share of the OLAS inflation to staked components (resp. agents) referenced in the whitelisted services.
   Specifically, assuming that during an epoch,
   a. the registered services s_j are appreciated with donations r_j
   b. the number of staked components (respectively agents) referenced in the service s_j is denoted with $Ncomp(s_j)$ (respectively $Nagents(s_j)$)
   c. the staked component c_i (agent a_i) is referenced in the registered *whitelisted* services $s_{j,i}$, $j = 1, ..., l_i$
   d. the *whitelisted services* are s_k, $k = 1, ..., w$

5. $TopUpC$ (respectively $TopUpA$) is the share of the epoch inflation that the protocol aims to distribute to the staked components (respectively agents) referenced in whitelisted services

then the protocol will distribute to the component c_i with the following amount of OLAS

$$topUp_{c_i} = \frac{TopUpC}{\sum_{1 \leq k \leq w} r_k} \cdot \sum_{1 \leq j \leq l_i} \frac{r_{j,i}}{\text{Ncomps}(s_{j,i})} \qquad (3)$$

(respectively to agent a_i

$$topUp_{a_i} = \frac{TopUpA}{\sum_{1 \leq k \leq w} r_k} \cdot \sum_{1 \leq j \leq l_i} \frac{r_{j,i}}{\text{Nagents}(s_{j,i})} \qquad (4)$$

).

How and when the bonding mechanism is incentivized

In a nutshell, the bonding mechanism can be summarized as follows.
An investor with an enabled Uniswap V2 LP-pair asset (e.g. OLAS-DAI, or OLAS-USDC, or OLAS-ETH) can deposit his assets via the Autonolas depository smart contract at time t. Then at time *t+tv*, where *tv* is the vesting time, the investor receives OLAS at a discount relative to the price quoted on the relevant DEX.

So, assume that an investor wants to bond certain numbers of his shares of an enabled Uniswap V2 LP-pair at a time t whose price is *priceShares* on a relative DEX. Then at time *t+tv* the bonder receives the following number of OLAS whose price at time *t* is

(1+epsilon(t))* *priceShares*

The value epsilon(t) can be defined as the *interest rate* on purchased bonds with price *priceShares* and DF(t)=1/(1+epsilon(t)) is defined as the *discount factor.*

Therefore when one aims to incentivize the growth of bond demand, the interest epsilon(t) can be set to a large value and DF(t) to a small one. Vice versa when aimed to discourage the bond demand the epsilon(t) can be set to a very small value and DF(t) to a large one would.

Currently, the protocol aims to *incentive bonding when there is a large potential output of code (agents/components) production* in the ecosystem. In particular, a production function (cf. subsection below) is used to measure the potential code production during one epoch and, in turn, to establish the interest rate and so the discount factor which a bonder can receive on purchased bonds.

Product function

The potential code production is measured using a **production function**. In Economics, a **production function** relates outputs of a production process to inputs of production. So it is a mathematical function that relates to the amount of output that can be obtained from a given number of inputs. Generally, these inputs are capital and labor.

In our case

Output = valuable code (staked agents and components) to enable services that, in turn, can provide donations to the protocol.

Production process = creation of components and agents

Inputs

- $K(n)$ is the capital (right now, intended as the Ethers that becomes owned by the protocol) accrued by the protocol during the n-th epoch
- $D(n)$ is the number of *valuable* developers that is the *stakers* of *useful* code (e.g. agents or components code referenced in appreciated services, cf. section above to see how the signal of appreciation is intended) during the n-th epoch.

Then, the production function is computed as

$$f(K(n), D(n)) = d(n) * (k(n) * K(n) + D(n)),$$

where

- $k(n)$ is the (average) number of valuable developers that can be funded by the protocol with one *unit of capital* (currently, one unit of capital is intended to be one Ether) during one epoch
- $d(n)$ is the (average) number of *units of useful code* that can be built by a developer during one epoch (currently one *unit of code* is intended to be either one component or two agents).

Specifically, $f(K(n), D(n))$ outputs the number of *units of useful code* that can be produced during one epoch with $K(n)$ as input capital and $D(n)$ as input number of developers.

In particular, this means that when $K(n)$ and $D(n)$ are large enough, then there is potential large production of useful code, while when $K(n)$ and $D(n)$ approaches zero, there is scarce potential of useful code production.

How interest rate and discount factor are regulated

As mentioned earlier, the protocol aims to *incentive bonding when there is a large potential output of code (agents/components) production* in the ecosystem.

Let epsilon be the maximum possible interest rate on a bond that the DAO governance wants to give.
Since when $f(K(n), D(n))$ is large there is high production of useful code, then bonding should be incentivized, and the interest rate for bonding during the (n+1)-th epoch maximum interest rate is set equal to epsilon.
While, when $f(K(n), D(n))$ is small and in particular $f(K(n), D(n))$ / 100 is smaller than the epsilon rate, there is potentially low production of useful code, so the bonding should be disincentivized, and the interest rate on the bonding during the (n+1)-the epoch is set equal to $f(K(n), D(n))$ / 100

Bonding respects the OLAS inflation schedule

There is an inflation schedule that dictates how much OLAS can mint per epoch at a maximum. Since bonding implies minting new OLAS tokens, we need to take into account that during an epoch we have a maximum amount that can be bonded.

Tokenomics smart contract

High-level description

A high-level description of Autonolas tokenomics smart contract architecture follows.

Tokenomics.sol

This smart contract contains the logic of the mathematical model. Specifically, it contains the logic required to calculate the incentives for components and agents' code in terms of their usefulness in the ecosystem and the logic to regulate the discount factor of the bonding mechanism in terms of the potential code production.

TokenomicsProxy.sol

This smart contract stores the data and uses the logic of the tokenomics contract by means of the `delegatecall()` function. Specifically, this is the single contract for storage that is considered to be immutable for a long time. The tokenomics implementation is just stacked up to it via the deployed tokenomics version address.

Depository.sol

This contract implements the logic behind the bonding mechanism: create or close bond programs for specific LP pairs, allow users to deposit their LP pairs, and accrue the matured OLAS at discount in exchange for their deposited pairs.

GenericBondCalculator.sol

This contract is primarily implemented to calculate the amount of OLAS that a bonder will receive in exchange for their deposited LP assets[1].

Treasury.sol

This contract contains the logic for the management of the Autonolas protocol Treasury. It provides the means to donate to the protocol, withdraw assets from the treasury, drain assets in other protocol contracts, and mint OLAS.

---

[1] Note that the system is immune to price oracle attacks because the calculation made here and used in another contract is based only on DAO inputs assigned when a bonding program is opened and on a discount factor that uses internal parameters accumulated with the tokenomics contract.

Dispenser

This contract allows agent/component developers to claim their incentives (e.g. the incentives autonomously assigned by the protocol to reward useful code).

DonatorBlacklist.sol

This contract allows managing an address blacklist. This is necessary to filter the donator addresses and allow the protocol to only accept donations from non-blacklisted addresses.

TokenomicsConstants.sol

Abstract contract that contains constants related to tokenomics such as the OLAS annual supply cap and inflation amount for ten years.

Technical description

Depository.sol

Ownable smart contract with managers Treasury and Tokenomics contracts. Other than having the functions to update owner, managers, and the BondCalculator contract, the Depository contract has the following methods.

1. Create function allows the *contract owner* to start a bonding program.
   Input parameters to correctly add for creating the program:
   a. `token:` Uniswap v2 LP token address *enabled by the Treasury*
   b. `priceLP:` LP token price with 18 decimals and *non-zero* at which an LP share is priced during the bonding program
   c. `supply:` OLAS supply (*non-zero and beyond the limit fixed by the tokenomics to fund bonding programs and not overflowing the contract limit*) that will be reserved to fund OLAS for this bonding program
   d. `vesting:` the vesting time (*bigger or equal to the minimal vesting value*) in seconds that a bonder has to wait before being able to withdraw OLAS

Whether the inputs parameters are correct and the function caller is the contract owner,

- the bonding program with a new `productId` is created and will mature after the `vesting` time
- the `Product` information, e.g. `priceLP, token, supply, maturity` are pushed in the `mapBondProducts[productId]` mapping
- the event `CreateProduct(token, productId, supply)` is emitted

2. Close function allows the *contract owner* to terminate a set of bonding programs:
   a. `productIds`: array of bonding program identifiers to be terminated

   When the function caller is the contract owner
   - If the OLAS `supply` reserved for each `productIds[i]` of bonding program was not completely used, it is again accounted for in the supply reserved by the tokenomics for the bonding mechanism
   - Each bonding program `productIds[i]` will be closed
   - The relative information in `mapBondProducts[productIds[i]]` will be deleted
   - The events `CloseProduct(token, productIds[i])` are emitted

3. Deposit function allows a user to purchase OLAS at a discount by creating a bond in exchange for a deposit of the user's LP-share (of an Uniswap v2 LP `token`) via an open bonding program previously created with `token` as input address.
   Input parameters to correctly add for a successful bond creation:
   a. `productId`: *open* bonding program identification for bond creation
   b. `tokenAmount`: (*non-zero*) share of LP token of the user's balance sold to the protocol for OLAS

   Whether the user calling the function has priory approved the treasury for spending the `tokenAmount` (e.g. has successfully `approved(Treasury.address, tokenAmount))`, the input parameters are

correct and the OLAS payout is beyond the `supply` limit allowed for the bonding program `productId`

- the bonding program `supply` is updated,
- a bond with `bondId` is created
- The `Bond` information, e.g. user's `address, payout, maturity, productId` are pushed in the `mapUserBond[bondId]` mapping
- The tokenomics contract function `depositTokenForOLAS` is called to deposit the `tokenAmount` in the protocol treasury
- the event `CreateBond(token, productId, payout, tokenAmount)` is emitted

4. Redeem function allows a bonder owner to accrue the OLAS payout arising from its matured bonds.
   Input parameters to correctly redeem OLAS:
   a. `bondIds:` *bonder's* bonds *matured* (e.g. function called after vesting time)

   Whether the input parameters are correct, the bonds exist, the function caller is the bonds owner, and the `payout` is not zero
   - the `payout` is transferred to the function caller
   - the relative information in `mapUserBond[ bondId[i]]` will be deleted
   - the relative information in `mapUserBond[ bondId[i]]` will be deleted

5. getProducts function allows a user to get information on all (not-closed yet) bonding programs.
   Input parameter to correctly receive the information.
   a. `active: true` for having the information on all the active products and `false` otherwise

   If the input is `true` the function will return the array of active bonding programs identifiers `productIds` and if the input is `false` the function will return the array of inactive bonding programs identifiers `productIds`

6. isActiveProduct function allows a user to get information on the status of a bonding program.
   Input parameters to correctly receive the information:
   a. `productId:` identifier of a bonding program
   Whether the input parameter is an existing bonding program then the function will return
   - `true` whether programs identifiers `productId` is still active e.g. has non-zero OLAS supply left
   - `false` whether programs identifiers `productId` is not active e.g. has zero OLAS supply left

7. getBonds function allows a user to get information on matured or unmatured (non-redeemed) bonds and the total OLAS payout of a certain `account.`
   Input parameters to correctly receive the information:
   a. `account:` *non-zero* address of a bonder
   b. `matured:` *true or false* to get information of non-redeemed but matured `account'`s bonds or non-redeemed and unmatured `account'`s bonds.

   Whether the input parameters are correct the function will show the (non-zero) array of (matured or not) `bondId`s that the `account` has and the total OLAS payout `account` can accrue when he redeems.

8. getBondStatus function allows a user to get information on the maturity and the payout of a single `bondId`.
   Input parameters to correctly receive the information:
   a. `bondId:` identifier of the (*pending*, e.g. *non-zero*) bond
   Whether the input parameters are correct the function will show the (non-zero) OLAS payout for the `bondId` and will show true if it is matured (and can be then redeemed) and false otherwise.

9. getCurrentPriceLP function allows a user to get information on the amount of OLAS tokens a user can receive by removing one LP share from the pool of Uniswap V2 LP pair with the address `token`.
   When the address `token` is a correct and existing Uniswap V2 LP pair address the function will return the amount of OLAS tokens a user can receive by removing one of the LP shares.

GenericBondCalculator.sol

Smart contract for calculating the OLAS payout for the bonding programs. It has the following methods. Among others, this contact uses some methods from the interface IUniswapV2Pair.sol.

1. calculatePayoutOLAS function calculates the amount of OLAS payout depositing an amount equal to `tokenAmount` into an active bonding program created with input parameter `priceLP`.
   Input parameters
   a. `tokenAmount`: LP token shares
   b. `priceLP`: the price of an LP share

   First, `totalTokenValue=mulDiv(priceLP,tokenAmount)` is made in such a way it is possible to check for an overflow of `totalTokenValue` and not of each of the multipliers. If this `totalTokenValue` does not overflow and the call is made during the e-th epoch, the `getLastIDF()` is accrued from the tokenomics contract. This provides the inverse of the discount factor calculated by the tokenomics at the end of the (e-1)-th epoch using data accrued during such an epoch. Then the following calculation is made

   ```
   amountDF =(getLastIDF() *  totalTokenValue) / 1e36;
   ```

   The division is due to the fact that the results in output require the same decimals of `tokenAmount` and both `getLastIDF(), priceLP` are multiplied by 18 decimals.

Once the above calculation is done, the function will return `amountOLAS=amountDF`. The latter is equal to the OLAS payout that is possible to receive by depositing an amount of `tokenAmount` share of Uniswap LP-pair to an active bonding program created using such LP-pair and pricing each of the shares by `priceLP`.

2. getCurrentPriceLP function gives information on the amount of OLAS that can be received by removing one LP share from a liquidity pool whose UniswapV2 LP-pair is OLAS and token Y and address `token`.
   The input is
   a. `token:` Uniswap v2 LP token address (e.g. OLAS-ETH).
   Whether the input correctly is a pool with one of the tokens being OLAS, the function uses IUniswapV2Pair.sol methods
   - `totalSupply()` of LP tokens in the pool
   - `getReserves()` of the OLAS reserve and token Y reserve in the pool.
   Then it outputs
   ```
   (OLAS_reserve*1e18) / totalSupply;
   ```
   which is the amount of OLAS it is possible to receive by removing one LP share from the Uniswap v2 pool with the address `token`.

Treasury.sol

Ownable smart contract with managers Tokenomics, Depository, and Dispenser contracts. Other than having the functions to update the owner and managers, Treasury has the following methods.

1. receive is the fallback payable function executed whenever the contract receives plain Ether without data. Provided that at least `minAcceptedETH` is sent, this

function allows the contract to receive `msg.value` Ether, add `msg.value` to the balance of the contract, and sum `msg.value` to the `ETHOwned` public variable. Whenever the contract correctly receives `msg.value` Ether the following event is emitted `ReceiveETH(msg.sender, msg.value)`.

2. changeMinAcceptedETH function that allows the contract owner to change `minAcceptedETH` which is the minimal amount of Ether that can be sent to the contract via the execution of the receive fallback payable function.
   The input to correctly execute the function is
   a. `_minAcceptedETH:` a value (*non-zero and smaller than 2^96)*
   If correctly executed
   - the event `MinAcceptedETHUpdated(_minAcceptedETH)` is emitted
   - the receive fallback function will be triggered when an amount of Ether larger than `minAcceptedETH` is sent to the contract.

3. depositTokenForOLAS function allows the Depository contract to deposit an asset in exchange for OLAS.
   Input parameters for the correct function execution:
   a. `account:` address making a deposit of a share of LP-pair with address `token`
   b. `tokenAmount:` share of the LP-pair that `account` wants to deposit in an active bonding contract
   c. `token:` address of the Uniswap LP-pair used in the active bonding program
   d. `olasMintAmount:` payout in OLAS that `account` can receive after the bond matures.
   Whether the input parameters are correct, this function is called by the depository contract, the `allowance(account, Treasury.address)` is larger or equal to `tokenAmount`, and the transfer of `tokenAmount` from `account` to Treasury successfully happens:
   - the `tokenAmount` is summed to the treasury LP-reserve of the `token` accounted for in the mapping `mapTokenReserves[token]`

- `olasMintAmount` of OLAS is minted for the Depository contract
- the event `DepositTokenFromAccount(account, token, tokenAmount, olasMintAmount)` is emitted.

4. depositServiceDonationsETH function allows anyone to deposit Ether donations for services.
   Input parameters for the correct function execution:
   b. `serviceIds`: array of service identifiers `serviceId`s for which the donor wants to signal appreciation via Ether donation
   c. `amounts`: array of corresponding Ether quantities to allocate for each service with identifiers in `serviceIds` (e.g. `amounts[i]` assigned for `serviceIds[i])`.
   Provided that at least `minAcceptedETH` is sent, the `reentrancyGuard` passes, the arrays `serviceIds` and `amounts` have the same length, `amounts[i]` is always non-zero, the sum of the `amounts[i]` is equal to the amount `msg.value` sent, and there is no value overflow
   - the sum of the `amounts[i]` is summed and accounted for in the `ETHFromServices` public variable
   - the `trackServiceDonation` method in the Tokenomics contract is called (in such a way that Tokenomics contract can correctly track donations)
   - the event `DonateToServicesETH(msg.sender, msg.value)` is emitted.

5. withdraw function allows the contract owner to transfer to the address `to`, a `tokenAmount` from the Treasury reserve of the LP-asset or Ether with address `token`.
   Input parameters for the correct function execution:
   a. `to`: address *different* from the Treasury one
   b. `tokenAmount`: *non-zero* amount to withdraw from the treasury reserve
   c. `token`: address of the one of Treasury owned LP-asset or of the Treasury owned Ether

Provided that the function caller is the contract owner, the asset (whether an LP-pair) with address `token` was enabled by the Treasury and at least `tokenAmount` are owned by treasury

- the reserve owned by the treasury is updated,
- the asset is transferred to the address `to`
- transfer happens and if it is successfully `true` is returned and one of the following is emitted:
  - `Withdraw(ETH_TOKEN_ADDRESS), tokenAmount)` when `token` is Ether address
  - `Withdraw(token, tokenAmount)` when `token` is an enabled LP-pair address

6. withdrawToAccount function allows the Dispenser contract to transfer to the address `account`, the amounts `accountReward` and `accountTopUps`.
   Input parameters for the correct function execution:
   d. `account:` an address
   e. `accountReward`: the amount of Ether to withdraw from the Treasury (accounted for in the treasury services endpoint donation (`ETHFromServices`)
   f. `accountTopUps: the` amount of OLAS that will be minted

   Note that, this function can be only called by the dispenser contract in order to allow code-NFTs owners to accrue the rewards and the top-ups amount that the tokenomics assigned for their staked NFTs. The check on the correct allocation of OLAS amount `accountTopUps` is made by the tokenomics contract, so the inflation schedule is correctly respected. While on the treasury side, it is checked that `accountReward` amount of Ether is really accumulated in `ETHFromServices` Treasury endpoint.
   Provided that the function is called by the dispenser contract, the contract isn't paused, and the check on `accountReward` passed then
   - `ETHFromServices` is updated
   - the Ether and/or OLAS are transferred to the address `account`

- after a successful transfer, whether `accountTopUps` are correctly minted the event `TransferToDispenserOLAS(accountTopUps)` is emitted and `true` is returned.
-

7. rebalanceTreasury function allows the Tokenomics contract to rebalance `treasuryRewards` for a specific epoch.
   Input parameters for the correct function execution:
   a. `treasuryRewards:` amount of Ether reserved by the tokenomics to the treasury
   Whether the function call is done by the Tokenomics contract and the Treasury contract is not paused the function returns
   - `true` when
     - `treasuryRewards` is zero (e.g. no Ether are allocated to the treasury by the tokenomics)
     - `treasuryRewards` is not-zero and less or equal than the Ether in the `ETHFromServices` endpoint. The information that `treasuryRewards` Ethers are now in possession of the Treasury is pushed in the `ETHOwend` public variable.
   - `false` when `treasuryRewards` is non-zero and larger than `ETHFromServices.`

8. drainServicesSlashedFunds function allows the contract owner to drain slashed funds from the service registry.
   Input parameters for the correct function execution:
   a. `amount:` amount of slashed Ether in the service registry contract.
   Whether the function call is done by the contract owner, and the Treasury contract is not paused the function returns
   - `true` when
     - `treasuryRewards` is zero (e.g. no Ether are allocated to the treasury by the tokenomics)

- `treasuryRewards` is not-zero and less or equal than the Ether in the `ETHFromServices` endpoint. The information that `treasuryRewards` Ethers are now in possession of the Treasury is pushed in the `ETHOwend` public variable.
- `false` when `treasuryRewards` is non-zero and larger than `ETHFromServices`.

9. enableToken function allows the contract owner to enable Treasury to receive an asset with an address `token`.
   Input parameters for the correct function execution:
   a. `token`: non-zero address
   Provided that the function is called by the contract owner and the `token` isn't the zero and not yet approved
   - the information that such an asset is enabled is pushed in the map `mapEnabledTokens[token]=true`
   - the event `EnableTokens(token)` is emitted.

   This function is essential to enable the creation of bonding programs accepting deposits for Uniswap v2 LP pair with address `token`.

10. disableToken function allows the contract owner to disable the Treasury to receive an asset with an address `token`.
    Input parameters for the correct function execution:
    a. `token`: address of an asset (previously approved)

    Provided that the function is called by the contract owner, the `token` was previously approved, and there is zero amount of such an asset in the treasury
    - the information that such an asset is disabled is pushed in the map `mapEnabledTokens[token]=false`
    - the event `DisableTokens(token)` is emitted.

11. isEnabled function allows checking whether an asset with an address `token` is enabled/disabled.
    Input parameters for the correct function execution:
    a. `token:` address of an asset
    The method return
    - `true` whether the asset with address `token` is enabled
    - `false` whether the asset with address `token` is disabled

12. pause method has no input parameter and allows the contract owner to pause some contract interaction (e.g. withdrawToAccount, rebalanceTreasury). The event `PauseTreasury` is emitted when the contract owner calls this method.

13. unpause method has no input parameter and allows the contract owner to unpause some contract interaction (e.g. withdrawToAccount, rebalanceTreasury). The event `UnpauseTreasury` is emitted when the contract owner calls this method.

Dispenser.sol

Ownable smart contract with managers Tokenomics and Treasury contracts. Other than having the functions to update the owner and managers, Dispenser has the following method.

1. claimOwnerIncentives function allows owners of code NFTs owners to claim their incentives.
   Input parameters for the correct function execution:
   a. `unitTypes:` array of unitType for which the rewards are claimed
   b. `unitIds:` array of code NFT identifiers corresponding to the `unitType` for which rewards are claimed.

   Note that, `unitTypes[i]` can be either:
   - 0 e.g. it represents a component

- 1 e.g. it represents an agent.

Moreover, `unitIds[i]` is the identifier of the NFT uniquely representing on the Autonolar registries the code with unit type `unitIds[i]`.

If the reentrancyGuard passes the following steps happen

- a call to the Tokenomics method `accountOwnerIncentives` is made. Specifically, `msg.sender` that made a call to this function will be the input account for the tokenomics method `accountOwnerIncentives`. The latter returns the amount of `reward` (in Ether) and `topUp` (in OLAS) that the `msg.sender` should receive.
- Whether `reward+topUp` is bigger than zero the contract calls the Treasury method `withdrawToAccount(msg.sender, reward,topUp)` which will transfer to `msg.sender` an amount of `reward` Ether and `topUp` OLAS. If this method execution and in particular the transfer is successful, `true` is returned.
- When the `withdrawToAccount` returns `true`, the method `claimOwnerIncentives` is correctly executed and returned `(reward,topUp)`.
- When the `withdrawToAccount` returned `false`, the method `claimOwnerIncentives` is reverted.

DonatorBlacklist.sol

Ownable smart contract with the following methods.

1. setDonatorsStatuses function allows the contract owner to control addresses in order to prevent any donation to some addresses.
   Input parameters for the correct function execution:
   a. `accounts`: array of addresses for which the contract owner what to change the blacklisted statuses
   b. `statuses`: array of bool values that the contract owner what to assign to the corresponding address in `accounts`.

Specifically, `statuses[i]` has to be set to `true` to blacklist `accounts[i]` and prevent donations from it. While `statuses[i]` has to be set to `false` to remove `accounts[i]` from the blacklist and accept donations from it.

Whether the method is called by the contract owner, `accounts` and `statuses` have the same length, and no-zero address is in the array `accounts`

- the information on the blacklisted status of each of the `accounts[i]` is pushed to the map
- the following events are emitted `DonatorBlacklistStatus(accounts[i], statuses[i])`
- `True` is returned.

2. isDonatorBlacklisted function allows checking the blacklist status of the address `account`.

   Input parameter:

   a. `Account:` address for which one wants to check the status

   The function shows

   - `true` if the address `account` is blacklisted
   - `false` if the address `account` isn't blacklisted.

TokenomicsConstants.sol

Abstract contract smart containing the public constant of the tokenomics implementation version: `VERSION = "1.0.0"` and the following public pure function.

1. getSupplyCapForYear function that returns the annual `supplyCap` for the year `numYears`.

   Input parameter

   a. `numYears:` year (starting from zero)

   When `numYears`

   - is strictly smaller than 10, the corresponding `supplyCap` for that year `numYears` is returned.

- is larger or equal to 10, the corresponding `supplyCap` for that year `numYears` is returned after that it is calculated taking into an annual maximum inflation of 2%.

2. getInflationForYear function that returns the annual `inflationAmount` for the year `numYears`.

Input parameter

a. `numYears`: year (starting from zero)

When `numYears`

- is strictly smaller than 10, the corresponding `inflationAmount` (e.g. the new number of OLAS minted with respect to the old supply) for that year `numYears` is returned.
- is larger or equal to 10, the corresponding `inflationAmount` for the year `numYears`

TokenomicsProxy.sol

This smart contract stores the data and uses the logic of the tokenomics contract by means of `delegatecall()`. The implementation contract (e.g tokenomics (logic) contract) is just stacked up to it via the deployed tokenomics version address. This allows a proxy implementation to follow the Universal Upgradeable Proxy Standard (UUPS) EIP-1822 standard. Specifically, this is the single contract for storage that is considered to be immutable for a long time and contains the following functions.

1. constructor

   Input parameters

   a. `tokenomics`: address of the first tokenomics contract implementation

   b. `tokenomicsData`: callable data with parameters of the `initializeTokenomics()` function from the Tokenomics contract plus the function name and arguments itself

The contract is correctly deployed when `tokenomicsData` does not have zero length and its storage is correctly initialized with the arguments of the tokenomics contract.

2. fallback

The proposed fallback follows the common pattern described in https://eips.ethereum.org/EIPS/eip-1822 and stores the address of the tokenomics (logic) contract at the defined storage position.

Tokenomics.sol

Ownable smart contract managed by Treasury, Depository, and Dispenser contracts. Other than having the functions to update the owner, managers, Autonolas registries contract (to register code NFTs), and the DonatorBlacklist, the contract has the following methods.

1. initializeTokenomics function

   Input parameters

   a. `_olas:` address of the OLAS token contract
   b. `_treasury:` address of the Treasury contract
   c. `_depository:` address of the Depository contract
   d. `_ve:` address of the veOLAS token contract
   e. `_epochLen:` length of the epoch
   f. `_componentRegisty:` address of the component registry contract
   g. `_agentRegisty:` address of the agent registry contract
   h. `_serviceRegisty:` address of the service registry contract
   i. `_donatorBlacklist:` address of the donor blacklist contract

   Whether none of the input addresses is zero, the `_epochLen` isn't smaller than a minimum value and isn't larger than one year in seconds, and the call of this initializer does not happen more than one year later than the launch of the OLAS,

the function is successfully executed. Note that the initialization can be made only once.

The first time that this function is successfully called, the following is done:

- Initial storage variables are initialized, specifically `owner = msg.sender, _locked = 1, epsilonRate = 1e17, veOLASThreshold = 5_000e18`.
- Other passed input parameters are assigned, specifically, `olas = _olas, treasury = _treasury, depository = _depository, dispenser = _dispenser, ve = _ve, epochLen = uint32(_epochLen), componentRegistry = _componentRegistry, agentRegistry = _agentRegistry, serviceRegistry = _serviceRegistry, donatorBlacklist = _donatorBlacklist`
- it calculates the inflation per second for the launch year of the OLAS token (e.g. year zero).
- the block timestamp of the initializer function call is set as the end time of the zero epoch and the starting time of the first epoch.
- the epoch counter `epochConuter` is set to one (so epoch counter stats from one)
- the initial parameter `devPerCapital` e.g. the developers that can be funded with a unit of capital is set to `1e18`
- the initial inverse of the discount factor `idf` (e.g. 1+the interest rate) is set to `1e18`
- the numerator of the fraction's reward (in ETH) reserved for all the components `tp.unitPoints[0].rewardUnitFraction` is set to `66` and that reserved for all the agents `tp.unitPoints[1].rewardUnitFraction` is set to `34`
- a *unit of valuable code* is measured as `agentWeight` agents or `componentWeight` components. Specifically, the initial parameter of `agentWeight` is set to `1e18` and `componentWeight` is set to `2e18`
- the numerator of the maximum amount of the OLAS inflation reserved for bonding `_maxBondFraction` is set to `49`

- the numerator of the fraction of the OLAS inflation reserved for staking incentives of components `tp.unitPoints[0].topUpUnitFraction` is set to `34` and the one reserved for the agents `tp.unitPoints[1].rewardUnitFraction` is set to `17`.

2. tokenomicsImplementation function has no input parameter and allows to return of the contract address of the tokenomics implementation
3. changeTokenomicsImplementation function allows the contract owner to change the address of the tokenomics contract implementation.
   Input parameter
   a. `implementation:` address of the new tokenomics logic contract.
   If the function is called by the owner,
   - the implementation contract with the new address is stored under the designed storage slot of the tokenomics Proxy contract
   - the event `TokenomicsImplementationUpdated(implementation)` is emitted.
4. changeTokenomicsParameters function allows the contract owner to change several tokenomics parameters.
   Input parameter
   a. `_devsPerCapital:` (*larger than* `MIN_PARAM_VALUE`) number of valuable developers that can be paid per unit of capital per epoch
   b. `_epsilonRate:` value (larger than zero and smaller than 17e18) of the maximum interest rate that bonders can have on their bonds
   c. `_epochLen:` the value (*larger than or equal to* `MIN_EPOCH_LENGTH` *and smaller than or equal to* `ONE_YEAR`) of the length of one epoch in seconds
   d. `_veOLASThreshold:` the number (*non-zero*) of veOLAS that service owners or donators need to be whitelisted
   e. `componentWeight:` the number (*larger than* `MIN_PARAM_VALUE`) of components corresponding to one unit of code
   f. `agentWeight:` the number (*larger than* `MIN_PARAM_VALUE`) of agents corresponding to one unit of code

If the inputs are correctly chosen e.g. what is in the parentheses above is verified and the caller of the function is the contract owner

- it is set a flag signaling that tokenomics parameters are requested to be updated. This flag is used in such a way that the parameters will be updated at checkpoint and can be used from epoch number `eCounter+1`
- The event is emitted at the epoch number `TokenomicsParametersUpdateRequested(epochCounter + 1,_devsPerCapital, _epsilonRate, _epochLen, _veOLASThreshold, _componentWeight, _agentWeight).`

Note that if one of the inputs is provided with a value not verifying what is in the parenthesis, no change to such a parameter is made.

5. changeIncentiveFractions function allows the contract owner to change the share of donations accrued by the protocol reserved to component/agent rewards, or the share of the OLAS inflation reserved for bonding, or component/agent top-ups. Input parameters
    a. `_rewardComponentFraction:` fraction's numerator of donations accrued by the protocol reserved to component rewards (e.g. `_rewardComponentFraction/100*`*donations* is the share of donations reserved for components)
    b. `_rewardAgentFraction:` fraction's numerator of total epoch donations accrued by the protocol reserved to component rewards (e.g. `_rewardAgentFraction/100*`*epochDonations* is the share of the epoch donations reserved for components)
    c. `_maxBondFraction:` fraction's numerator of the epoch OLAS inflation reserved for the bonding mechanism (e.g. `_maxBondFraction/100*`epochOLASinflation is the share of inflation for the epoch reserved for bonding)
    d. `_topUpComponentFraction:` fraction's numerator of epoch OLAS inflation reserved for the component top-up (e.g.

`_topUpComponentFraction/100*`epochOLASinflation is the share of inflation for the epoch reserved for component top-ups)

e. `_topUpAgentFraction`: fraction's numerator of epoch OLAS inflation reserved for the agent top-up (e.g. `_topUpAgentFraction/100*`epochOLASinflation is the share of inflation for the epoch reserved for agent top-ups)

If the caller of the function is the contract owner and the inputs are correctly chosen, e.g. `_rewardAgentFraction+_rewardComponentFraction` and `_maxBondFraction+_topUpComponentFraction+_topUpAgentFraction` are smaller than or equal to 100

- in `eCounter` is stored the number of the next epoch at which the fractions can be updated
- it is set a flag signaling that tokenomics fractions are requested to be updated with the new input values. This flag is used in such a way the parameters will be updated at the checkpoint and can be used from epoch number `eCounter+1`
- The event is emitted at the epoch number `IncentiveFractionsUpdateRequested(eCounter,rewardAgentFraction,_rewardComponentFraction,_maxBondFraction,_topUpComponentFraction,_topUpAgentFraction).`

6. reserveAmountForBondProgram function allows the depository contract to reserve a certain amount of the OLAS inflation for new bonding programs.
   Input parameter
   a. `amount`: an amount of the OLAS that will be reserved when a new bonding program is created.

   If the input `amount` does not exceed the amount of the OLAS inflation schedule reserved for the bonding mechanism which is accounted for in the `eBond` variable) and the caller of the method is the Depository contract,
   - the reserved amount of OLAS for the next bonding programs is updated, e.g. `eBond-=amount`
   - the event `EffectiveBondUpdated(eBond)` is emitted.

7. refundBondProgram function allows the depository contract to refund a certain unused amount of the OLAS from closed bonding programs.
   Input parameter
   a. `amount`: an amount of the OLAS that was unused when some bonding programs are closed.

   If `eBond=effectiveBond+amount` is smaller than 2^96, where `amount` is the input parameter and `effectiveBond` is the amount of the OLAS inflation schedule reserved for the bonding mechanism, and the caller of the method is the Depository contract,
   - the reserved amount of OLAS for the next bonding programs is updated, e.g. `eBond=effectiveBond+amount`
   - the event `EffectiveBondUpdated(eBond)` is emitted.

8. _finalizeIncentivesForUnitId function allows finalizing the incentives for a specified agent or component Id.
   Note that, in terms of the formulas described in the section How the staking model for agents and component code is incentivized, what the algorithm does is
   A. multiply $Fcomps$ = `rewardUnitFraction/100` (resp. $Fagents$) times the *pending relative rewards* per component (resp. agent) unit which is the summand of equation (1) (resp. (2))
   B. multiply $\frac{TopUpC}{\sum_{1\leq k\leq w} r_k}$ = `totUpsUnitFraction/(100*sumUnitTopUpsOLAS)` (resp $\frac{TopUpA}{\sum_{1\leq k\leq w} r_k}$) times the *pending relative topUps* per component (resp. agent) unit which is the right-hand side summand of equation (3) (resp. (4))

   Input parameters
   a. `epochNum`: the epoch to finalize the incentive for
   b. `unitType`: type of the unit for which the incentive is finalized
   c. `unitId`: identified of the unit for which the incentive is finalized

The finalization described in A. is made there is no zero pending relative reward per the `unitId` with type `unitType`.
Similarly, the finalization described in B. is made when there is no zero pending relative top-up per the `unitId` with type `unitType`.

9. _trackServiceDonations function allows recording the donations which signal the appreciation for some services into the corresponding data structure.
   Input parameters
   a. `serviceIds`: the array of service identifiers for which appreciation is signaled with a donation
   b. `amounts`: the array of donations amounts sent to signal appreciation to the services with identifiers `serviceIds`
   c. `curEpoch`: number of the current epoch

   First it is checked which of the unitFraction (for both rewards and top-ups) is non-zero to identify the units for which the incentives should be accounted for.
   The first part has an algorithm that calculates
   - the *pending rewards and the pending tops* (when eligible for top-ups) per each of the components referenced in one of the services with identifier `serviceIds`.
     In terms of formulas described in the section How the staking model for agents and component code is incentivized, the summand in equation (1) is computed. Moreover, when eligible for top-ups, the summand on the right-hand side of equation (3) and the `sumUnitTopUpsOLAS` which is the summand at the denominator of the left-hand side factor of equation (3) are computed.
   - Similarly, the *pending rewards and the pending tops* (when eligible for top-ups) per each of the agents referenced in one of the services with identifier `serviceIds`.
     In terms of formulas described in the section How the staking model for agents and component code is incentivized, the summand in equation (2) is

computed. Moreover, when eligible for top-ups, the summand on the right-hand side of equation (4) and the `sumUnitTopUpsOLAS` which is the summand at the denominator of the left-hand side factor of equation (4) are computed.

Then, the second and final part has an algorithm that records the new component/agent code staked and referenced in appreciated services during the current epoch and the new owners of such component/agents.

10. trackServiceDonations function allows the treasury contract to track and record the donations received as a signal the appreciation for some services into the corresponding data structure.
    Input parameters
    a. `donator`: the address that sends a donation for the services with identifiers in `serviceIds`
    b. `serviceIds`: the array of service identifiers for which appreciation is signaled with a donation
    c. `amounts`: the array of donations amounts sent to signal appreciation to the services with identifiers `serviceIds`
    d. `donationETH`: overall donations in Ether received
    If the caller of the function is the treasury, the donator blacklist is enabled, and the address of `donator` isn't blacklisted
    - the current epoch number is accounted in `curEpoch`
    - the information that new amount `donationETH` has to be added to the endpoint of total donations received for the services is pushed in the map `mapEpochTokenomics[curEpoch]` e.g. `donationETH` is summed to the previous Ether amount accounted in `mapEpochTokenomics[curEpoch].epochPoint.totalDonationsETH`
    - a call to the following internal function is made `_trackServiceDonations(serviceIds, amounts, curEpoch)`.

11. _calculateIDF function allows the calculation of the inverse discount factor, e.g. (1+interest_rate) (cf. section How and when the bonding mechanism is incentivized).
    Input parameters
    a. `treasuryRewards`: the share of donations given to the treasury
    b. `numNewComponents`: number of the new components staked and referenced in appreciated services in the current epoch
    c. `numNewAgnts`: number of the new agents staked and referenced in appreciated services in the current epoch
    d. `numNewOwners`: number of new stakers of components/agents referenced in appreciated services in the current epoch.

    The method calculates

    $$\texttt{fKD} = f(K(n), D(n))/100 = d(n)/100 * (k(n) * K(n) + D(n))$$

    (where f is the chosen Product function) using
    - `treasuryRewards` in place of the input $K(n)$
    - `numNewOwners` in place of the input $D(n)$
    - `devsPerCapital` in place of $k(n)$
    - `codeUnits/denominator` in place of $d(n)/100$

    This method will output
    - `idf = 1e18 + fKD` when there is a small production of code, e.g. $f(K(n), D(n))/100$ is checked to be smaller than `epsilonRate`
    - `idf = 1e18 + epsilonRate` when there is a large production of code, e.g. $f(K(n), D(n))/100$ is checked to be larger than `epsilonRate`

    where `epsilonRate` is the maximum possible interest rate on a bond that the DAO governance wants to give.

12. checkpoint function allows recording of the global data and updating tokenomics parameters and/or fractions when changeTokenomicsParameters and/or changeIncentiveFractions method are called.

Firstly, it gets the (*non-zero*) address `implementation` which is the tokenomics logic contract written in the PROXY_TOKENOMICS address slot.

If this method is called more than one year later or less than the epoch's length since the last checkpoint, the function fails and returns `false`.
Otherwise, the following is done

- the donations accrued by the protocol are split (using the given fractions for the treasury and the agents/components rewards) between `treasuryRewards, componentRewards, agentRewards`.
- the inflation per epoch is computed and in particular, if the year change in the middle of an epoch it is necessary to adjust the inflation number to account for the year change (in particular `getInflationForYear` method of the TokenomicsConstants abstract contract is used in this case).
- the inflation per epoch is then split (using the given fractions for bonding and agents/components top-ups) between bonding and agents/component top-ups.
- If there is a flag signaling that the changeIncentiveFractions method was called, the fraction is updated with the newly requested fractions, and the event `IncentiveFractionsUpdated(eCounter+1)` is emitted. If there is no flag, the older fractions are used.
- If there is a flag signaling that the changeTokenomicsParameters method was called, the parameters are updated with the newly requested one (when correctly chosen), and the event `TokenomicsParametersUpdated(eCounter + 1` is emitted. If there is no flag, the old
- The max bond value is adjusted if the next epoch is going to be the year change epoch and the effective bond will be computed
- The inverse of the discount factor `IDF` is updated when the protocol accrues some donations and `IDF` is set to its default value `1e18` otherwise.
- Whether there are no rewards for the treasury or when a share of the received Ether donations is accounted for the treasury rewards and the `rebalanceTreasury` method of the Treasury contract is correctly

executed the event `EpochSettled(eCounter, incentives[1], accountRewards, accountTopUps)` is emitted and a new epoch starts. Note that, when there are rewards for the treasury but the `rebalanceTreasury` method is not successfully executed, the checkpoint is reverted and the new epoch does not start.

- `True` is returned when the checkpoint is correctly executed.

13. accountOwnerIncentives function allows the dispenser contract to get the calculated incentives for the owner of staked code.
    Input parameters
    a. `account`: the address of the owner of each unit of code (agents/components) with identifiers in `unitIds`
    b. `unitTypes`: array of unit types (agent or component) for which the incentives are accounted
    c. `unitIds`: array of unit identifiers for which the incentives are accounted for.

    If the caller of the method is the dispenser contract, the length of `unitTypes` and `unitIds` coincides, the value `unitTypes[i]` correctly corresponds to either an agent unit (e.g. 1) or component unit (e.g. 0), `unitIds` are given in strictly ascending order and none of them is bigger than the total number of units staked in the Autonolas registries, and the `account` is the owner of all the unit with identifiers in `unitIds` the following happens

    - `eCounter` accounts for the current epoch
    - `lastEpoch` accounts for the last epoch in which the incentives were accumulated
    - when the pending incentives were not yet finalized then the tokenomics method `_finalizeIncentivesForUnitId(lastEpoch, unitTypes[i], unitIds[i])` is called in order to finalize the incentives and in the map

`mapUnitIncentives[unitTypes[i]][unitIds[i]].lastEpoch` it is pushed the information that incentives were finalized

- `reward` accounts for accumulated rewards
- `topUp` accounts for accumulated top-ups
- the information that such rewards and top-ups balances were accrued is pushed in the map `mapUnitIncentives[unitTypes[i]][unitIds[i]]` by setting `mapUnitIncentives[unitTypes[i]][unitIds[i]].reward=0` and `mapUnitIncentives[unitTypes[i]][unitIds[i]].topUp=0`

14. getOwnerIncentives function allows having information on the incentives of the for staked code.
    Input parameters
    a. `account`: the address of the owner of each unit of code (agents/components) with identifiers in `unitIds`
    b. `unitTypes`: array of unit types (agent or component) for which the the information of incentives is requested
    c. `unitIds`: array of unit identifiers for which the information of incentives is requested.

    If the length of `unitTypes` and `unitIds` coincides, the value `unitTypes[i]` correctly corresponds to either an agent unit (e.g. 1) or component unit (e.g. 0), `unitIds` are given in strictly ascending order and none of them is bigger than the total number of units staked in the Autonolas registries one of the following happens
    - the `account` is the owner of all the unit with identifiers in `unitIds` and the pending incentives were already finalized the accumulated rewards `reward` and accumulated top-ups `topUp` are returned
    - the `account` is the owner of all the units with identifiers in `unitIds` and the pending incentives were not finalized yet these are finalized and the accumulated rewards `reward` and accumulated top-ups `topUp` are returned.

- the `account` is not the owner of one of the units with identifiers in `unitIds` and the call of the function is reverted.

15. getInflationPerEpoch function has no input parameter and allows getting information on the inflation for the last epoch.

16. getUnitPoint function allows getting the tokenomics point information for a specific `epoch` and a specific `unitType`.
    Input parameters
    a. `epoch`: number of the epoch for with the point is requested (*no larger than the current epoch number*)
    b. `unitType`: type of the unit for which the point is requested (*e.g. 0 for component type and 1 for agent type*).
    Whether the parameters are *correctly* added a non-zero map `mapEpochTokenomics[epoch].unitPoints[unitType]` is returned.

17. getIDF function allows getting the inverse of the discount factor `IDF` (with 18 decimals) for a specific `epoch`.
    Input parameter
    a. `epoch`: number of the epoch for with the value is requested
    It is returned
    - `mapEpochTokenomics[epoch].epochPoints.idf` when non-zero
    - `1e18` e.g. the default value of `idf` otherwise.
    In particular, even if the `epoch` is larger than the current one, a non-zero value is returned.

18. getLastIDF function has no input parameter and allows getting the inverse of the discount factor `IDF` (with 18 decimals) for the last epoch.