StakingRouter

- Source code
- Deployed contract

StakingRouter is a registry of staking modules, each encapsulating a certain validator subset, e.g. the Lido DAO-curated staking module. The contract allocates stake to modules, distributes protocol fees, and tracks relevant information.

What is StakingRouter?

StakingRouter is a top-level controller contract for staking modules. Each staking module is a contract that, in turn, manages its own subset of validators, e.g. theCurated staking module is a set of Lido DAO-vetted node operators. Such modular design opens the opportunity for anyone to build a staking module and join the Lido staking platform, including permissionless community stakers, DVT-enabled validators or any other validator subset, technology or mechanics.

StakingRouter performs a number of functions, including:

- maintaining a registry of staking modules,
- allocating stake to modules, and
- distributing protocol fees.

Module Management

Registering a module

Modules are registered with StakingRouter through the Lido DAO voting process. To be considered by the governance, the applying module contract should implement the appropriate module interface, meet security requirements, and have a fee structure aligned with the Lido protocol sustainability. Once voted in, the module starts receiving stake and protocol fees.

Staking modules are registered using theaddStakingModule function, providing details such as:

- the module's name: a human-readable name;
- address of the deployed staking module contract;
- target share, a relative hard cap on deposits within Lido;
- module fee, a percentage of staking rewards to be awarded to the module,
- and treasury fee, a percentage of staking rewards to be directed to the protocol treasury.

Pausing modules

Each staking module has a status: a state that determines whether the module can perform deposits and receive rewards:

- Active
- , can make deposits and receives rewards,
- DepositsPaused
- , deposits are not allowed but receives rewards,
- Stopped
- , cannot make deposits and does not receive rewards.

enum

StakingModuleStatus

{ Active ,

// deposits and rewards allowed DepositsPaused ,

// deposits NOT allowed, rewards allowed Stopped // deposits and rewards NOT allowed }

Exited and stuck validators

When the withdrawal requests demand exceed buffered ether sitting in Lido together with projected rewards, the protocol signals to node operators to start exiting validators to cover the withdrawals. In this connection, StakingRouter distinguishes two types of validator states:

- exited
- validators,
- and stuck validators, meaning those validators that failed to comply with the exit signal.

StakingRouter keeps track of both of these types of validators in order to correctly allocate stake and penalize stuck validators. These stats are kept up to date via the oracles submitting the data to the contract.

Stake allocation

StakingRouter carries out a vital task of distributing depositable ether to staking modules in a manner that aligns with their growth targets set by the DAO. This design ensures a regulated and controlled growth for the modules that have been newly integrated into the system. The principles governing this methodology are comprehensively discussed inADR: Staking Router .

Deposit

The deposit workflow involves submitting batches of 32 ether deposits, along with associated validator keys, toDepositContract in one transaction. Given that each staking module handles its own deposits, every batch deposit is restricted to keys originating from a single module.

The deposit operation is, at its core, a sequence of contract calls sparked by an off-chain software, thedepositor bot . This bot gathers guardian messages to confirm that there are no pre-existing keys in the registry that could take advantage of thefrontrunning vulnerability . Once the necessary quorum of guardians is reached, the bot forwards these messages along with the module identifier to the DepositSecurityModule (not to be confused with a staking module). This contract first verifies the messages, then initiates the deposit function on Lido, passing along the maximum number of deposits that the current block size can accommodate. Subsequently, Lido calculates the maximum number of deposits that can be included in the batch based on the existing deposit buffer, and triggers StakingRouter's deposit function. The StakingRouter then determines the distribution of buffered ether to the module that will use its keys in the deposit, and finally executes the batch deposit.

Thedeposit function begins by verifying the sender's identity and checking the withdrawal credentials and the status of the staking module. After these checks, it updates the contract's local state by recording the current timestamp and block number as the last deposit time and block for the staking module. It then emits an event to log the deposit transaction, and checks if the deposit value matches with the required total deposit size. If there are deposits to be made, it gets deposit data (public keys and signatures) from the staking module contract. It then makes deposits toDepositContract using the obtained data. Finally, it confirms that all deposited ETH has been correctly transferred to the contract by comparing the contract's ether balance before and after the deposit transaction.

Allocation algorithm

The algorithm used for the allocation process is designed to consider various factors, including the limit of deposits per transaction, the quantity of ether ready for deposit, the active and available keys within each module, and each module's target share. It then estimates the maximum deposit based on these parameters. The algorithm also identifies the key limits for all modules, and initiates the allocation of keys, starting with the module that has the least active keys. This continues until there is no more ether to allocate or when all the modules have reached their capacity. At the end of the process, any remaining ether is returned.

The allocation function uses theMinFirstAllocationStrategy algorithm to allocate new deposits among different staking modules.

Here is a breakdown of the process:

1. The function takes as input_depositsToAllocate
2. , which represents the amount of new deposits that need to be allocated among the staking modules.
3. It starts by calculating the total active validators in the system (totalActiveValidators
4.) and loading the current state of staking modules into a cache (stakingModulesCache
5.).
6. It then creates anallocations
7. array of the same size as the number of staking modules, with each index in this array representing a staking module's current allocation (i.e., the current number of active validators in that module).
8. If there are staking modules available (stakingModulesCount > 0
9.), the function goes into the allocation process:
10. a. It calculates a new estimated total of active validators, adding the new deposits to the total active validators (totalActiveValidators += _depositsToAllocate
11.).
12. b. It creates acapacities
13. array of the same size as the number of staking modules. Each entry in this array represents the maximum capacity of a particular staking module, i.e., the maximum number of validators that module can have. This is calculated as the minimum of either:
14.
    - The target number of validators for a module, which is based on a desired target share (stakingModulesCache[i].targetShare * totalActiveValidators / TOTAL_BASIS_POINTS

15.
    -), or
16.
    - The sum of the current active validators and the available validators in the module (stakingModulesCache[i].activeValidatorsCount + stakingModulesCache[i].availableValidatorsCount
17.
    -).
18. c. Finally, it calls theallocate
19. function fromMinFirstAllocationStrategy
20. , passing in theallocations
21. ,capacities
22. , and_depositsToAllocate
23. . The amount successfully allocated is stored inallocated
24. .

To sum up, this function is using theMinFirstAllocationStrategy algorithm to distribute new deposits (validators) across different staking modules in a way that prioritizes filling the least populated modules, while taking into account each module's target share and capacity. The resulting allocations and total allocated amount are then returned for further use.

Fee distribution

The fee structure is set independently in each module. There are two components to the fee structure: the module fee and the treasury fee, both specified as percentages (basis points). For example, a 5% (500 basis points) module fee split between node operators in the module and a 5% (500 basis points) treasury fee sent to the treasury. Additionally,StakingRouter utilizes a precision factor of 100 * 10^18 for fees that prevents arithmetic operations from truncating the fees of small modules.

Because the protocol does not currently account for per-validator performance, the protocol fee is distributed between modules proportionally to active validators and the specified module fee. For example, a module with 75% of all validators in the protocol and a 5% module fee will receive 3.75% of the total rewards across the protocol. This means that if the modules' fee and treasury fee do not exceed 10%, the total protocol fee will not either, no matter how many modules there are. There is also an edge case where the module is stopped for emergency while its validators are still active. In this case the module fee will be transferred to the treasury and once the module is back online, the rewards will be returned back to the module from the treasury.

The distribution function itself works as follows:

1. The function first loads the current state of the staking modules into a cache (_loadStakingModulesCache
2.) and calculates the number of these modules (stakingModulesCount
3.).
4. If there are no staking modules or no active validators in the system, it returns an empty response.
5. Otherwise, it initializes arrays to store the module IDs (stakingModuleIds
6.), the addresses of reward recipients (recipients
7.), and the fees of each recipient (stakingModuleFees
8.). It also sets theprecisionPoints
9. to a constantFEE_PRECISION_POINTS
10. , which represents the base precision number that constitutes 100% fee.
11. Then it loops through each staking module. For each module that has at least one active validator, it:
12.
    - Stores the module ID and recipient address in the respective arrays.
13.
    - Calculates thestakingModuleValidatorsShare
14.
    - , which is the proportion of total active validators that are part of this staking module.
15.
    - Calculates thestakingModuleFee
16.
    - as the product ofstakingModuleValidatorsShare
17.
    - and the fee of the staking module divided byTOTAL_BASIS_POINTS
18.
    - (i.e., the proportion of the staking module's fee to the total possible fees). If the module is not stopped, this fee is stored in thestakingModuleFees
19.
    - array.
20.
    - Adds tototalFee
21.
    - the sum of the staking module's fee and a fee going to the treasury (calculated similarly tostakingModuleFee

22.
    -), where the treasury is a central pool of funds.
23. After looping through all modules, it makes an assertion thattotalFee
24. doesn't exceed 100% (represented byprecisionPoints
25.).
26. If there are staking modules with no active validators, it shrinks thestakingModuleIds
27. ,recipients
28. , andstakingModuleFees
29. arrays to exclude those modules.

Finally, the function returns five arrays/values:recipients ,stakingModuleIds ,stakingModuleFees ,totalFee , andprecisionPoints . These give the caller an overview of how rewards are distributed amongst the staking modules.

Helpful links

- Staking Router ADR
- Staking Router LIP
- Lido on Ethereum Validator Exits Policy

View methods

getLido

Returns the address of the coreLido contract.

function

getLido ( )

public

view

returns

(address)

getStakingModules

Returns the list of structs of all registered staking modules. Each staking module has an associated data structure,

struct

StakingModule

{ uint24 id ; address stakingModuleAddress ; uint16 stakingModuleFee ; uint16 treasuryFee ; uint16 targetShare ; uint8 status ; string name ; uint64 lastDepositAt ; uint256 lastDepositBlock ; uint256 exitedValidatorsCount } function

getStakingModules ( )

external

view

returns

(StakingModule [ ]

memory res) Returns:

Name Type Description res StakingModule[] list of structs of all registered staking modules

getStakingModuleIds

Returns the list of ids of all registered staking modules.

function

getStakingModuleIds ( )

public

view

returns

(uint256 [ ]

memory stakingModuleIds) Returns:

Name Type Description stakingModuleIds uint256[] list of id of all staking modules

getStakingModule

Returns the struct of the specified staking module by its id.

function

getStakingModule (uint256 _stakingModuleId)

public

view

returns (StakingModule memory) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description StakingModule staking module information

getStakingModulesCount

Returns the number of registered staking modules.

function

getStakingModulesCount ( )

public

view

returns

(uint256)

hasStakingModule

Return a boolean value indicating whether a staking module with the specified id is registered.

function

hasStakingModule (uint256 _stakingModuleId)

external

view

returns

(bool) Parameters:

Name Type Description _stakingModuleId uint256 staking module id

getStakingModuleStatus

Return the status of the staking module.

function

getStakingModuleStatus (uint256 _stakingModuleId)

public

view

returns

(StakingModuleStatus) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description StakingModuleStatus status of the staking module

getStakingModuleSummary

Returns the struct containing a short summary of validators in the specified staking module, as shown below,

struct

StakingModuleSummary

{ uint256 totalExitedValidators ; uint256 totalDepositedValidators ; uint256 depositableValidatorsCount ; } function getStakingModuleSummary(uint256 _stakingModuleId) public view returns (StakingModuleSummary) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description StakingModuleSummary summary of the staking module's validators

getNodeOperatorSummary

Returns the summary of a node operator from the staking module, as shown below,

struct NodeOperatorSummary { bool isTargetLimitActive; uint256 targetValidatorsCount; uint256 stuckValidatorsCount; uint256 refundedValidatorsCount; uint256 stuckPenaltyEndTimestamp; uint256 totalExitedValidators; uint256 totalDepositedValidators; uint256 depositableValidatorsCount; } function

getNodeOperatorSummary (uint256 _stakingModuleId , uint256 _nodeOperatorId)

public

view

returns

(NodeOperatorSummary) Parameters:

Name Type Description _stakingModuleId uint256 staking module id _nodeOperatorId uint256 node operator id Returns:

Name Type Description NodeOperatorSummary summary of the node operator

getAllStakingModuleDigests

Returns the digests of all staking modules, as show below,

struct

StakingModuleDigest

{ uint256 nodeOperatorsCount ; uint256 activeNodeOperatorsCount ; StakingModule state ; StakingModuleSummary summary ; } function getAllStakingModuleDigests() external view returns (StakingModuleDigest[]) Returns:

Name Type Description StakingModuleDigest[] array of staking module digests

getStakingModuleDigests

Returns the digest of the specified staking modules.

function getStakingModuleDigests(uint256[] memory _stakingModuleIds) public view returns (StakingModuleDigest[]) Parameters:

Name Type Description _stakingModuleIds uint256[] array of staking module ids Returns:

Name Type Description StakingModuleDigest[] array of staking module digests

getAllNodeOperatorDigests

Returns the digests of all node operators in the specified staking module,

struct

NodeOperatorDigest

{ uint256 id ; bool isActive ; NodeOperatorSummary summary ; } function getAllNodeOperatorDigests(uint256 _stakingModuleId) external view returns (NodeOperatorDigest[]) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description NodeOperatorDigest[] array of node operator digests

getNodeOperatorDigests

Returns the digests for the specified node operators in the staking module.

function

getNodeOperatorDigests (uint256 _stakingModuleId ,

uint256 [ ]

memory _nodeOperatorIds)

public

view

returns

(NodeOperatorDigest [ ]) Parameters:

Name Type Description _stakingModuleId uint256 staking module id _nodeOperatorIds uint256[] array of node operator ids Returns:

Name Type Description NodeOperatorDigest[] array of node operator digests

getStakingModuleIsStopped

Return a boolean value whether the staking module is stopped.

function

getStakingModuleIsStopped (uint256 _stakingModuleId)

external

view

returns

(bool) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description bool true if the staking module is stopped, false otherwise

getStakingModuleIsDepositsPaused

Return a boolean value whether deposits are paused for the staking module.

function getStakingModuleIsDepositsPaused(uint256 _stakingModuleId) external view returns (bool) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description bool true if deposits are paused for the staking module, false otherwise

getStakingModuleIsActive

Return a boolean value whether the staking module is active.

function getStakingModuleIsActive(uint256 _stakingModuleId) external view returns (bool) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description bool true if the staking module is active, false otherwise

getStakingModuleNonce

Get the nonce of a staking module.

function getStakingModuleNonce(uint256 _stakingModuleId) external view returns (uint256) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description uint256 nonce of the staking module

getStakingModuleLastDepositBlock

Get the block number of the last deposit to the staking module.

function getStakingModuleLastDepositBlock(uint256 _stakingModuleId) external view returns (uint256) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description uint256 block number of the last deposit

getStakingModuleActiveValidatorsCount

Returns the number of active validators in the staking module.

function getStakingModuleActiveValidatorsCount(uint256 _stakingModuleId) external view returns (uint256 activeValidatorsCount) Parameters:

Name Type Description _stakingModuleId uint256 staking module id Returns:

Name Type Description uint256 number of active validators

getStakingModuleMaxDepositsCount

Calculates the maximum number of deposits a staking module can handle based on the available deposit value.

function getStakingModuleMaxDepositsCount(uint256 _stakingModuleId, uint256 _maxDepositsValue) public view returns (uint256) Parameters:

Name Type Description _stakingModuleId uint256 staking module id _maxDepositsValue uint256 maximum amount of deposits based on the available ether Returns:

Name Type Description uint256 maximum number of deposits that can be made using the given staking module

getStakingFeeAggregateDistribution

Returns the total fee distribution proportion.

function getStakingFeeAggregateDistribution() public view returns (uint96 modulesFee, uint96 treasuryFee, uint256 basePrecision) Returns:

Name Type Description modulesFee uint96 total fees for all staking modules treasuryFee uint96 total fee for the treasury basePrecision uint256 base precision number, a value corresponding to the full fee

getStakingRewardsDistribution

Get the shares table.

function

getStakingRewardsDistribution ( )

public

view

returns

(address [ ]

memory recipients , uint256 [ ]

memory stakingModuleIds , uint96 [ ]

memory stakingModuleFees , uint96 totalFee , uint256 precisionPoints) Returns:

Name Type Description recipients address[] total staking module addresses stakingModuleIds uint256[] staking module ids stakingModuleFees uint96[] staking module fees totalFee uint96[] total fee precisionPoints uint256 base precision number, a value corresponding to the full fee

getDepositsAllocation

Calculates the deposit allocation after the distribution of the specified number of deposits using the Min-First algorithm.

function

getDepositsAllocation (uint256 _depositsCount)

external

view

returns

(uint256 allocated ,

uint256 [ ]

memory allocations) Parameters:

Name Type Description _depositsCount uint256 deposits to allocate between staking modules Returns:

Name Type Description allocated uint256 total staking module addresses allocations uint256[] array of new total deposits between staking modules

getWithdrawalCredentials

Get the withdrawal credentials.

function

getWithdrawalCredentials ( )

public

view

returns

(bytes32) Returns:

Name Type Description bytes32 withdrawal credentials

Write methods

addStakingModule

Register a staking module.

function

addStakingModule (string

calldata _name , address _stakingModuleAddress , uint256 _targetShare , uint256 _stakingModuleFee , uint256 _treasuryFee)

external ; Parameters:

Name Type Description _name string human-readable name of the module _stakingModuleAddress address address of the module contract _targetShare uin256 module target share _stakingModuleFee uin256 module fee _treasuryFee uint256 module treasury fee

updateStakingModule

Register a staking module.

function

updateStakingModule (uint256 _stakingModuleId , uint256 _targetShare , uint256 _stakingModuleFee , uint256 _treasuryFee)

external ; Parameters:

Name Type Description _stakingModuleId address id of the module _targetShare uin256 updated module target share _stakingModuleFee uin256 updated module fee _treasuryFee uint256 updated module treasury fee

updateTargetValidatorsLimits

Updates the limit of the validators that can be used for deposit.

function

updateTargetValidatorsLimits (uint256 _stakingModuleId , uint256 _nodeOperatorId , bool _isTargetLimitActive , uint256 _targetLimit)

external ; Parameters:

Name Type Description _stakingModuleId uin256 id of the module _nodeOperatorId uin256 id of the node operator _isTargetLimitActive bool active flag _targetLimit uint256 target limit of the node operator

updateRefundedValidatorsCount

Updates the number of the refunded validators in the staking module with the given node operator id.

function

updateRefundedValidatorsCount (uint256 _stakingModuleId , uint256 _nodeOperatorId , uint256 _refundedValidatorsCount ,)

external ; Parameters:

Name Type Description _stakingModuleId uin256 id of the module _nodeOperatorId uin256 id of the node operator _refundedValidatorsCount uint256 new number of refunded validators of the node operator

reportRewardsMinted

Updates the number of the refunded validators in the staking module with the given node operator id.

function

reportRewardsMinted (uint256 [ ]

calldata _stakingModuleIds , uint256 [ ]

calldata _totalShares ,)

external ; Parameters:

Name Type Description _stakingModuleIds uin256[] list of the reported staking module ids _totalShares uin256[] total shares minted to the given staking modules

updateExitedValidatorsCountByStakingModule

Update total numbers of exited validators for staking modules with the specified module ids.

function

reportRewardsMinted (uint256 [ ]

calldata _stakingModuleIds , uint256 [ ]

calldata _exitedValidatorsCounts ,)

external ; Parameters:

Name Type Description _stakingModuleIds uin256[] list of the reported staking module ids _exitedValidatorsCounts uin256[] new counts of exited validators for the specified staking modules

reportStakingModuleExitedValidatorsCountByNodeOperator

Updates exited validators counts per node operator for the staking module with the specified id.

function

reportStakingModuleExitedValidatorsCountByNodeOperator (uint256 _stakingModuleId , bytes

calldata _nodeOperatorIds , bytes

calldata _exitedValidatorsCounts ,)

external ; Parameters:

Name Type Description _stakingModuleId uin256 staking module id _nodeOperatorIds bytes ids of the node operators _exitedValidatorsCounts bytes new counts of exited validators for the specified node operators

unsafeSetExitedValidatorsCount

Sets exited validators count for the given module and given node operator in that module without performing critical safety checks, e.g. that exited validators count cannot decrease.

function

unsafeSetExitedValidatorsCount (uint256 _stakingModuleId , bytes

calldata _nodeOperatorIds , bool _triggerUpdateFinish , ValidatorsCountsCorrection memory _correction)

external ; whereValidatorsCountsCorrection is a struct as seen below,

struct

ValidatorsCountsCorrection

{ /// @notice The expected current number of exited validators of the module that is /// being corrected. uint256 currentModuleExitedValidatorsCount ; /// @notice The expected current number of exited validators of the node operator /// that is being corrected. uint256 currentNodeOperatorExitedValidatorsCount ; /// @notice The expected current number of stuck validators of the node operator /// that is being corrected. uint256 currentNodeOperatorStuckValidatorsCount ; /// @notice The corrected number of exited validators of the module. uint256 newModuleExitedValidatorsCount ; /// @notice The corrected number of exited validators of the node operator. uint256 newNodeOperatorExitedValidatorsCount ; /// @notice The corrected number of stuck validators of the node operator. uint256 newNodeOperatorStuckValidatorsCount ; } Parameters:

Name Type Description _stakingModuleId uin256 staking module id _nodeOperatorIds bytes ids of the node operators _triggerUpdateFinish bool flag to callonExitedAndStuckValidatorsCountsUpdated on the module after applying the corrections _correction ValidatorsCountsCorrection correction details

reportStakingModuleStuckValidatorsCountByNodeOperator

Updates stuck validators counts per node operator for the staking module with the specified id.

function

reportStakingModuleStuckValidatorsCountByNodeOperator (uint256 _stakingModuleId , bytes

calldata _nodeOperatorIds , bytes

calldata _stuckValidatorsCounts ,)

external ; Parameters:

Name Type Description _stakingModuleId uin256 staking module id _nodeOperatorIds bytes ids of the node operators _stuckValidatorsCounts bytes new counts of stuck validators for the specified node operators

onValidatorsCountsByNodeOperatorReportingFinished

Post-report hook called by the oracle when the second phase of data reporting finishes, i.e. when the oracle submitted the complete data on the stuck and exited validator counts per node operator for the current reporting frame.

function

onValidatorsCountsByNodeOperatorReportingFinished ( )

external ; Edit this page Previous OracleDaemonConfig Next NodeOperatorsRegistry