Rocket Pool Documentation

Release latest

Oct 15, 2019

Contents

Rocket Pool is a decentralised Ethereum staking network built to be compatible with the Ethereum 2.0 beacon chain.

CHAPTER 1

Contents

1.1 Introduction to Rocket Pool

1.1.1 What does Rocket Pool do?

Rocket Pool is a first of its kind decentralised staking network for Ethereum. An excellent guide to Rocket Pool for new users can be found in our FAQ 101 guide here https://medium.com/rocket-pool/rocket-pool-101-faq-ee683af10da9

It has several objectives within the Ethereum landscape, each relevant to a different audience. It aims to:

1. Provide a network of decentralised nodes to perform proof of stake validation services for the Ethereum network;
2. Allow users who don't possess the minimum Ether required to become a validator, or the necessary technical skills to run a validation node, to participate in staking and earn rewards;
3. Allow users and businesses to run a validation node for ETH2 with only 16 ETH in the Rocket Pool network to earn rewards and additional income;
4. Provide any third party businesses with seamless staking services for their users via the free to access and use Rocket Pool API.

1.1.2 Contents

The following sections give a high-level overview of the various concepts used in Rocket Pool and how they work:

Staking

Overview

Staking is not unique to Rocket Pool, it's a feature coming to Ethereum 2.0 in the near future. Ethereum 2.0 will include a new chain called the Beacon Chain, which anyone can become a validator on by making a deposit (or "stake") of ether. Validators are responsible for ensuring the integrity of the Ethereum network, and earn interest on their deposit by doing so.

The Beacon Chain will require a large sum of ether in order to become a validator, which many may not have access to. Rocket Pool aims to solve this problem by combining small amounts of ether from multiple users so that they can stake together!

Staking Durations

Rocket Pool provides a few different staking duration options to node operators and users. Whenever someone makes a deposit into Rocket Pool, they specify the duration that they want to stake for. Each staking duration has its own deposit queue, and deposits are only matched to other deposits with the same duration. This ensures that everyone who gets pooled together is on the same page regarding when they want to withdraw from the Beacon Chain and claim their rewards.

Deposits

Overview

Deposits made into Rocket Pool are tracked closely from the moment they enter the network. When a user makes a deposit, it gets assigned a unique ID, and all of its information is recorded. This includes the total amount deposited, how much of it is in the deposit queue, and how much has been assigned to minipools, refunded, and withdrawn. We also keep a list of every minipool the deposit gets assigned to along with how much of the deposit was assigned to it.

The Deposit Queue

When deposits are made into Rocket Pool, they first enter a deposit queue. Each staking duration has its own deposit queue and is processed separately. The queue is first-in, first-out - so whoever deposited first gets assigned to minipools first. Whenever a deposit is made, the queue gets processed as long as there are minipools available for assignment. It also gets processed when a node operator creates a new minipool.

If the queue is moving too slowly and a user wants to pull out, they can refund their deposit from the queue. However, if any of their deposit has already been assigned to minipools, it's too late to withdraw that - they are locked in.

The deposit queue has a maximum size (in ether) to prevent too large of a back-log from forming. Once the queue gets close to its maximum size, deposit sizes are limited so that they don't exceed it. If it fills up entirely and there are no minipools available for assignment, deposits will be disabled until more are added.

Chunking

Deposits over a certain size (4 ETH) are broken into "chunks" and spread out over different minipools instead of being put in one. This distributes risk over the network more evenly so that users stand to lose less if a node operator experiences problems. For example, a deposit of 16 ETH would be assigned to four minipools (as long as there are at least four available) instead of one. If a node operator owning one of those minipools has problems, only 25% of the deposit is in danger.

Fees

Fees are all taken as a percentage of the rewards that a deposit earns, never from the initial deposit itself. Fees are primarily taken by the node operator, as they provide the computation power which earns rewards in the first place. Rocket Pool also charge a small fee to users (not the node operator), and the group which a user deposited with may charge a fee as well.

The Rocket Pool and group fees for each deposit are "locked in" when it gets assigned to a minipool. The node operator fee is "locked in" when the minipool begins staking on the beacon chain. This ensures that fees can't be increased after a user has begun staking, taking more than they were comfortable with initially.

When a user withdraws from a minipool, the node operator's fee and Rocket Pool's fee are calculated and deducted first. Then the group's fee is calculated based on the remaining rewards, and deducted. If the minipool did not earn a profit, no fees are taken and the user is reimbursed as much as possible.

Rewards & the rETH Token

During Phases 0 and 1 of the Ethereum 2.0 rollout, withdrawals from the Beacon Chain will not be implemented. Rocket Pool aims to provide liquidity to node operators and users through the design of the rETH token, so they can access their rewards before Phase 2 is launched in the future.

When a minipool's validator finishes staking on the Beacon Chain, the minipool is marked as ready for withdrawal, and rETH tokens equal to the validator's final balance are minted to it. The node operator and users staking in that minipool can then withdraw their share of rETH tokens from it and spend them as they wish. rETH tokens are backed by Beacon Chain ether 1:1, and should trade on the open market for slightly less than 1 ETH in value.

When Phase 2 of the Ethereum 2.0 rollout is launched, users holding rETH will be able to swap it for Beacon Chain ether via a contract on a shard. This effectively burns the rETH, removing it from circulation.

Backup Withdrawal Addresses

After making a deposit, a user can set a backup withdrawal address on it in case they lose access to their main address. This allows users to withdraw using a different wallet than the one they used to make their deposit. Withdrawals can only be made from backup addresses after a certain amount of time has passed and the deposit is still unclaimed.

Groups

Overview

All user deposits into the Rocket Pool network are made through a group. An organisation can register as a group with Rocket Pool, allowing them to move funds into the network for staking. This effectively allows them to use Rocket Pool as staking infrastructure, using their own deposit processes and application front-ends. Rocket Pool employs an "eat your own dog food" approach, using its own registered group for processing user interactions.

Group Depositors and Withdrawers

Groups register Depositor and Withdrawer contracts which their users can use to deposit ether into Rocket Pool, and withdraw their rewards as rETH. A group can have any number of each, allowing them to funnel user interactions from various sources with custom behaviour. Rocket Pool supplies default contracts if a group doesn't require any custom deposit and withdrawal processes.

Nodes

Overview

Nodes are the workhorses of the Rocket Pool network. Anyone can register as a node with Rocket Pool, and begin staking. When a node operator wants to stake, they deposit 16 ETH into Rocket Pool, which is matched with 16

ETH of user deposits. The node performs all of the validation duties required by the Ethereum network, and earns a percentage of the users' rewards as compensation.

Nodes constantly check in with the network to indicate that they are doing their job. If a node fails to check in for a long time, it will be deactivated by other nodes. This won't prevent it from validating (in case it still is), but will stop users from being assigned to any new minipools it creates.

The RPL Token

Nodes must also deposit an amount of RPL (Rocket Pool tokens) in order to stake. RPL is a token designed to regulate the capacity of the Rocket Pool network.

The amount of RPL required for a deposit adjusts dynamically based on demand for node operators. During periods of low utilisation (when there are lots of available minipools and the network doesn't need more), the RPL required to stake is higher. During periods of high utilisation (when there are few available minipools left and more are needed for users), the RPL required is lower, and can even drop to zero.

Node operators get their RPL back once their minipools finish staking and they withdraw from them. It is "locked up" while staking, rather than spent permanently. This means that node operators only need to buy more RPL if they intend to stake more ether at the same time.

The Node Fee

The fee charged to users by node operators is determined democratically by the whole network, rather than being different for each individual operator. This ensures that user deposits don't get assigned to a node which is charging more than others.

When nodes check in with the network, they also vote to increase or decrease the current node fee (or leave it as is). Every 24 hours, the number of votes for each option is tallied, the winning vote is enacted, and a new voting cycle begins. The node fee only increases or decreases by 0.5% at a time, in order to prevent large swings in the fee.

Node operators are naturally incentivised to increase the fee to a certain level to maximise their profits. However, increasing it too high would mean that users stop using the network and node operators start losing ether. In this way, the network reaches a natural equilibrium.

Watchtower Nodes

Some special nodes owned by Rocket Pool and trusted partners are designated as "watchtower" nodes. Watchtower nodes are responsible for reporting the Beacon Chain state back to the PoW chain.

Specifically, they report when a minipool's validator on the Beacon Chain exits, and when it is ready for withdrawal. The minipool is updated accordingly, so that the Rocket Pool network can track its progress. When it's ready for withdrawal, the watchtower node also mints rETH to the minipool matching the validator's final balance, which can be withdrawn by users and the node operator.

Minipools

Overview

Minipools are the matchmaking service at the heart of the Rocket Pool network. Whenever a node operator deposits ether into Rocket Pool, they create a minipool. This minipool contains the node operator's ether, and accepts another 16 ETH of user deposits. As users deposit into Rocket Pool, their deposits move through a queue until they are

randomly assigned to one or more minipools. Users are also matched based on how long they agree to stake for, as everyone needs to withdraw together.

Minipools keep a strict record of all the funds deposited into them, and what has been withdrawn back out. This includes the node operator's deposit, and the deposits of any users assigned to it. This way, the minipool knows how much to give back to everyone once it has finished staking. They also record a lot of other information about the node which created them, their staking status, and more.

The Active Minipool Set

Rocket Pool maintains an "active minipool set", consisting of four minipools chosen pseudo-randomly from all available nodes. This number is chosen to strike the right balance between dispersing risk (by splitting deposits up between minipools), and getting deposits staking quickly. If the active minipool set size was too small, deposits wouldn't be split up between enough minipools. If it was too large, it would take a long time for all minipools in the active set to fill up and begin staking.

Whenever the deposit queue is processed, deposits at the start of the queue are split up into 4 ETH chunks and assigned to minipools in the active set, in rotation. Once all minipools have filled up and begun staking, a new active set is chosen. If only one node has minipools available for assignment, it's possible that the active minipool set will only consist of a single pool.

Stalled Minipools

If a minipool is assigned some user deposits but fails to begin staking for a long time (i.e. if no new deposits are made), it will be marked as stalled. When a minipool stalls, the node operator and users who deposited to it can all refund their ether and RPL. No fees are charged in this case, since the network failed to provide a reliable staking service.

1.2 The Rocket Pool Smart Contracts

1.2.1 Introduction

The Rocket Pool Smart Contracts form the foundation of the Rocket Pool network. They are the base layer of infrastructure which all other elements of the network are built on top of, including the JavaScript library, the Smart Node software stack, and all web or application interfaces.

Direct interaction with the contracts is usually not necessary, and is facilitated through the use of other software (such as the JavaScript library). This section provides a detailed description of the contract design, and information on how to build on top of Rocket Pool via the API, for developers wishing to extend it. All code examples are given as Solidity v0.5.0.

1.2.2 Contents

Contract Design & Upgradability

Architecture

The Rocket Pool network contracts are built with upgradability in mind, using a hub-and-spoke architecture. The central hub of the network is the `RocketStorage` contract, which is responsible for storing the state of the entire network. This is implemented through the use of maps for key-value storage, and getter and setter methods for reading and writing values for a key.

The `RocketStorage` contract also stores the addresses of all other network contracts (keyed by name), and restricts data modification to those contracts only. Using this architecture, the network can be upgraded by deploying new versions of an existing contract, and updating its address in storage. This gives Rocket Pool the flexibility required to fix bugs or implement new features to improve the network.

Interacting With Rocket Pool

To begin interacting with the Rocket Pool network, first create an instance of the `RocketStorage` contract using its interface:

```
import "RocketStorageInterface.sol";

contract Example {

    RocketStorageInterface rocketStorage = RocketStorageInterface(0);

    constructor(address _rocketStorageAddress) public {
        rocketStorage = RocketStorageInterface(_rocketStorageAddress);
    }

}
```

The above constructor should be called with the address of the `RocketStorage` contract on the appropriate network.

Because of Rocket Pool's architecture, the addresses of other contracts should not be used directly, but retrieved from the blockchain before use. Network upgrades may have occurred since the previous interaction, resulting in outdated addresses.

Other contract instances can be created using the appropriate interface taken from the Rocket Pool repository, e.g.:

```
import "RocketStorageInterface.sol";
import "RocketPoolInterface.sol";

contract Example {

    RocketStorageInterface rocketStorage = RocketStorageInterface(0);

    constructor(address _rocketStorageAddress) public {
        rocketStorage = RocketStorageInterface(_rocketStorageAddress);
    }

    exampleMethod() public {
        address rocketPoolAddress = rocketStorage.getAddress(keccak256(abi.
↪encodePacked("contract.name", "rocketPool")));
        RocketPoolInterface rocketPool = RocketPoolInterface(rocketPoolAddress);
        ...
    }

}
```

The names of the Rocket Pool contracts, as defined in `RocketStorage`, are:

- `rocketAdmin`
- `rocketPool`
- `rocketRole`
- `rocketNode`

- rocketPIP
- rocketUpgrade
- rocketUpgradeApproval
- rocketDepositAPI
- rocketGroupAPI
- rocketNodeAPI
- rocketDeposit
- rocketDepositIndex
- rocketDepositQueue
- rocketDepositVault
- rocketGroupAccessorFactory
- rocketNodeFactory
- rocketNodeKeys
- rocketNodeTasks
- rocketNodeWatchtower
- rocketMinipoolDelegateNode
- rocketMinipoolDelegateStatus
- rocketMinipoolDelegateDeposit
- rocketMinipoolFactory
- rocketMinipoolSet
- rocketMinipoolSettings
- rocketDepositSettings
- rocketGroupSettings
- rocketNodeSettings
- rocketPoolToken
- rocketETHToken
- taskDisableInactiveNodes
- taskCalculateNodeFee
- utilMaths
- utilPublisher
- utilAddressQueueStorage
- utilBytes32QueueStorage
- utilAddressSetStorage
- utilBytes32SetStorage
- utilStringSetStorage

Many of these are used for internal processing, and only a few are likely to be useful for extension, specifically the Group & Deposit API contracts (`rocketGroupAPI`, `rocketGroupContract`, and `rocketDepositAPI`). The following sections cover the various API methods available; for information on methods of other contracts, consult their interfaces in the Rocket Pool repository.

Groups & Accessors

Overview

All user deposits into the Rocket Pool network are made through a Group. An organisation can register as a Group with Rocket Pool, allowing them to move funds into the network for staking. This effectively allows them to use Rocket Pool as staking infrastructure, using their own deposit processes and application front-ends. Rocket Pool employs an "eat your own dog food" approach, using its own registered Group for processing user interactions.

After registering with Rocket Pool, a Group must create and deploy one or more Depositor and Withdrawer contracts to handle user deposits and withdrawals. These contracts must be registered with the Group, and then user deposits and withdrawals for the Group can be made via them. If no custom deposit or withdrawal logic is required, Rocket Pool provides a "default" Group Accessor contract factory, which provides simple functionality for both. This contract simply acts as a proxy, forwarding user deposits into the network, and refunds and withdrawals back into user accounts.

Registration

Registering a Group is performed via the `RocketGroupAPI.add` method, accepting the following parameters:

- `name` (*string*): A unique name for the Group (must not already be in use)
- `stakingFee` (*uint256*): A percentage of rewards to charge the Group's users, given as a fraction of 1 ether, in wei (e.g. 50000000000000000 = 5%)

This method also requires a transaction value of 0.05 ETH; this amount is charged to discourage excessive Group registrations. Registering a Group creates a new `RocketGroupContract` instance, registers it with the network, and emits a `GroupAdd` event with an `ID` property corresponding to its address. The *owner* of the Group is considered to be the address which registered it.

Accessor Creation

A "default" Group Accessor contract can be created via the `RocketGroupAPI.createDefaultAccessor` method, accepting a single parameter:

- `ID` (*address*): The ID of the Group (its `RocketGroupContract` instance address)

This emits a `GroupCreateDefaultAccessor` event with an `accessorAddress` property corresponding to the created contract's address.

Alternatively, custom Depositor and Withdrawer contracts may be created and deployed to the network.

All custom Depositor contracts *must* implement the RocketGroupAccessorContractInterface. Specifically, they must have an external, payable `rocketpoolEtherDeposit` method which returns `true` to indicate success. This allows them to receive ether being sent back from refunded deposits.

There are no other strict requirements for Depositor and Withdrawer contracts, but they should provide the following functionality:

Depositor:

- A payable "deposit" method to accept payments and transfer them to Rocket Pool

- A "refund queued deposit" method to handle refunds of deposits which are still queued
- A "refund stalled deposit" method to handle refunds of deposits assigned to stalled minipools

Withdrawer:

- A "withdraw" method to handle withdrawals of funds from minipools which have completed staking
- A "withdraw during staking" method to handle (penalised) withdrawals of funds from staking minipools
- A "set backup withdrawal address" method to set a backup withdrawal address for a deposit

These methods should all interact with the `RocketDepositAPI` contract; refer to *its documentation* or to the "default" Group Accessor contract for implementation examples.

Note: the `RocketDepositAPI` contract address should *not* be hard-coded in custom Group Accessor contracts, but retrieved from `RocketStorage` dynamically.

Accessor Registration

Once Group Accessor contracts have been created, they can be registered with the Group via the `RocketGroupContract.addDepositor` and `RocketGroupContract.addWithdrawer` methods. These methods are both restricted to the owner of the Group contract, and accept a single parameter:

- `address` (*address*): The address of the Accessor contract to register with the Group

Deposits

Overview

User deposits into the Rocket Pool network are made via a *Group Accessor* contract, which interacts with the `RocketDepositAPI`. Transactions cannot be sent to the `RocketDepositAPI` contract directly; they are only valid if they originate from a Group Depositor or Withdrawer. The following information is provided for the development of custom Group Accessor contracts which will interact with it.

Note: the `RocketDepositAPI` contract address should *not* be hard-coded in custom Group Accessor contracts, but retrieved from `RocketStorage` dynamically.

Making Deposits

Deposits can be made by calling the `RocketDepositAPI.deposit` method with the value of the deposit, accepting the following parameters:

- `groupID` (*address*): The ID of the Group to deposit under (its `RocketGroupContract` instance address)
- `userID` (*address*): The ID of the user to deposit under (e.g. the address which is making the deposit via the Group)
- `durationID` (*string*): The ID of the duration to stake the deposit for

The deposit method should be called with a value as follows:

```
address rocketDepositAPIAddress = rocketStorage.getAddress(keccak256(abi.encodePacked(
↪"contract.name", "rocketDepositAPI")));
rocketDepositAPI = RocketDepositAPIInterface(rocketDepositAPIAddress);
bool success = rocketDepositAPI.deposit.value(value)(groupID, userID, durationID);
```

This method returns a boolean flag to indicate success, and emits a `Deposit` event with a 32-byte `depositID` property corresponding to the ID of the new deposit.

Refunding Deposits

Queued deposits (or portions of deposits still remaining in the queue) can be refunded with the `RocketDepositAPI.depositRefundQueued` method, accepting the following parameters:

- `groupID` (*address*): The ID of the Group the deposit was made under
- `userID` (*address*): The ID of the user the deposit was made by
- `durationID` (*string*): The ID of the duration the deposit was staked for
- `depositID` (*bytes32*): The ID of the deposit

Deposits in stalled minipools can be refunded with the `RocketDepositAPI.depositRefundMinipoolStalled` method, accepting the following parameters:

- `groupID` (*address*): The ID of the Group the deposit was made under
- `userID` (*address*): The ID of the user the deposit was made by
- `depositID` (*bytes32*): The ID of the deposit
- `minipool` (*address*): The address of the stalled minipool

Both methods transfer the refunded ether to the Depositor contract the transaction was sent to, which is responsible for transferring it to the user. They return the amount which was refunded, and emit a `DepositRefund` event with the deposit's details.

Withdrawing Deposits

Deposits (plus rewards) can be withdrawn from minipools which have finished staking with the `RocketDepositAPI.depositWithdrawMinipool` method, accepting the following parameters:

- `groupID` (*address*): The ID of the Group the deposit was made under
- `userID` (*address*): The ID of the user the deposit was made by
- `depositID` (*bytes32*): The ID of the deposit
- `minipool` (*address*): The address of the minipool

Deposits can be withdrawn early from staking minipools (forfeiting rewards and with a penalty) with the `RocketDepositAPI.depositWithdrawMinipoolStaking` method, accepting the following parameters:

- `groupID` (*address*): The ID of the Group the deposit was made under
- `userID` (*address*): The ID of the user the deposit was made by
- `depositID` (*bytes32*): The ID of the deposit
- `minipool` (*address*): The address of the minipool
- `amount` (*uint256*): The amount of the deposit to withdraw from the minipool in wei

Both methods transfer the withdrawn rETH to the Withdrawer contract the transaction was sent to, which is responsible for transferring it to the user. They return the amount which was withdrawn, and emit a `DepositWithdraw` event with the deposit's details.

Backup Withdrawal Addresses

Backup withdrawal addresses can be set for deposits, allowing users to withdraw from an alternate address after staking in the case of lost keys for their primary account. This is performed via the `RocketDepositAPI.setDepositBackupWithdrawalAddress` method, accepting the following parameters:

- `groupID` (*address*): The ID of the Group the deposit was made under
- `userID` (*address*): The ID of the user the deposit was made by
- `depositID` (*bytes32*): The ID of the deposit
- `backup` (*address*): The backup withdrawal address to set for the deposit

This method returns a boolean flag to indicate success, and emits a `DepositSetBackupAddress` event with the deposit's updated details. After assigning a backup withdrawal address, users should be able to withdraw their deposit by sending a transaction from it to the Withdrawer contract's withdrawal method.

API Reference

RocketGroupAPI

Accessors:

- `getGroupName(address ID)`: Get the registered name of the group with the specified ID

Mutators:

- `add(string name, uint256 stakingFee)`: Register a group with Rocket Pool with the specified name and staking fee (the percentage of rewards to charge the group's users, as a fraction of 1 ether, in wei)
- `createDefaultAccessor(address ID)`: Create a default group accessor contract for the group with the specified ID (must be registered with `RocketGroupContract.addDepositor` & `RocketGroupContract.addWithdrawer`)

RocketGroupContract

Accessors:

- `getOwner()`: Get the address of the group's owner
- `getFeePerc()`: Get the percentage of rewards to charge the group's users, as a fraction of 1 ether, in wei
- `getFeePercRocketPool()`: Get the percentage of rewards charged to the group's users by Rocket Pool, as a fraction of 1 ether, in wei
- `getFeeAddress()`: Get the address to send the group's fees to (default: the account which registered the group)
- `hasDepositor(address value)`: Returns true if the group has a registered depositor contract with the specified address
- `hasWithdrawer(address value)`: Returns true if the group has a registered withdrawer contract with the specified address

Mutators (restricted to group owner):

- `setFeePerc(uint256 value)`: Update the percentage of rewards to charge the group's users, as a fraction of 1 ether, in wei (does not affect any currently assigned deposits)

- `setFeeAddress(address value)`: Update the address to send the group's fees to (default: the account which registered the group)
- `addDepositor(address value)`: Add a depositor contract to the group; emits `DepositorAdd`
- `removeDepositor(address value)`: Remove a depositor contract from the group; emits `DepositorRemove`
- `addWithdrawer(address value)`: Add a withdrawer contract to the group; emits `WithdrawerAdd`
- `removeWithdrawer(address value)`: Remove a withdrawer contract from the group; emits `WithdrawerRemove`

RocketDepositAPI

Mutators (restricted to group depositors):

- `deposit(address groupID, address userID, string durationID)`: Deposit into Rocket Pool for the specified staking duration; emits `Deposit`
- `depositRefundQueued(address groupID, address userID, string durationID, bytes32 depositID)`: Refund a queued deposit or portion of a deposit; emits `DepositRefund`
- `depositRefundMinipoolStalled(address groupID, address userID, bytes32 depositID, address minipool)`: Refund a deposit from a stalled minipool; emits `DepositRefund`

Mutators (restricted to group withdrawers):

- `depositWithdrawMinipoolStaking(address groupID, address userID, bytes32 depositID, address minipool, uint256 amount)`: Withdraw early from a staking minipool; emits `DepositWithdraw`
- `depositWithdrawMinipool(address groupID, address userID, bytes32 depositID, address minipool)`: Withdraw from a minipool which has finished staking; emits `DepositWithdraw`
- `setDepositBackupWithdrawalAddress(address groupID, address userID, bytes32 depositID, address backup)`: Set a backup withdrawal address for a deposit; emits `DepositSetBackupAddress`

1.3 The Rocket Pool JavaScript Library

1.3.1 Introduction

The Rocket Pool JavaScript library is the primary means of interacting with the Rocket Pool network for users and applications. It is used by applications such as the Rocket Pool website to retrieve information about the network and facilitate user interaction such as deposits and withdrawals. It provides wrapper methods which load the Rocket Pool contracts with web3.js and abstractions to use many of their features easily.

This guide assumes familiarity with JavaScript and provides all code samples as JS.

1.3.2 Contents

Getting Started

Installation

The Rocket Pool JavaScript library can be added to your application via NPM, and requires web3.js:

```
npm install github:rocket-pool/rocketpool-js
npm install web3
```

Initialisation

The library must be initialised with a web3 instance and a Truffle `RocketStorage` contract artifact:

```
import Web3 from 'web3';
import RocketPool from 'rocketpool';
import RocketStorage from './contracts/RocketStorage.json';

const web3 = new Web3('http://localhost:8545');

const rp = new RocketPool(web3, RocketStorage);
```

Usage

The Rocket Pool library is divided into several modules, each for interacting with a different aspect of the network:

- `contracts`: Handles dynamic loading of the Rocket Pool contracts
- `deposit`: Manages user deposits
- `group`: Manages groups registered with Rocket Pool
- `node`: Manages the nodes making up the Rocket Pool network
- `pool`: Manages the main minipool registry and individual minipools
- `settings.deposit`: Provides information on user deposit settings
- `settings.group`: Provides information on group settings
- `settings.minipool`: Provides information on minipool settings
- `settings.node`: Provides information on smart node settings
- `tokens.reth`: Manages rETH token interactions
- `tokens.rpl`: Manages RPL token interactions

All methods typically return promises due to the asynchronous nature of working with the Ethereum network. Getters return promises which resolve to their value, while mutators (methods which send transactions) return promises which resolve to a transaction receipt. Mutators also accept a transaction options object, and an `onConfirmation` callback handler to handle specific confirmation numbers on transactions.

When using the Rocket Pool library in your project, you may handle the promises returned in the traditional way, or use async/await syntax if supported, e.g.:

```
rp.contracts.get('rocketPool')
    .then(rocketPool => rocketPool.methods.getPoolsCount().call())
    .then(poolsCount => { console.log(poolsCount); });
```

or:

```
let rocketPool = await rp.contracts.get('rocketPool');
let poolsCount = await rocketPool.methods.getPoolsCount().call();
console.log(poolsCount);
```

Contracts

Overview

The `contracts` module loads Rocket Pool contract ABIs and addresses from `RocketStorage`, where all network contracts are registered. Contract ABIs and addresses are loaded from the chain, the ABIs are decompressed and decoded, and then web3 contract instances are created from them. This is performed dynamically because Rocket Pool contracts can be upgraded and their ABIs and addresses may change.

This module is used by other library modules internally, and generally does not need to be used directly. However, it is exposed publicly for when direct access to web3 contract instances is desired or the library wrapper methods are insufficient.

Loading Contracts & ABIs

Network contracts can be loaded via the `contracts.get()` method, which accepts either a single contract name as a string, or a list of contract names as an array of strings. If a single contract name is passed, this method returns a promise resolving to a web3 contract instance. If a list of contract names is passed, it returns a promise resolving to an array of web3 contract instances, in the same order.

Contract ABIs can be loaded in a similar fashion via the `contracts.abi()` method, which accepts either a single contract name, or a list of names, to retrieve ABIs for. This returns a promise resolving to an ABI as a JavaScript object, or an array of ABIs.

Creating Contract Instances

Some network contracts, such as `RocketGroupContract`, `RocketNodeContract` and `RocketMinipool` have multiple instances deployed at a number of different addresses. To create an instance of one of these contracts, use the `contracts.make(name, address)` method. It accepts the name of the contract and the address of the specific instance required, both as strings, and returns a promise resolving to a web3 contract instance.

Alternate Contract Versions

When Rocket Pool network contracts are upgraded, old versions remain on the chain and can still be accessed if required. A "contract version set", consisting of all versions of a contract by name, can be loaded with the `contracts.versions(name)` method. This method accepts the name of the contract to load, and returns a promise resolving to the version set object.

Contract version sets are primarily used for accessing old event data. They provide the following methods:

- `versionSet.current()`: Returns the current version of the contract
- `versionSet.first()`: Returns the first version of the contract deployed
- `versionSet.at(index)`: Returns the version of the contract at the specified version index (0 = first version)
- `versionSet.getPastEvents(eventName, options)`: As per web3's contract.getPastEvents, but returns a promise resolving to the events for all versions of the contract

Deposits

Overview

The `deposit` module loads user deposit data from the chain. This includes user deposits made via any group (Rocket Pool or other third party groups). It does not include node deposits (made by node operators to create minipools).

Data Types

`DepositDetails` objects define the various details of a user deposit:

```
DepositDetails {
    id               // The 32-byte deposit ID as a string
    totalAmount      // The total amount of ether deposited (in wei)
    queuedAmount     // The amount of the deposit still remaining in the queue (in wei)
    stakingAmount    // The amount of the deposit assigned to minipools for staking␣
↪(in wei)
    refundedAmount   // The amount of the deposit which has been refunded (in wei)
    withdrawnAmount  // The amount of the deposit which has been withdrawn (in wei)
    pools            // An array of DepositPoolDetails objects
    backupAddress    // The backup withdrawal address set for the deposit, or null
}
```

`DepositPoolDetails` objects define the details for a minipool which a deposit is (fully or partially) assigned to:

```
DepositPoolDetails {
    address          // The address of the minipool assigned to
    stakingAmount    // The amount of the deposit which is assigned to this minipool
}
```

Methods

- `deposit.getDeposits(groupId, userId, durationId)`: Get all deposits made by the specified user, via the specified group (addresses), for the specified staking duration ID (string); returns `Promise<DepositDetails[]>`
- `deposit.getQueuedDeposits(groupId, userId, durationId)`: As above, but only returns deposits which are still at least partially queued
- `deposit.getDeposit(depositId)`: Get the details of the deposit with the specified ID (string); returns `Promise<DepositDetails>`
- `deposit.getDepositStakingPools(depositId)`: Get the details of the minipools the deposit with the specified ID (string) is assigned to; returns `Promise<DepositPoolDetails[]>`
- `deposit.getDepositCount(groupId, userId, durationId)`: Get the number of deposits made by the specified user, via the specified group (addresses), for the specified staking duration ID (string); returns `Promise<number>`
- `deposit.getDepositAt(groupId, userId, durationId, index)`: Get the ID of the deposit made by the specified user, via the specified group (addresses), for the specified staking duration ID (string), at the specified index (number); returns `Promise<string>`
- `deposit.getQueuedDepositCount(groupId, userId, durationId)`: As above, but only returns the number of deposits which are still at least partially queued

- `deposit.getQueuedDepositAt(groupId, userId, durationId, index)`: As above, but only returns the ID if a deposit which is still at least partially queued
- `deposit.getDepositTotalAmount(depositId)`: Get the total amount of a deposit by ID (string) in wei; returns `Promise<string>`
- `deposit.getDepositQueuedAmount(depositId)`: Get the amount of a deposit by ID (string) still remaining in the queue, in wei; returns `Promise<string>`
- `deposit.getDepositStakingAmount(depositId)`: Get the amount of a deposit by ID (string) assigned to minipools for staking, in wei; returns `Promise<string>`
- `deposit.getDepositRefundedAmount(depositId)`: Get the amount of a deposit by ID (string) which has been refunded, in wei; returns `Promise<string>`
- `deposit.getDepositWithdrawnAmount(depositId)`: Get the amount of a deposit by ID (string) which has been withdrawn, in wei; returns `Promise<string>`
- `deposit.getDepositStakingPoolCount(depositId)`: Get the number of minipools a deposit is staking under by ID (string); returns `Promise<number>`
- `deposit.getDepositStakingPoolAt(depositId, index)`: Get the address of a minipool with the specified index (number) that the deposit with the specified ID (string) is staking under; returns `Promise<string>`
- `deposit.getDepositStakingPoolAmount(depositId, minipoolAddress)`: Get the amount of a deposit by ID (string) that has been assigned to the minipool with the specified address (string) for staking, in wei; returns `Promise<string>`
- `deposit.getDepositBackupAddress(depositId)`: Get the backup withdrawal address of a deposit by ID (string); returns `Promise<string | null>`

Groups

Overview

The `group` module manages Rocket Pool groups. It loads group data from the chain, and can be used to register new groups and create default group accessors. It also provides group contract functionality (for group owners to manage their group), and group accessor contract functionality (for group users to manage their deposits).

Data Types

`GroupDetails` objects define the various details of a group:

```
GroupDetails {
    owner           // The owner address of the group
    groupFee        // The fee charged to the group's users, as a fraction of their␣
↪rewards
    rocketPoolFee   // The fee charged to the groups's users by Rocket Pool, as a␣
↪fraction of their rewards
    groupFeeAddress // The address which group fees are paid to
}
```

`GroupContract` objects wrap a web3 contract instance and provide methods for managing a group. Mutator methods are restricted to the group's owner.

`GroupAccessorContract` objects wrap a web3 contract instance and provide methods for managing user deposits through the group. Deposit refund and withdrawal methods are restricted to the user who made the deposit.

Methods

Group Module:

- `group.getName(groupId)`: Get the name of the group with the specified ID (contract address); returns `Promise<string>`
- `group.getContract(address)`: Get a contract instance for the group at the specified address; returns `Promise<GroupContract>`
- `group.getAccessorContract(address)`: Get a contract instance for the group accessor at the specified address; returns `Promise<GroupAccessorContract>`
- `group.add(name, stakingFeeFraction, options, onConfirmation)`: Register a group with Rocket Pool with the specified name (string) and fraction of rewards to charge users (number); returns `Promise<TransactionReceipt>`
- `group.createDefaultAccessor(groupId, options, onConfirmation)`: Create a default group accessor contract for the group with the specified ID (contract address); returns `Promise<TransactionReceipt>`

GroupContract:

- `GroupContract.getDetails()`: Get the group's details; returns `Promise<GroupDetails>`
- `GroupContract.getOwner()`: Get the group's owner address; returns `Promise<string>`
- `GroupContract.getGroupFee()`: Get the fee charged to the group's users, as a fraction of their rewards; returns `Promise<number>`
- `GroupContract.getRocketPoolFee()`: Get the fee charged to the group's users by Rocket Pool, as a fraction of their rewards; returns `Promise<number>`
- `GroupContract.getGroupFeeAddress()`: Get the address which group fees are paid to; returns `Promise<string>`
- `GroupContract.setGroupFee(feeFraction, options, onConfirmation)`: Set the fee charged to the group's users (number), as a fraction of their rewards; returns `Promise<TransactionReceipt>`
- `GroupContract.setGroupFeeAddress(address, options, onConfirmation)`: Set the address which group fees are paid to; returns `Promise<TransactionReceipt>`
- `GroupContract.addDepositor(address, options, onConfirmation)`: Register a depositor contract at the specified address with the group; returns `Promise<TransactionReceipt>`
- `GroupContract.removeDepositor(address, options, onConfirmation)`: Remove the depositor contract at the specified address from the group; returns `Promise<TransactionReceipt>`
- `GroupContract.addWithdrawer(address, options, onConfirmation)`: Register a withdrawer contract at the specified address with the group; returns `Promise<TransactionReceipt>`
- `GroupContract.removeWithdrawer(address, options, onConfirmation)`: Remove the withdrawer contract at the specified address from the group; returns `Promise<TransactionReceipt>`

GroupAccessorContract:

- `GroupAccessorContract.deposit(durationId, options, onConfirmation)`: Deposit an amount of ether into Rocket Pool for the specified staking duration (string); returns `Promise<TransactionReceipt>`
- `GroupAccessorContract.refundQueuedDeposit(durationId, depositId, options, onConfirmation)`: Refund the portion of a deposit by ID (string) staking for the specified staking duration (string) which is still in the deposit queue; returns `Promise<TransactionReceipt>`

- `GroupAccessorContract.refundStalledMinipoolDeposit (depositId, minipoolAddress, options, onConfirmation)`: Refund the portion of a deposit by ID (string) assigned to the stalled minipool at minipoolAddress (string); returns `Promise<TransactionReceipt>`
- `GroupAccessorContract.withdrawStakingMinipoolDeposit (depositId, minipoolAddress, weiAmount, options, onConfirmation)`: Withdraw weiAmount (string) wei of a deposit by ID (string) from the staking minipool at minipoolAddress (string); returns `Promise<TransactionReceipt>`
- `GroupAccessorContract.withdrawMinipoolDeposit (depositId, minipoolAddress, options, onConfirmation)`: Withdraw the portion of a deposit by ID (string) assigned to the minipool at minipoolAddress (string) which has finished staking; returns `Promise<TransactionReceipt>`
- `GroupAccessorContract.setDepositBackupAddress (depositId, backupAddress, options, onConfirmation)`: Set a backup withdrawal address for a deposit by ID (string); returns `Promise<TransactionReceipt>`

Nodes

Overview

The `node` module manages nodes in the Rocket Pool network. It loads node data from the chain, and can be used to register new nodes. It also provides node contract functionality (for node owners to manage their node).

Data Types

`NodeDetails` objects define the various details of a node:

```
NodeDetails {
    owner                   // The owner address of the node
    rewardsAddress          // The address which node fees and rewards are paid to
    ethBalance              // The current ETH balance of the node contract, in wei
    rplBalance              // The current RPL balance of the node contract, in wei
    hasDepositReservation   // Whether the node has a current deposit reservation
}
```

`NodeDepositReservation` objects define deposit reservations made by the node:

```
NodeDepositReservation {
    created                 // The time at which the deposit reservation was made
    etherRequired           // The ether required to complete the deposit, in wei
    rplRequired             // The RPL required to complete the deposit, in wei
    durationId              // The ID of the staking duration that the node will␣
↪stake for
    validatorPubkey         // The validator public key which was submitted with the␣
↪deposit reservation
    validatorSignature      // The validator signature which was submitted with the␣
↪deposit reservation
}
```

`NodeContract` objects wrap a web3 contract instance and provide methods for managing a node. Mutator methods are restricted to the node's owner.

Methods

Node Module:

- `node.getAvailableCount(stakingDurationId)`: Get the number of nodes with minipools available for assignment staking for the specified duration (string); returns `Promise<number>`
- `node.getRPLRatio(stakingDurationId)`: Get the currentl RPL ratio by staking duration ID (string); returns `Promise<number>`
- `node.getRPLRequired(weiAmount, stakingDurationId)`: Get the current RPL requirement in wei for the specified ether amount in wei (string), staking for the specified duration (string); returns `Promise<[string, number]>`
- `node.getTrusted(nodeOwner)`: Get a flag indicating whether the node with the specified owner (address) is trusted; returns `Promise<bool>`
- `node.getTimezoneLocation(nodeOwner)`: Get the timezone location of the node with the specified owner (address); returns `Promise<string>`
- `node.getContractAddress(nodeOwner)`: Get address of the node contract with the specified owner (address); returns `Promise<string>`
- `node.getContract(address)`: Get a contract instance for the node at the specified address; returns `Promise<NodeContract>`
- `node.add(timezone, options, onConfirmation)`: Register a node with Rocket Pool with the specified timezone location (string); returns `Promise<TransactionReceipt>`
- `node.setTimezoneLocation(timezone, options, onConfirmation)`: Set the timezone location (string) for a node; returns `Promise<TransactionReceipt>`

NodeContract:

- `NodeContract.getDetails()`: Get the node's details; returns `Promise<NodeDetails>`
- `NodeContract.getDepositReservation()`: Get the node's current deposit reservation details; returns `Promise<NodeDepositReservation>`
- `NodeContract.getOwner()`: Get the owner address of the node; returns `Promise<string>`
- `NodeContract.getRewardsAddress()`: Get the address which node fees and rewards are paid to; returns `Promise<string>`
- `NodeContract.getEthBalance()`: Get the current ETH balance of the node contract in wei; returns `Promise<string>`
- `NodeContract.getRplBalance()`: Get the current RPL balance of the node contract in wei; returns `Promise<string>`
- `NodeContract.getHasDepositReservation()`: Get a flag indicating whether the node has a current deposit reservation; returns `Promise<boolean>`
- `NodeContract.getDepositReservationCreated()`: Get the time at which the node's deposit reservation was made; returns `Promise<Date>`
- `NodeContract.getDepositReservationEthRequired()`: Get the ether required to complete the node's current deposit reservation, in wei; returns `Promise<string>`
- `NodeContract.getDepositReservationRplRequired()`: Get the RPL required to complete the node's current deposit reservation, in wei; returns `Promise<string>`
- `NodeContract.getDepositReservationDurationId()`: Get the ID of the staking duration that the node's current deposit reservation will stake for; returns `Promise<string>`

- `NodeContract.getDepositReservationValidatorPubkey()`: Get the validator public key which was submitted with the node's current deposit reservation; returns `Promise<string>`
- `NodeContract.getDepositReservationValidatorSignature()`: Get the validator signature which was submitted with the node's current deposit reservation; returns `Promise<string>`
- `NodeContract.setRewardsAddress(address, options, onConfirmation)`: Set the address which node fees and rewards are paid to; returns `Promise<TransactionReceipt>`
- `NodeContract.reserveDeposit (durationId, validatorPubkey, validatorSignature, options, onConfirmation)`: Reserve a deposit for the specified staking duration (string), with the specified validator public key and signature (strings); returns `Promise<TransactionReceipt>`
- `NodeContract.cancelDepositReservation(options, onConfirmation)`: Cancel the node's current deposit reservation; returns `Promise<TransactionReceipt>`
- `NodeContract.completeDeposit(options, onConfirmation)`: Complete the node's current deposit reservation; returns `Promise<TransactionReceipt>`
- `NodeContract.withdrawMinipoolDeposit(minipoolAddress, options, onConfirmation)`: Withdraw the ETH / rETH & RPL deposit from the specified minipool which has timed out or finished staking; returns `Promise<TransactionReceipt>`
- `NodeContract.withdrawEth(weiAmount, options, onConfirmation)`: Withdraw the specified amount of ether, in wei (string) from the node contract to the node account; returns `Promise<TransactionReceipt>`
- `NodeContract.withdrawRpl(weiAmount, options, onConfirmation)`: Withdraw the specified amount of RPL, in wei (string) from the node contract to the node account; returns `Promise<TransactionReceipt>`

Pools

Overview

The `pool` module loads general minipool data from the chain. It also provides minipool contract functionality (which loads individual minipool data).

Data Types

`NodeDetails` objects define the details of the node a minipool is owned by:

```
NodeDetails {
    owner           // The owner address of the node the minipool is owned by
    contract        // The node contract address of the node the minipool is owned by
    depositEth      // The amount of ether deposited by the node owner in wei
    depositRpl      // The amount of RPL deposited by the node owner in wei
    trusted         // The trusted status of the node when the minipool was launched
    depositExists   // Whether the node deposit is still in the minipool
    balance         // The node owner's current deposited ether balance in wei
    userFee         // The fraction of rewards charged to users by the node operator
}
```

`DepositDetails` objects define the details of a specific deposit assigned to a minipool:

```
DepositDetails {
    exists                  // Whether the deposit exists in the minipool
    userId                  // The address of the user who made the deposit
    groupId                 // The ID of the group the deposit was made via
    balance                 // The current balance of the deposit in the minipool, in␣
↪wei
    stakingTokensWithdrawn  // The amount of the deposit withdrawn as rETH while␣
↪staking, in wei
    rocketPoolFee           // The fraction of rewards charged by Rocket Pool for␣
↪this deposit
    groupFee                // The fraction of rewards charged by the group for this␣
↪deposit
    created                 // The time that the deposit was assigned to the minipool
}
```

`StatusDetails` objects define a minipool's status:

```
StatusDetails {
    status                          // The current status code of the minipool
    statusChangedTime               // The time the minipool's status was last changed
    statusChangedBlock              // The block the minipool's status was last␣
↪changed at
    stakingDurationId               // The ID of the staking duration the minipool is␣
↪staking for
    stakingDuration                 // The duration in blocks that the minipool is␣
↪staking for
    validatorPubkey                 // The validator public key submitted by the node␣
↪operator
    validatorSignature              // The validator signature submitted by the node␣
↪operator
    userDepositCapacity             // The total capacity for user deposits in wei
    userDepositTotal                // The total amount of assigned user deposits in␣
↪wei
    stakingUserDepositsWithdrawn    // The total amount of deposits withdrawn as rETH␣
↪while staking, in wei
}
```

`MinipoolContract` objects wrap a web3 contract instance and provide methods for retrieving a minipool's information.

Methods

Pool Module:

- `pool.getPoolExists(address)`: Get whether or not a minipool with a given address exists; returns `Promise<boolean>`
- `pool.getPoolCount()`: Get the total number of minipools; returns `Promise<number>`
- `pool.getPoolAt(index)`: Get a minipool address by index (number); returns `Promise<string>`
- `pool.getTotalEthAssigned(stakingDurationId)`: Get the total network ether assigned for the specified staking duration (string), in wei; returns `Promise<string>`
- `pool.getTotalEthCapacity(stakingDurationId)`: Get the total network ether capacity for the specified staking duration (string), in wei; returns `Promise<string>`

- `pool.getNetworkUtilisation(stakingDurationId)`: Get the current network utilisation for the specified staking duration (string) as a fraction; returns `Promise<number>`
- `pool.getMinipoolContract(address)`: Get a contract instance for the minipool at the specified address; returns `Promise<MinipoolContract>`

MinipoolContract:

- `MinipoolContract.getNodeDetails()`: Get the details of the node the minipool is owned by; returns `Promise<NodeDetails>`
- `MinipoolContract.getNodeOwner()`: Get the owner address of the node the minipool is owned by; returns `Promise<string>`
- `MinipoolContract.getNodeContract()`: Get the contract address of the node the minipool is owned by; returns `Promise<string>`
- `MinipoolContract.getNodeDepositEth()`: Get the amount of ether deposited by the node owner in wei; returns `Promise<string>`
- `MinipoolContract.getNodeDepositRpl()`: Get the amount of RPL deposited by the node owner in wei; returns `Promise<string>`
- `MinipoolContract.getNodeTrusted()`: Get the trusted status of the node when the minipool was launched; returns `Promise<boolean>`
- `MinipoolContract.getNodeDepositExists()`: Get whether the node deposit is still in the minipool; returns `Promise<boolean>`
- `MinipoolContract.getNodeBalance()`: Get the node owner's current deposited ether balance in wei; returns `Promise<string>`
- `MinipoolContract.getNodeUserFee()`: Get the fraction of rewards charged to users by the node operator; returns `Promise<number>`
- `MinipoolContract.getDepositCount()`: Get the total number of deposits assigned to the minipool; returns `Promise<number>`
- `MinipoolContract.getDepositDetails(depositId)`: Get the details of a deposit assigned to the minipool by ID (string); returns `Promise<DepositDetails>`
- `MinipoolContract.getDepositExists(depositId)`: Get whether the deposit with the specified ID (string) exists in the minipool; returns `Promise<boolean>`
- `MinipoolContract.getDepositUserID(depositId)`: Get the user ID of a deposit assigned to the minipool by ID (string); returns `Promise<string>`
- `MinipoolContract.getDepositGroupID(depositId)`: Get the group ID of a deposit assigned to the minipool by ID (string); returns `Promise<string>`
- `MinipoolContract.getDepositBalance(depositId)`: Get the current balance of a deposit assigned to the minipool by ID (string), in wei; returns `Promise<string>`
- `MinipoolContract.getDepositStakingTokensWithdrawn(depositId)`: Get the amount of a deposit by ID (string) withdrawn as rETH while staking, in wei; returns `Promise<string>`
- `MinipoolContract.getDepositRocketPoolFee(depositId)`: Get the fraction of rewards charged by Rocket Pool for a deposit by ID (string); returns `Promise<number>`
- `MinipoolContract.getDepositGroupFee(depositId)`: Get the fraction of rewards charged by the group for a deposit by ID (string); returns `Promise<number>`
- `MinipoolContract.getDepositCreated(depositId)`: Get the time at which a deposit by ID (string) was assigned to the minipool; returns `Promise<Date>`

- `MinipoolContract.getStatusDetails()`: Get the details of the minipool's current status; returns `Promise<StatusDetails>`
- `MinipoolContract.getStatus()`: Get the minipool's current status code; returns `Promise<number>`
- `MinipoolContract.getStatusChangedTime()`: Get the time at which the minipool's status was last changed; returns `Promise<Date>`
- `MinipoolContract.getStatusChangedBlock()`: Get the block that the minipool's status was last changed at; returns `Promise<number>`
- `MinipoolContract.getStakingDurationId()`: Get the ID of the staking duration the minipool is staking for; returns `Promise<string>`
- `MinipoolContract.getStakingDuration()`: Get the duration in blocks that the minipool is staking for; returns `Promise<number>`
- `MinipoolContract.getValidatorPubkey()`: Get the validator public key submitted by the node operator; returns `Promise<string>`
- `MinipoolContract.getValidatorSignature()`: Get the validator signature submitted by the node operator; returns `Promise<string>`
- `MinipoolContract.getUserDepositCapacity()`: Get the minipool's total capacity for user deposits in wei; returns `Promise<string>`
- `MinipoolContract.getUserDepositTotal()`: Get the total amount of user deposits assigned to the minipool in wei; returns `Promise<string>`
- `MinipoolContract.getStakingUserDepositsWithdrawn()`: Get the total amount of deposits withdrawn as rETH from the minipool while staking, in wei; returns `Promise<string>`
- `MinipoolContract.getStakingBalanceStart()`: Get the total amount of ether which was deposited to the beacon chain for staking, in wei; returns `Promise<string>`
- `MinipoolContract.getStakingBalanceEnd()`: Get the total amount of ether returned from the beacon chain after staking, in wei; returns `Promise<string>`

Settings

Overview

The `settings` module loads Rocket Pool network settings data from the chain, and is divided into 4 submodules:

- `settings.deposit`: Loads information on user deposit settings
- `settings.group`: Loads information on group settings
- `settings.minipool`: Loads information on minipool settings
- `settings.node`: Loads information on smart node settings

Methods

Deposit Settings:

- `settings.deposit.getDepositAllowed()` Get whether user deposits are currently enabled; returns `Promise<boolean>`

- `settings.deposit.getDepositChunkSize()` Get the 'chunk' size to divide deposits into in wei; returns `Promise<string>`
- `settings.deposit.getDepositMin()` Get the minimum deposit amount in wei; returns `Promise<string>`
- `settings.deposit.getDepositMax()` Get the maximum deposit amount in wei; returns `Promise<string>`
- `settings.deposit.getChunkAssignMax()` Get the maximum number of chunks that are assigned per deposit transaction; returns `Promise<number>`
- `settings.deposit.getDepositQueueSizeMax()` Get the maximum size of the deposit queue before user deposits are restricted or disabled, in wei; returns `Promise<string>`
- `settings.deposit.getRefundDepositAllowed()` Get whether queued or stalled deposit refunds are currently enabled; returns `Promise<boolean>`
- `settings.deposit.getWithdrawalAllowed()` Get whether deposit withdrawals are currently enabled; returns `Promise<boolean>`
- `settings.deposit.getStakingWithdrawalFeePerc()` Get the fee charged to deposits withdrawn from a staking minipool as a fraction; returns `Promise<number>`
- `settings.deposit.getCurrentDepositMax(durationId)` Get the current maximum deposit amount for the specified staking duration (string) in wei; returns `Promise<string>`

Group Settings:

- `settings.group.getDefaultFee()` Get the default fee Rocket Pool charges to group users as a fraction of rewards; returns `Promise<number>`
- `settings.group.getMaxFee()` Get the maximum fee Rocket Pool can charge to group users as a fraction of rewards; returns `Promise<number>`
- `settings.group.getNewAllowed()` Get whether new group creation is enabled; returns `Promise<boolean>`
- `settings.group.getNewFee()` Get the fee required for new groups to be registered in wei; returns `Promise<string>`
- `settings.group.getNewFeeAddress()` Get the address that group registration fees are sent to; returns `Promise<string>`

Minipool Settings:

- `settings.minipool.getMinipoolLaunchAmount()` Get the amount of ether required to launch a minipool (depositing it into the beacon chain) in wei; returns `Promise<string>`
- `settings.minipool.getMinipoolCanBeCreated()` Get whether new minipool creation is currently possible; returns `Promise<boolean>`
- `settings.minipool.getMinipoolNewEnabled()` Get whether new minipool creation is currently enabled; returns `Promise<boolean>`
- `settings.minipool.getMinipoolClosingEnabled()` Get whether minipools can currently be closed; returns `Promise<boolean>`
- `settings.minipool.getMinipoolMax()` Get the maximum number of minipools in the network (0 = unlimited); returns `Promise<number>`
- `settings.minipool.getMinipoolWithdrawalFeeDepositAddress()` Get the address that Rocket Pool fees are sent to; returns `Promise<string>`

- `settings.minipool.getMinipoolTimeout()` Get the period in seconds after which a minipool will time out if it has not begun staking; returns `Promise<number>`
- `settings.minipool.getMinipoolActiveSetSize()` Get the maximum size of the active minipool set (the set of minipools that deposits are assigned to); returns `Promise<number>`
- `settings.minipool.getMinipoolStakingDuration(durationId)` Get the staking duration for the specified duration ID in blocks; returns `Promise<number>`

Node Settings:

- `settings.node.getNewAllowed()` Get whether new node registration is currently enabled; returns `Promise<boolean>`
- `settings.node.getEtherMin()` Get the minimum ether required for a new node account to register, in wei; returns `Promise<string>`
- `settings.node.getInactiveAutomatic()` Get whether nodes are automatically made inactive after failing to check in; returns `Promise<boolean>`
- `settings.node.getInactiveDuration()` Get the period in seconds after which a node will be made inactive after failing to check in; returns `Promise<number>`
- `settings.node.getMaxInactiveNodeChecks()` Get the number of other nodes that a node will check for inactivity on checkin; returns `Promise<number>`
- `settings.node.getFeePerc()` Get the fee charged to users by nodes as a fraction of rewards; returns `Promise<number>`
- `settings.node.getMaxFeePerc()` Get the maximum fee that can be charged to users by nodes as a fraction of rewards; returns `Promise<number>`
- `settings.node.getFeeVoteCycleDuration()` Get the duration in seconds of a node fee voting cycle; returns `Promise<number>`
- `settings.node.getFeeVoteCyclePercChange()` Get the amount that the node fee changes per voting cycle as a fraction of rewards; returns `Promise<number>`
- `settings.node.getDepositAllowed()` Get whether node deposits are currently enabled; returns `Promise<boolean>`
- `settings.node.getDepositReservationTime()` Get the duration in seconds that a deposit reservation remains valid for; returns `Promise<number>`
- `settings.node.getWithdrawalAllowed()` Get whether node withdrawals are currently enabled; returns `Promise<boolean>`

Tokens

Overview

The `tokens` module manages the various Rocket Pool tokens, and is broken down into two submodules:

- `tokens.reth`: Manages rETH token interactions
- `tokens.rpl`: Manages RPL token interactions

Each submodule has the same interface for interacting with its underlying ERC-20 token. Mutator methods are restricted to their respective accounts.

Methods

- `tokens.[token].balanceOf(account)`: Get the token balance of the specified account (address) in wei; returns `Promise<string>`
- `tokens.[token].allowance(account, spender)`: Get the allowance of the specified account, for the specified spender (addresses) in wei; returns `Promise<string>`
- `tokens.[token].transfer(to, amount, options, onConfirmation)`: Transfer the specified amount of tokens in wei (string) to the 'to' address; returns `Promise<TransactionReceipt>`
- `tokens.[token].approve(spender, amount, options, onConfirmation)`: Approve an allowance of the specified amount in wei (string) for the specified spender (address); returns `Promise<TransactionReceipt>`
- `tokens.[token].transferFrom(from, to, amount, options, onConfirmation)`: Transfer the specified amount of tokens in wei (string) from the 'from' address to the 'to' address; returns `Promise<TransactionReceipt>`

1.4 The Rocket Pool Smart Node Stack

1.4.1 Introduction

The Rocket Pool Smart Node software stack provides all of the necessary infrastructure for running a Smart Node in the Rocket Pool network. The software stack consists of a number of Docker containers, as well as a command-line utility for performing tasks such as starting and stopping the Rocket Pool service.

1.4.2 Contents

Getting Started

OS & Hardware Requirements

Currently, the Rocket Pool Smart Node software stack is only officially supported on Ubuntu 16.04 and up. Support for additional operating systems will be added incrementally, after successful testing of the existing version. However, the majority of the stack runs within Docker containers, so running it on other Unix-based operating systems should require minimal customization.

The Smart Node stack requires at least 3GB of memory and 8GB of (SSD) hard disk space in order to run.

Note that a node operator must have **root** access to their node in order to install the dependencies and register the services required by the Smart Node stack.

Installation

Firstly, check if you have cURL installed on your system by running the following command in your terminal:

```
curl --version
```

If you don't, install it:

```
sudo apt-get install curl
```

Then, the Smart Node stack can be installed by running the following command:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/download/0.0.1/
↪setup.sh -o setup.sh && chmod 755 setup.sh && ./setup.sh && rm setup.sh
```

This will download the installation shell script, run it, and remove it once complete. For verbose output, add a `-v` flag to the setup command:

```
curl -L https://github.com/rocket-pool/smartnode-install/releases/download/0.0.1/
↪setup.sh -o setup.sh && chmod 755 setup.sh && ./setup.sh -v && rm setup.sh
```

The installation script will perform the following actions:

- Install the following OS-level dependencies via `apt-get`:
  - `apt-transport-https`
  - `ca-certificates`
  - `gnupg-agent`
  - `software-properties-common`
  - `docker-ce`
- Install the `docker-compose` tool via cURL
- Install the `rocketpool` command-line utility
- Download all Docker images required by the Smart Node stack
- Create a Rocket Pool data folder at `~/.rocketpool`
- Set the `RP_PATH` environment variable to the data folder
- Download various configuration files used by Docker to the data folder

The installation script can be run safely if any of the listed software is already installed; these items will be skipped. However, any existing Rocket Pool Docker configuration files will be overwritten.

Once installation has finished, you will be prompted to answer some questions for the initial configuration. Then, you'll be prompted to restart your terminal, and you can begin!

The Rocket Pool Service

Starting the Service

Start the Rocket Pool service by running:

```
rocketpool service start
```

This will "build up" the Smart Node stack, which runs it and also ensures that it stays running. If any of the running containers crash or you restart your node, Docker will start them back up to ensure that no uptime is lost.

You can check that the containers are running correctly with:

```
docker ps
```

You should see (in any order) entries for containers with `IMAGE` names similar to the following:

- `rocketpool/smartnode-cli:v0.0.1`
- `rocketpool/smartnode-node:v0.0.1`

- `rocketpool/smartnode-minipools:v0.0.1`
- `rocketpool/smartnode-watchtower:v0.0.1`
- `ethereum/client-go:stable`
- `traefik`

Pausing the Service

If you want to pause the Rocket Pool service for any reason, run:

```
rocketpool service pause
```

This will stop all running containers, suspending their execution, but leave them intact. Note that this will stop validators from performing their validation duties, so use this command with caution. The service can be started up again with `rocketpool service start`.

Stopping the Service

If you have finished interacting with the Rocket Pool network and want to stop the service entirely, run:

```
rocketpool service stop
```

This will "tear down" the Smart Node stack, stopping and removing all running containers, and deleting their state. Not only will validators stop performing validation duties, but all Ethereum clients will need to re-sync if the service is restarted. It is advised to use this command only if the node has no minipools remaining and no balances to withdraw in its network contract.

As a security measure, node data at `RP_PATH` (including the node & validators' private keys) will be preserved, and must be manually deleted if desired (this is not recommended).

Reconfiguring the Service

If you want to make any configuration changes to the Rocket Pool service, run:

```
rocketpool service config
```

This will repeat the configuration prompts that were run on installation, and will overwrite your node's Docker configuration file accordingly. For the changes to take effect, restart the Rocket Pool service with `rocketpool service start`.

Scaling the Service

This feature creates additional containers for an image, and is for advanced use only. For example, to run 3 Ethereum PoW client containers in parallel, run:

```
rocketpool service scale pow=3
```

RPC requests from other containers will be load-balanced between the available PoW client containers.

Viewing Service Information

You can view the logs for all running containers in real-time with:

```
rocketpool service logs
```

To view the logs for a single container, add its name at the end, e.g.:

```
rocketpool service logs pow
```

Press Ctrl-C to stop.

You can also view the hardware usage for each container with:

```
rocketpool service stats
```

Press Ctrl-C to stop.

Node Setup, Registration & Management

Initializing Your Node

With the Rocket Pool service running, the first thing you'll need to do is initialize your node password & account:

```
rocketpool node init
```

This will prompt you to enter a node password, and will then save it to disk along with a newly generated private key for your node account. You won't need to enter your node password again, it will simply be used by the Smart Node to unlock your account.

Your node password is stored at `~/.rocketpool/password`, while your private key is stored under `~/.rocketpool/accounts/`. Feel free to back these up in a safe and secure storage area which can't be accessed by anyone else.

You can make sure your account was created successfully with:

```
rocketpool node status
```

This should display something like: `Node account 0x0123...0123 has a balance of 00.00 ETH, 00.00 rETH and 00.00 RPL.`

Seeding Your Node Account

Next, you'll need to load your node account up with ETH and RPL to deposit into Rocket Pool. You'll also need at least 1 ETH in order to register with Rocket Pool. This isn't paid to Rocket Pool - we simply ensure that node operators have enough to cover their gas costs.

Special commands are available to withdraw ETH and RPL from faucets to your node account, for the Rocket Pool beta only. You can check your current faucet allowances with:

```
rocketpool faucet allowance
```

Then, withdraw resources with:

```
rocketpool faucet withdraw [amount] [unit]
```

where `[amount]` and `[unit]` specify how much of which resource ("ETH" or "RPL") to withdraw. Note that it may take a couple of minutes for tokens to be minted to your node account; you can check its balances again with:

```
rocketpool node status
```

Registering Your Node

Once you're ready, register with:

```
rocketpool node register
```

You will be prompted to either detect your timezone location automatically, or enter it manually. This information is not used for KYC purposes, but is sent to Rocket Pool during registration in order to display accurate node information to users. You may abstain by manually entering a location such as `Hidden/Hidden`.

Once you've registered successfully, you can check your status again:

```
rocketpool node status
```

This should now display additional information like: `Node registered with Rocket Pool with contract at 0xcdef...cdef, timezone 'Etc/Etc' and a balance of 2.00 ETH and 0.00 RPL`

Registering your node will create a new node contract which is used to interact with the Rocket Pool network. Some operations (such as depositing to Rocket Pool or withdrawing from minipools which have finished staking) will affect the ETH and token balances of this contract. The node contract is linked directly to your node account, restricting access of these operations to you.

Updating Your Registration

If you want to update the timezone your node is registered in, run:

```
rocketpool node timezone
```

This will repeat the prompts run during registration, and update your node's information in the network.

Withdrawing From Your Node Contract

If one of your node's minipools stalls (expires after a long period of inactivity) or withdraws from staking, its ETH & RPL balances will be sent to your node contract. You can simply leave these to be used for future deposits, or to withdraw them to your account, use:

```
rocketpool node withdraw [amount] [unit]
```

where `[amount]` and `[unit]` specify how much of which resource ("ETH" or "RPL") to withdraw.

Sending From Your Node Account

If you want to send ETH or tokens from your node account to another Ethereum address at any time, use:

```
rocketpool node send [address] [amount] [unit]
```

This will send the specified amount of ETH, rETH or RPL from the node account to the specified address.

Node Fee Voting

Setting the Target User Fee

All nodes within the Rocket Pool network charge users the same fee, based on a percentage of staking rewards earned. This fee changes dynamically based on votes from node operators in a similar fashion to how the Ethereum block gas limit is influenced by miners.

You can set your target user fee to vote for with:

```
rocketpool fee set [percent]
```

During each voting cycle, if the current network user fee is less than or greater than your target fee, your node will vote to raise or lower it respectively. Otherwise, it will vote to leave it as is.

Note that the network user fee only moves up or down by a small amount each voting cycle to prevent large swings or unpredictability. The user fee is also "locked in" on each minipool when it begins staking, in the interest of fairness to both node operators and users.

You can view information about the current network user fee, and your target user fee, with:

```
rocketpool fee display
```

Making & Managing Deposits

Checking Deposit Requirements

Before making a deposit, you'll need to load your node account up with the required ETH and RPL. Deposits always require 16 ETH, but the amount of RPL varies depending on current Rocket Pool network utilisation. To check on the current RPL requirements and network utilisation, run:

```
rocketpool deposit required
```

This will display a series of message like: `Depositing 16.00 ETH for 3m requires 32.00 RPL @ 2.00 RPL / ETH. Current network utilisation for 3m is 30%.`.

Making a Deposit

You can make a deposit with:

```
rocketpool deposit make [duration]
```

where `[duration]` is the time period you want to stake for.

Because of the dynamic nature of the RPL requirement, deposits are performed in two steps. First of all, they are "reserved", which locks in the RPL requirement for the deposit for 24 hours. Then, they are completed with a second transaction. This gives you time to acquire the necessary ETH and RPL without having to worry about fluctuating prices.

After your deposit is reserved, its ETH & RPL requirements, staking duration and expiry time will be displayed. Then, you will be prompted to select one of the following options:

1. Complete the deposit
2. Cancel the deposit
3. Finish later

Completing the deposit will immediately complete the process and deposit your ETH and RPL into Rocket Pool. This requires the necessary ETH and RPL to be sent to the node contract. If the node contract's balances are insufficient, you will be prompted to send ETH and/or RPL to it from your node account. After successfully completing the deposit, your new minipool's address will be displayed.

Canceling the deposit will cancel the reservation so that you can create a new one later. This may be useful if, for example, you want to wait for the RPL requirement to drop and deposit at a lower RPL cost.

Finishing later simply stops the deposit process until you run the command again at a later time. When you do, it will pick up where you left off.

Managing & Withdrawing From Minipools

Checking Minipool Status

Once you have made one or more deposits from your node, you can view the status of your created minipools with:

```
rocketpool minipool status [filter]
```

where `[filter]` is an optional status to filter the list by (`initialized`, `prelaunch`, `staking`, `loggedout`, `withdrawn`, `timedout`). This will list the following properties of all minipools created by your node:

- `Address`: The minipool's address
- `Status`: The minipool's current state in its lifecycle, one of:
  - `initialized`: The minipool has been created and has no user deposits assigned yet
  - `pre-launch`: The minipool has user deposits assigned to it but not enough to begin staking
  - `staking`: The minipool has filled with user deposits, sent its balance to the beacon chain, and begun staking
  - `logged out`: The minipool has been logged out from the beacon chain but is not yet ready to withdraw from
  - `withdrawn`: The minipool has withdrawn from the beacon chain and its rewards may now be withdrawn as rETH
  - `timed out`: The minipool timed out after a period of inactivity and deposited ETH and RPL may be withdrawn from it
- `Status Updated @ Time`: The time at which the minipool reached its current state
- `Status Updated @ Block`: The block at which the minipool reached its current state
- `Staking Duration`: The duration the minipool will stake for before logging out from the beacon chain
- `Staking Total Blocks`: The total number of blocks that the minipool will stake for
- `Staking Until Block` (staking minipools only): The block that the minipool should log out at
- `Staking Blocks Left` (staking minipools only): The number of blocks remaining until the minipool should log out

- `Staking Complete Approx` (staking minipools only): The approximate time at which the minipool should log out
- `Node Deposit Withdrawn` (withdrawn minipools only): Whether you have already withdrawn your deposit & rewards from the minipool
- `Node ETH Deposited`: The ETH balance of the deposit you made to the minipool from your node contract
- `Node RPL Deposited`: The RPL balance of the deposit you made to the minipool from your node contract
- `User Deposit Count`: The number of user deposits assigned to the minipool
- `User Deposit Total`: The total amount of user deposits in ETH which the minipool is currently holding
- `User Deposit Capacity`: The total amount of user deposits in ETH which the minipool can hold

Withdrawing From Minipools

Once one or more of your node's minipools have finished staking, you can withdraw your rewards and RPL from them with:

```
rocketpool minipool withdraw
```

This will list all of your minipools which are available for withdrawal and prompt you to select one (or all) of them to withdraw from. Initialized minipools (newly created ones with no user deposits assigned yet) can be withdrawn from and destroyed, but will not be affected if you select "all". Withdrawn rETH will be sent to your node account, while withdrawn RPL will be sent to the node contract to be re-used in your next deposit. If you want to withdraw it to your node account too, run:

```
rocketpool node withdraw [amount] rpl
```

Command Reference

Service Commands

- `rocketpool service start`: Start the Smart Node containers
- `rocketpool service pause`: Stop the execution of the Smart Node containers
- `rocketpool service stop`: Stop the execution of and remove all Smart Node containers and their state
- `rocketpool service scale [service=NUM]`: Scale the number of containers for a Smart Node service
- `rocketpool service config`: Reconfigure the Smart Node service (requires restart for changes to take effect)
- `rocketpool service logs [services]`: View the logs for the Smart Node stack or for an individual container
- `rocketpool service stats`: View resource usage statistics for the Smart Node stack

Node Commands

- `rocketpool node status`: View the node's status and balances
- `rocketpool node init`: Initialise the node with a password and an account

- `rocketpool node register`: Register the node with Rocket Pool
- `rocketpool node withdraw [amount] [unit]`: Withdraw the specified amount of ETH or RPL from the node contract
- `rocketpool node send [address] [amount] [unit]`: Send the specified amount of ETH, rETH or RPL from the node account to the specified address
- `rocketpool node timezone`: Change the timezone location the node is registered under

Faucet Commands

- `rocketpool faucet allowance`: Check your ETH and RPL faucet allowances
- `rocketpool faucet withdraw [amount] [unit]`: Withdraw the specified amount of ETH or RPL from the faucet to your node account

Node Fee Commands

- `rocketpool fee display`: Display the current network user fee and the target user fee to vote for
- `rocketpool fee set [percent]`: Set the target user fee percentage to vote for

Deposit Commands

- `rocketpool deposit required`: View the current network RPL requirements and utilization stats
- `rocketpool deposit make [duration]`: Make a deposit for the specified staking duration

Minipool Commands

- `rocketpool minipool status [filter]`: Check the status of all minipools owned by the node, optionally filtered by status
- `rocketpool minipool withdraw`: Withdraw ETH or rETH and RPL from withdrawn or stalled minipools