# The Currency of Trust

Technical Paper v1.0

# Contents

Introduction ........ 3
1 The Blockchain ........ 4
1.1 Description ........ 4
1.1.1 Subsystems ........ 4
1.1.2 The P2P Network ........ 4
1.1.3 The Global Shared State and Execution Layer ........ 4
1.1.4 Consensus Layer ........ 5
1.1.5 Block Execution Layer ........ 6
1.2 Graphene Trading Engine ........ 6
1.3 Transaction Types ........ 7
1.4 Transaction Fees ........ 7
2 Tokens ........ 8
2.1 DasCoin ........ 8
2.2 WebEuro ........ 8
2.2.1 Issuance Authority ........ 8
2.2.2 Pegging Mechanism ........ 8
2.3 Cycles ........ 8
3 Minting and issuance ........ 9
3.1 Description ........ 9
3.2 Licence levels ........ 11
4 Governance ........ 12
4.1 Structure ........ 12
4.1.1 Voting Nodes ........ 12
4.1.2 The DasCoin Board ........ 12
4.1.3 Chain Authorities ........ 13
5 Frequency ........ 16
5.1 Description ........ 16
5.2 Global Frequency ........ 16
5.3 Centralize vs. Calculation Algorithm ........ 16
5.4 Upgrades ........ 16
5.4.1 Description ........ 16

5.5 Current Cycle implementation .... 17
5.6 Bonus Cycles .... 17
5.7 Cycles Issuance authority description .... 17
6 Nodes .... 18
6.1 Master Nodes .... 18
6.2 Ledger Nodes .... 18
6.3 Node Operations .... 19
6.3.1 Create Witness Operation .... 19
6.3.2 Update Witness Operation .... 19
6.3.3 Remove witness operation .... 20
6.3.4 Activate witness operation .... 20
7 WebWallet .... 21
7.1 Security .... 22
7.2 Validator .... 22
7.2.1 Account recovery .... 23
7.3 Identity management .... 23
7.3.1 KYC .... 23
7.4 Sample UI .... 24
8 API .... 25
8.1 Websocket Calls & Notifications .... 25
8.1.1 Requesting API access .... 26
8.1.2 Database Notifications .... 27
8.1.3 Example Session .... 28
8.2 Operations .... 28
9 Appendix A: Blockchain API .... 29
9.1 Database API .... 29
9.2 Account History API .... 46
9.3 Crypto API .... 47
9.4 Network Broadcast API .... 48
9.5 Network Nodes API .... 48
9.6 Wallet API .... 49
10 Appendix B: Operations .... 83

# Introduction

DasCoin has been designed to solve the core problems inherent to storing and exchanging value. The DasCoin Blockchain is a mutual distributed ledger that creates and distributes cryptographic assets, and then securely facilitates their storage and exchange. A prime objective of DasCoin is to use the infrastructure of a digital asset system to build an effective network of trust, enabling all participants and stake holders to share a common goal of increasing the value of the network and cultivating its growth. A private, permissioned blockchain architecture has been incorporated due to its enhanced security, inherent efficiency, and ability to scale more easily (due to deployment control). Fortifying this secure foundation is the authentication of all users in accordance with bankingstandard KYC (Know Your Customer) requirements and the implementation of a “hardware required” digital wallet system.

However, just like every other system of value DasCoin must establish a few fundamental elements. These include defining: initial money supply, initial distribution, basis of value, expansion/contraction mechanisms of the money supply, who controls the means of production, and the allocation of inflation (and/or allocation of credit). DasCoin offers a hybrid structure to solve the issues associated with these economics-based elements.

This technical whitepaper will go in depth about the technical specifications of the DasCoin blockchain that will establish these fundamental elements. We will start with a basic description of the blockchain to give the reader a general overview. We will follow up with more chapters that will clarify the specific individual parts that are used in the blockchain. At the end we will include a full list of API calls that are used in this paper. To finish off, we added an appendix for a more detailed explanation about the API itself.

# 1 The Blockchain

## 1.1 Description

The DasCoin Blockchain is a distributed ledger-based payment and token exchange system that supports the base system tokens (Cycles, DasCoin) and enables the creation of an arbitrary number of different asset tokens. The DasCoin Blockchain determines block production by assigning authorized nodes (master nodes) as block validators. The system relies on the centralized distribution of DasCoin through the minting process as well as a semi-decentralized governance mechanism in which the DasCoin Board determines chain parameters and determines block producers from eligible candidates.

The DasCoin Blockchain is built upon the Graphene toolkit, which in itself is the direct basis for the Bitshares blockchain and indirectly for the Steem and EOS blockchains. The DasCoin blockchain inherits the base consensus model from Graphene, as well as the P2P network, the execution of user operations, and system wide on-block-produced actions. The main differences lie in the implementation of governance and DasCoin distribution (minting), as well as the storing of user states and the implementation of user and administrative actions via on-chain operations.

### 1.1.1 Subsystems

The DasCoin Blockchain can be seen as a union of subsystems which handle the tasks of facilitating communication between nodes in the system, executing user and block operations, updating the global shared state, and maintaining the ledger of confirmed transaction updates.

### 1.1.2 The P2P Network

The P2P node network is the base layer of the DasCoin blockchain. It is a part of the Graphene Toolkit. The network is responsible for handling connections between nodes and user endpoints (wallet instances) through websockets, as well as relaying messages between nodes.

Upon receiving messages from a client endpoint, the network layer handles unmarshalling of the message and passes it off to the operations layer. The network layer also marshals the response back to the client (if any) regarding the success of the state update, query or interaction.

The network layer also handles the connections between nodes. The nodes do not have a discovery protocol, relying instead on preconfigured addresses. The network layer maintains the gossip state update protocol. Each node broadcasts relevant messages such as valid transactions and created blocks to all known nodes. It is left to the consensus mechanism and the global shared state fto handle deduplication of updates.

### 1.1.3 The Global Shared State and Execution Layer

The Global Shared state is an in-memory database of all relevant user and system data objects. It is a part of the Graphene Toolkit. All nodes have the entire copy of the global shared state. All objects have a unique identifier based on a three-number system (denoting domain, type and in-order number of instantiation). All object types can be accessed via a generalized object API, while more specific formatting APIs handle client needs.

The Global Shared State features a transaction rollback mechanism, necessary for establishing consensus. The rollback mechanism can detect and roll back the state in case of a dominant block history emerging, up to a maximum defined divergence point. This is most commonly manifested on block updates, where pending transaction state is discarded and replaced with the result of the confirmed transactions in the blocks.

The execution layer verifies and executes user-initiated operations. The operations received from the network layer are first checked for the presence of correct signatures. If an operation is validly signed, it is then first validated for checks independent of the current state (ie. for sending negative balance on transfer). If the operation is properly signed and valid independent of the current state, it is then checked against relevant objects in the state (ie. a transfer would check if the FROM account balance is sufficient). Once the operation is fully verified, the state is locked, and the operation is executed on the current state, performing a state transition. This multi-step validation system ensures an early fail of invalid transactions and lowers the time of execution. Operations can be additionally bundled into transactions, with similar functionality as in database management systems. Executed transactions are sent to the network layer to be rebroadcast to all nodes. The execution layer also features a deduplication mechanism, making sure that two of the same transaction cannot be executed.

Executed transactions (and thus operations) are placed in a pending state. They are verified and executed but have not been confirmed by consensus. The state of the node always matches the last transaction executed, regardless of its confirmed state. Once a block update is received, it is checked for signatures and whether it matches the block production schedule (is it signed by the right block producer at the right time). If that is the case, the transactions are rolled back to the last joining point - the next block in the blockchain. Unless there is a chain split, this is usually the last block. The transactions from the block update are then executed in order and a new confirmed state is formed.

### 1.1.4 Consensus Layer

The consensus layer ensures byzantine fault tolerance - the guarantee that the entire network can converge to a globally shared state. It also updates the local copy of the block ledger as well as maintains the set of execution rules and platform rules. The consensus layer is a modified Graphene implementation. DasCoin blockchain consensus is based on a known set of validators, which are nodes that have an authorized block signing private key. A block signed with the correct master key is properly signed. The block producers are added to the list of active block producers by the DasCoin Board, through signing the appropriate operation. The board also has the privilege to remove a master node from the block producer list.

The block production algorithm proceeds in rounds. Each block producer is shuffled into a schedule which is valid for a single round. Each block producer is given a time slot based on their respective position within the schedule. Each slot is the length of a unit of block time (initially set to 6 seconds). It is the duty of the block producer to collate all user transactions, verify and execute them, produce a signed block and send it through the network. The nodes actively listen to block updates. If a block update is received by a node on the network, and it is properly signed, it is checked against the schedule. If the block producer is not scheduled to produce a block at that moment in time, the block is rejected. This ensures that all block producers have a chance to produce a block, thus distributing trust

through the network, while at the same time maintaining a fast and predictable speed of confirmed transactions.

### 1.1.5 Block Execution Layer

Certain system wide actions can be attached to the predictable rhythm of producing blocks. Each node executes certain system defined code when a block is produced. This can also be set to a multiple number of blocks (every 10th block, for example, corresponds currently to each minute). This enables actions that are time based, with the maximum resolution of the block time (currently 6 seconds). The process of minting DasCoin, parts of trade engine operation, as well as cleanup of unused memory is implemented on this layer.

The system also periodically reserves time (currently each day) for a system wide maintenance interval during which block production is stalled. This is to allow all nodes to compute resource intensive changes of state - as no parallel execution is available. All changes that require reading or writing to a large part of the state (such as upgrades) are relegated to this interval.

## 1.2 Graphene Trading Engine

The decentralized exchange (DEX) is built on top of the consensus and operations layer of the blockchain. It's a native application feature that does not rely on user defined code execution, as all functionality is built in to the node software itself. The DEX and trading engine are both parts of the Graphene toolkit.

The DEX allows users to create limit orders to trade different assets on the DasCoin blockchain. At the present moment, the exchange supports the trade of DasCoin (DASC) and WebEuro (w€) assets. When a user creates a trade, it is placed on a market. The market tracks all open and filled orders for a certain trading pair and tracks the last price and 24h volume. There are two markets defined (DASC:w€ and w€:DASC), but the DEX supports an arbitrary number of markets (as well as assets). Any order is matched by the engine by the criteria of best price and time of creation once a fitting order is created on the DEX. The orders can be partially filled, with the remainder remaining in an open position. A user also has the option to create a fill-or-kill order that must be matched on creation. Users may also cancel open orders at any time.

The DEX is enabled by user signed operations that create and cancel limit orders. The user must properly sign a valid create limit order operation in order to create a limit order. The operation is considered valid if the user has enough funds to create the order. Once the operation is executed, the trading engine will attempt to match the order. This is NOT tied to block creation or the consensus mechanism of confirming transactions. The trading engine will proceed to search all limit orders for a given market pair sorted by price ratio and time of creation. If a match is made, and the order is fully filled, the trading engine will exchange the assets between trading accounts, swapping the appropriate amounts of tokens to complete the trade. If the order is partially matched, or if it is not matched, a unique order object is created on the global share state, and the user is notified of its id. This unique order object contains the information about the amount and type of tokens to sell, as well as the minimal amount to receive. The values are equal to the original values input by the user in the create limit order operation if the order was not filled, or they are adjusted to match the partial filling of the order if a match has happened, but

the order was not fully filled. This object persists in the state until it is either filled entirely by another created limit order or cancelled by the user. The user may cancel an open limit order by properly signing a valid cancel limit order operation. The operation is valid if it contains an id of an existing limit order object and is signed by the creator of the object (and the limit order itself).

An important thing to note is that trades are executed immediately, regardless of confirmation. To achieve additional confidence, the user may wish to wait for a confirmed limit order to appear, meaning only such order that was confirmed in a block, and then trade in accordance with the confirmed order.

### Block Types

As explained in the *Description* section of this document, the DasCoin blockchain contains a mechanism for executing system code when a block is being produced. This mechanism is used on Minting blocks, which are blocks designated to distribute DasCoin by the way of the minting queue. Blocks may also be designated to perform staking rewards and to execute delayed user code. For more information about these operations see Appendix: Operations.

## 1.3 Transaction Types

As explained in the *Description* section of this document, the DasCoin blockchain is based on user signed operations that modify the global shared state. All operations are bundled into transactions in such a way that all operations in a transaction must pass in order or they all fail and are rolled back. Each transaction thus must contain at least one operation.

Operations can be divided between authority and user operations. Authority operations must be signed by a named authority and are responsible for governing the chain and handling actions that require trust. A good example would be choosing block producers (handled by root authority) and issuing licenses (handled by license authority). Both of these actions require oracle input from outside the chain, and as such must be signed by trusted authorities. User operations can be signed by any user. They are actions that manipulate user state (such as token balance, open trade orders, etc.). User operations are valid only in the bounds of user state (and thus must be properly signed).

For more information and a list of all operations see Appendix B: Operations.

## 1.4 Transaction Fees

Almost all exchanges and digital wallets charge a fee for all conducted transactions. And in almost all cases, this fee is calculated in the currency you are trading and comes in the form of a percentage of the amount being traded. However, since the cryptocurrency market is so volatile, and the price of a coin can rise or fall drastically in a short period of time, the actual fee a user pays depends on the current value of the coin they are trading.

To avoid this kind of volatility, instead of charging fees in DasCoin, we will instead be using Cycles – the heart of the DasCoin platform – for our transaction fees. Any transaction a user conducts, regardless of the amount they are sending, will always cost only one Cycle. If a user uses up all their Cycles, they'll always be able to purchase more. The last DasCoin in a user's account is always reserved for the purchase of Cycles, so they can never get into a situation when they have neither Cycles nor DasCoin with which to purchase them.

# 2 Tokens

## 2.1 DasCoin

DasCoin is a hybrid currency designed to combine the best qualities of decentralized cryptocurrencies with the best aspects of centralized currencies – and eliminate their respective weaknesses. DasCoin is the convertible “store of value” unit that serves as the foundation of the digital asset system. DasCoin is our main crypto asset, and as previously described, customers acquire DasCoin through licenses and minting.

## 2.2 WebEuro

WebEuro is a cryptographic asset defined in the DasCoin Blockchain that represents the Euro denomination.

### 2.2.1 Issuance Authority

Issuance authority that is responsible for WebEuro is called issue_webasset, you can find more details about this operation in the Appendix section.

### 2.2.2 Pegging Mechanism

As mentioned before, WebEuro is an equivalent of a FIAT currency Euro, so its value has a fixed exchange rate. You earn WebEuro through trading of DasCoin on the internal market or through NetLeaders Commission bonuses. Customers can cash out their WebEuros into the same amount of Euros manually through our NetLeaders platform.

## 2.3 Cycles

Cycles are a cryptographic asset defined in the DasCoin Blockchain. Cycles represent stored capacity within DasNet and can be used either for paying fees in the DasCoin Blockchain, for paying network services, or they can be submitted to the minting queue to be exchanged for DasCoins. A cycle is an indivisible asset, meaning that we cannot have a half of a cycle, or a quarter. Cycles can only be acquired through the purchase of a system license using either Bitcoin or Euros or can be bought for DasCoin in order to buy fees on the DasCoin Blockchain.

Internally, cycles are represented as an asset, equal to WebEuro or DasCoin, but they cannot be traded on the Exchange nor can they be transferred from one wallet to another. They can still be transferred from a license on a vault which has been tethered to a wallet.

Periodically, in accordance with a predetermined Upgrade Interval, all Cycle balances will experience an Upgrade and those balances in the accounts will double. If Cycles are already in the DasCoin Minting Queue, these Cycles will be unaffected (unless they are associated with an account that has a Frequency Lock).

# 3 Minting and issuance

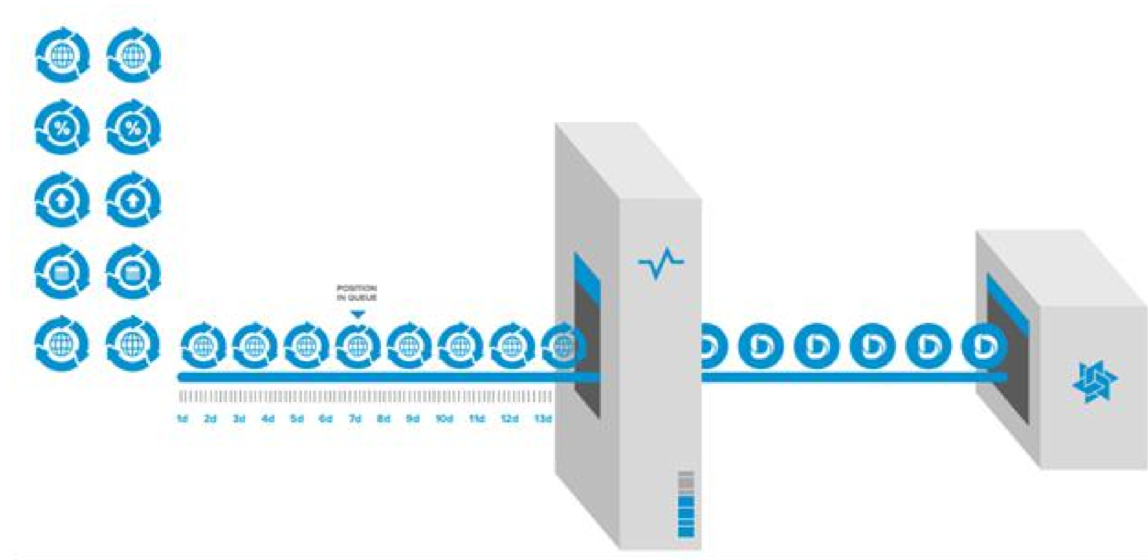
## 3.1 Description

The process of producing and distributing new DasCoins is known as Minting. Minting allows a person or entity to store value in the form of DasCoin. In order to obtain DasCoins people can submit Cycles to the network and then be assigned a place in the DasCoin Minting Queue. The Minting Queue functions on a first-in-first-out basis. DasCoins are distributed to the account that is next in line in the distribution queue. At each DasCoin Distribution Interval a specific amount of DasCoins are distributed to participants in the queue.

The amount of DasCoins a user receives is regulated by an adjustable conversion factor called Frequency. DasCoin distribution functions in such a way that the amount of Cycles submitted divided by the Frequency, equals the number of DasCoins that will be distributed to that person. Cycles are deducted at the time of submission to the “minting queue”, and at the time of distribution the DasCoins are automatically transferred to the account by the blockchain software.

The total amount of DasCoins that can be minted is capped at 2 to a power of 33.

There are no parties (neither executives nor developers) who are able to pre-mint, pre-mine or pre-distribute DasCoins to themselves. Cycles can only be received in exchange for value transferred to the system, and Cycles must be submitted to the system for there to be a direct distribution of DasCoins through the minting process.



For a customer to mint a DasCoin, they need to have cycles. Cycles are a resource defined in the DasCoin Blockchain. They can only be acquired through the purchase of a system license (NetLeaders) using either Bitcoin or Euros. They can either be used for network services or submitted in exchange for DasCoins. Cycles can only be received in exchange for value transferred to the system, and a user must submit them to the system in order to acquire DasCoins through the minting process.

Depending on the conversion factor (frequency), a customer will receive a certain amount of DasCoins at the end of the minting process. Frequency is the conversion factor used in the minting process of **Cycles** to **DasCoins**. The value of the Cycles you have in the blockchain is secured by the distributed ledger. The precise amounts are encoded in the blockchain state and are currently fixed. The only way to change the base amount of Cycles a license provides is by a vote involving all committed shareholders (of DasCoin), either through a direct vote or through delegates.

$$D_{as} = \frac{C}{F}$$

The Blockchain has a so called Minting queue that has the following elements:

- Amount of Cycles – the number of Cycles that where submitted by customers to the queue
- Frequency – a value determined by the license from which a user submits Cycles
- User ID – a unique account identifier

A customer can submit a certain number of Cycles from a license to the minting queue. The minting mechanism has two elements that determine how the queue is processed. The moment in time that the queue is processed is called a DasCoin reward event. During one DasCoin reward event there is only a certain amount of DasCoins that can be produced and distributed.

- Reward interval – the time interval between each DasCoin reward event
- Reward amount – the amount of coins that will be produced and distributed during the DasCoin reward event

For example, a Pro license has a higher number of Cycles than a Standard license, so we take a smaller amount of Cycles from the Standard license and a larger amount from the Pro. When DasCoin are minted, we redistribute the larger portion to the client who owned the Pro license and the smaller portion to the client who owned the Standard.

Your Cycles will begin to be converted to DasCoin and added to your wallet when your license is tapped for Cycles (which happens chronologically based on when your license was purchased i.e. clients who purchased licenses before you did will have their Cycles submitted before you will).

As more of your Cycles are used, they will, eventually, reach the determined amount of DasCoin that your license is supposed to receive. However, this is not an 'immediate' thing. It is a process that accumulates over time.

Upon license purchase Cycles are migrated to the WebWallet vault and the Cycles from the vault are submitted to the minting queue with a certain Frequency lock.

To calculate the amount of DasCoin you would receive from your initial submission, you would divide the amount of Cycles (C) associated with your license level by the Frequency Lock (FL) given at time of license activation.

When an upgrade period is reached, at this time, the amount of Cycles that have, thus far, been submitted to the DasNet system are matched by NetLeaders and the resulting amount of Cycles are resubmitted to the DasNet system for minting.

## 3.2 Licence levels

As previously mentioned, to participate in the minting of DasCoin, a customer needs to acquire a NetLeaders license.

A license provides the customer with three things:

- Authentication: Customers identity will be verified.
- Authorization: The customer will receive a license certificate and will be able to generate the keys to a WebWallet that will provide them access to DasNet.
- Capacity: The customer will receive Cycles – units of network capacity on DasNet.

Depending on the level of their license, a customer will receive a certain amount of Cycles that he or she can use to mint DasCoins.

There are currently 6 types of licenses:

| License Name | Number of Cycles | Bonus Cycles | Network Upgrades | Price | Frequency |
|---|---|---|---|---|---|
| Standard | 1100 | 440 | 1 | 100€ | Visit our website to see the current Frequency |
| Manager | 5500 | 2200 | 1 | 500€ | |
| Pro | 24000 | 9600 | 1 | 2000€ | |
| Executive | 65000 | 26000 | 2 | 5000€ | |
| Vice-president | 225000 | 90000 | 2 | 12500€ | |
| President | 325000 | 130000 | 3 | 25000€ | |

As this is something that can be subjected to change, you can see more details about Licenses and our marketing incentives through our NetLeaders platform here: https://netleaders.com/en/products

# 4 Governance

DasCoin is governed by elected individuals and businesses. The so-called committee (a set of many individuals), can change blockchain parameters such as block size, block confirmation time and others. Most importantly, though, they deal with the business plan of the blockchain and tweak costs and revenue streams (mainly transaction fees). Fortunately, the shareholders have the final say to approve the executive committee.

Hence, we see businesses competing for seats in the committee to define blockchain parameters.

If business ideas requires certain blockchain parameters or a particular set of fees to be profitable, there are several options:

- Argue with shareholders to approve committee members that vote in their favour
- Get elected as a committee member by showing that the business is worth being available in that particular chain
- Deploy the innovative business and have the shareholders approve the upgrade

## 4.1 Structure

### 4.1.1 Voting Nodes

Voting Nodes are non-authoritative influencers of the DasCoin Blockchain. These nodes do not control any of the daily functions of the DasCoin system, but fully control who sits on the DasCoin Board and what proposals are passed within the system. **Voting Nodes represent significant stakeholders who have demonstrated that they have a long-term view on DasCoin, and therefore are given the privilege of influence over chain governance.** Any qualified Voting Node can initiate a proposal. The DasCoin Board is responsible for organizing and presenting the proposals to the Voting Node population. There is also a mechanism within the voting system that allows the Voting Nodes to directly present a referendum to their membership without going through the Board review process (to prevent the Board from withholding a certain proposal from the Voting Nodes).

### 4.1.2 The DasCoin Board

The DasCoin Blockchain enables a governing board to regulate the parameters of the network. The DasCoin Board will be comprised of members elected by the Voting Nodes.

The role of the Board is to:

1. Propose and modify chain parameters to support the normal functioning and growth of the network
2. Delegate certain executive roles to certain chain executives (such as issuing licenses and authenticating said licenses)
3. To act as a check on the power of said executives by having the ability to terminate their access to the network.

The Board itself has no control on the state of the database or the construction of the DasCoin Blockchain and is programmatically prevented from making any changes to it. Since the network itself manages and maintains the state and the transaction ledger, the only way to make any undesired change is to subvert the majority of Master Nodes. The Board is designed to consist of 7 individual

Directors, each of whom is bestowed with full voting privileges. Generally, each Director serves for a 6-year term, though initial Directors will be serving staggered terms (of 2-6 years) to ensure continuity of experience. A minimum of 3 Directors are required for the governance of the DasCoin ecosystem, and as many as 9 may serve on the Board. In addition, there is an Ombudsman member of the Board, who does not vote and does not hold any responsibilities within the Board, but who attends all Board meetings and provides a degree of independence and transparency to the Board's governance process. Finally, there is an Executive Director who works for the Board and is responsible for ensuring that all of its decisions and initiatives are enacted and enforced. The Executive Director attends all Board meetings but is not permitted to vote. The Executive Director is responsible for directly overseeing all Chain Authorities

### 4.1.3 Chain Authorities

Chain authority roles exist to handle smooth inputs to the Blockchain of user data that exists outside of the system. The problem with fully decentralized systems is the fact that they cannot have reliable inputs: for example, Bitcoin is created internally in the Bitcoin blockchain and is merely transferred around. In order for Proof of Value to work, there must be certainty that the user is actually bringing value to the network. Value cannot exist without an independent observer – and so the only way to verify that the user has submitted value to the system is to maintain an impartial observer.

Each authority role is set up in such a way that:

1. There is no way for the authority to make a meaningful unwanted impact on the state of the network as the network can fall back to a failsafe state.
2. The actions of the authority are checked by a separate authentication authority and there are programed measures to assure there is minimal chance of collusion.
3. There are incentives to perform in the best interest of the network.
4. Any malicious action by the chain authority is transparent and will lead to that account being marked as untrustworthy, shut off from the network and penalized.

#### *4.1.3.1 DasCoin Board*

The Board sets the procedures and ensures the proper execution of the following:

- The level of the Frequency at each 2-week interval. The process involves the Board selecting which of a range of algorithms best reflects the current growth state of the network. The primary factors considered include:
    - Overall amount of Cycles in the system
    - Amount of Cycles authorized in the most recent 2-week period
    - Velocity of Cycle growth in previous periods, and projected Cycle growth within the system.
- The size of the Minting Blocks at the start of each Upgrade Interval
- The size of the Super Blocks and Treasury Blocks
- The proposals to be submitted to the Voting Nodes, the use of funds from Treasury Blocks
- The authorization of Master Nodes and Ledger Nodes within the DasNet infrastructure
- Intervention at times of crisis, and other elements

The Executive Director oversees the performance of all chain authorities and KYC functions, manages the flow of proposals for the Board to consider, facilitates referendums if the proper thresholds have been surpassed, is responsible for enforcing all Board initiatives and decisions, and is fully accountable to the Board for all operations within the system.

Chain authorities oversee the issuance and authentication of Licenses and WebEuros, under the supervision of the Executive Director.

Upgrades automatically occur on a system-wide basis on specific dates, starting 108 days from the launch of the DasCoin Blockchain. It is unlikely that this interval will be altered at any point in the future, but it remains within the capability of the Board to make such an adjustment should they feel it is needed.

Listed below is the set of parameters that the DasCoin Board can propose changes upon:

- **License Issuing Authority** – The privilege to assign a License to a Vault Account and to determine the level of license.
- **License Authenticating Authority** – The ability to cancel the issuance of a new License to an account in the event of error.
- **WebEuro Issuing Authority** – The privilege to transfer a WebEUR balance to a Vault Account.
- **WebEuro Authenticating Authority** – The ability to cancel the issuance of a WebEUR balance to an account in the event of error.
- **Cycle Upgrade Date & Interval** – The exact date of an Upgrade and the Upgrade Interval. The Upgrade Interval is current set at 108 days and is not expected to ever be changed.
- **Frequency** – The conversion factor by which Cycles can be exchanged for DasCoins as part of the minting process.
- **Block** – A measurement for accumulated transactions. The system currently records a block in accordance with a designated period, known as a Block Interval.
- **Block Interval** – The time it takes to create a confirmation, of a single block of transactions. By default, transactions are confirmed every 6 seconds. In the future, this will be decreased as the code base is further optimized.
- **Minting Block** – The amount of DasCoins distributed at the completion of each Minting Interval.
- **Minting Interval** – The time it takes for a Minting Block to be created. The default Minting Interval of the system is 10 minutes and is not expected to ever be changed.
- **Maintenance Period** – The number of blocks that must pass before maintenance is performed on the Blockchain.
- **Maintenance Skip Slots** – During a maintenance period some blocks will be skipped: this parameter sets how many the system should skip while performing a Maintenance Period.
- **Super Block** – The amount of DasCoins distributed at the completion of each type of Super Block Interval. The size of Super Blocks is expressed as a percentage of the cumulative DasCoins distributed by Minting Blocks within that Super Block Interval.
- **Super Block Intervals** – The time it takes for Super Blocks to be created. There are 3 Super Block Intervals within the system, each of which corresponds with a type of Super Block. The Voting Super Block Intervals are 1 week in duration, the Ledger Super Block Intervals are 2 weeks in duration, and the Master Super Block Intervals are 3 weeks in duration.

- **Treasury Block** – The amount of DasCoins distributed at the completion of a Treasury Block Interval. Like Super Blocks, Treasury Blocks are expressed as a percentage of the cumulative DasCoins distributed by Minting Blocks within the Treasury Block Interval.
- **Treasury Block Interval** – The time it takes for a Treasury Block to be created. Treasury Block Intervals are 4 weeks in duration.
- **Maximum Block Size** – Maximum size in bytes that a block can be that is signed to the Blockchain.
- **Maximum Transaction Size** – This is the maximum allowable size in bytes for a single transaction.
- **Maximum Witness Count** – This is the maximum number of Master Nodes that could be active on the network.

#### *4.1.3.2 License Issuing Authority*

Holds the privilege to assign a License to a Vault Account and to determine the level of license.

#### *4.1.3.3 License Authenticating Authority*

Holds the ability to cancel the issuance of a new License to an account in the event of error.

#### *4.1.3.4 License Administrator Authority*

Holds the ability to manage license limits, upgrades etc.

#### *4.1.3.5 WebEuro Issuing Authority*

Holds the privilege to transfer a WebEuro balance to a Vault Account.

#### *4.1.3.6 WebEuro Authenticating Authority*

Holds the ability to cancel the issuance of a WebEuro balance to an account in the event of error.

#### *4.1.3.7 Cycle Issuing Authority*

Holds the privilege to issue a Cycles to a Vault Account.

#### *4.1.3.8 Cycle Authenticating Authority*

Holds the privilege to allow cycle operations.

#### *4.1.3.9 Registrar Authority*

Holds the privilege to register an account.

#### *4.1.3.10 Personal Information Authority*

Holds the privilege of validation of personal information and key roll back.

#### *4.1.3.11 Wire out Authority*

Holds the privilege of Handling of wire_out payments.

# 5 Frequency

## 5.1 Description

**Frequency** is the conversion factor used in the minting process of **Cycles** to **DasCoins**. As the DasNet expands, a number of positive dynamics occur: risk is reduced, the circle of users widens, and the infrastructure of the system grows larger. Consequently, the value of a unit of stored value - DasCoin - in the system increases in relation to a unit of stored network capacity - Cycle. This results in more Cycles being needed to produce a single unit of DasCoin. This dynamic is reflected in an increase in the Frequency, which is proportionate to the incremental growth of the network. Creating an index based on the initial network size and increasing that index in direct proportion to the increase of the network size allows a steady decrease in the amount of DasCoin that is put into circulation. Steadily decreasing the level of DasCoin actually put into circulation allows for a certain amount of control over any potential loss of value due to inflation and ensures that DasCoin's monetary value and exchange rate are maintained and kept steady.

## 5.2 Global Frequency

There is basically no difference between the Global Frequency and the Frequency that you use in the calculation. Global Frequency is just a current representation of the Frequency on our website. However, during the calculation we use the frequency number that was the Global Frequency when you purchase the license. For Example, if the Global Frequency when a customer bought their license was "15", this number will be used later when that customer mints their coins, disregarding the current Global Frequency. We call this "Frequency locking", as frequency is locked to the number that was in effect when you purchased the license.

## 5.3 Centralize vs. Calculation Algorithm

Currently we are manually calculating the Frequency based on the network size and number of coins. However, we will soon implement a calculation algorithm which will do this automatically. As described in the "Minting" section, we take the total number of coins and calculate the frequency based on the distribution forecast, so we can control the number of DasCoins that will be available at any given time.

## 5.4 Upgrades

### 5.4.1 Description

Cycles are issued to licensees in generations (lasting 2-3 months). The current generation of Cycles began operations at 100% efficiency. However, the efficiency of Cycles gradually decreases over time due to the impact of Moore's Law and related technology dynamics. The efficiency of the current generation of Cycles is measured against the potential efficiency of the next generation of Cycles. Whenever the efficiency of the current generation of Cycles drops to 50%, an Upgrade occurs which immediately doubles the number of Cycles available to the network. This is why the number of "Upgrades" stated on your software license is so important – the more upgrades you are eligible for, the more times your Cycles have the opportunity to double in number.

Every 108 days all Cycles that haven't been submitted to the Minting Queue are doubled to reflect the growing power of the network. This is a powerful reward, allowing you to benefit as the efficiency of DasNet increases.

## 5.5 Current Cycle implementation

As mentioned before, Cycles are implemented as a cryptographic asset, and they do not have decimals. This means that, in the case that you have Cycles, the lowest value that you can have is 1 Cycle.

## 5.6 Bonus Cycles

With each purchased license, a customer is also rewarded with Bonus Cycles. The amount of added Cycles is 40% of what they receive just from license purchase. This is a part of our promotion event and also our way of saying “thank you” to new people who are helping us to expand the network.

## 5.7 Cycles Issuance authority description

The authority responsible for the issuance of Cycles is called issue_cycles_to_ license. You can find details in the Operations Appendix.

# 6 Nodes

## 6.1 Master Nodes

Master Nodes are exclusively hosted in data centers based on a requirement that access to the server rack is physically secured. They are compatible and connected to other data centers around the world over leased direct lines affording reliable and highly connected bandwidth. This approach gives DasNet control of the entire path between data centers and permits prevention of man-in-the-middle attacks as well as Denial of Service and Distributed Denial of Service attacks among the nodes that maintain the Blockchain and its connectivity to service. DasNet has 2 additional layers for handling transaction capture and network connectivity in addition to the core infrastructure features mentioned previously. The server configuration involves state-of-the-art quality components and protection for high-end threat prevention and hardware-based firewall solutions that are commonly utilized by banks and other highly secure environments. In addition, DasNet is hosted on powerful servers that operate with 44 cores per server which provides efficient space and power consumption to scale into very high traffic and global utilization. DasNet will have a total of 33 Core Master Nodes operating in data centers in 33 different jurisdictions throughout the world. Core Master Node installations will occur at a pace of approximately 2 per month. There will also be approximately 3,000 Core Ledger Nodes running on DasNet.

Please note that at the moment no one can connect their own Nodes to our peer-to-peer network as it is a permission-based network.

The role of the Master Node is to aggregate transactions with the intention to produce Blocks. Only Master Nodes have the authority to write transactions into the Blockchain ledger history. Each Master Node is aware of the others and they must have been voted in by the governing system. Master Nodes are novel in that their authority is represented with cryptographic keys. This means that each Master Node must have registered its Public Key and will sign with its Private Key during the time of Block Production. Therefore, it is possible to hold any one particular Master Node accountable for its actions.

At the moment there are 5 Master Nodes and by the end of 2018 our plan is to have much more.

## 6.2 Ledger Nodes

Ledger Nodes are non-authoritative maintainers of the DasCoin Blockchain. In other words, Ledger Nodes do not produce blocks, yet they aggregate transactions and pass them to the Master Nodes for Block inclusion. Ledger Nodes can verify transactions and are therefore useful for both increasing the footprint of the DasCoin Consensus Network and permitting connectivity to reach farther without requiring the need to assign authority to the nodes. Transaction propagation is accelerated because of Ledger Nodes.

Currently we have 16 Ledger Nodes that are located in the following locations:

- Belgrade
- London
- Oregon
- Taiwan
- Amsterdam

- Frankfurt
- Bangalore
- Sydney
- Brazil

The first five Ledger Nodes are deployed at the Belgrade Datacenter; the London, Oregon and Taiwan Ledger Nodes are located in Google Cloud Datacenters; and the rest are in DigitalOcean and Amazon Datacenters.

Ledger Node is the same node structure as Master Node, except it is not permitted to sign blocks. It does everything else in the same way as Master Node. Block production is the same as in Graphene. Master Node picking is different - there is no voting for Master Nodes. We are a permissioned blockchain and in our system Master Nodes are chosen by 'root authority'. To be able to promote your Ledger Node into a Master Node, you have to run the node as ledger, and on starting ledger you need to provide additional parameters as 'witness-id' and 'private-key'. Then 'root authority' has to issue operations that will make your node behave as a Master Node. Operations related to this functionality are:

- create_witness_operation
- update_witness_operation
- remove_witness_operation
- activate_witness_operation
- deactivate_witness_operation

## 6.3 Node Operations

### 6.3.1 Create Witness Operation

The operation 'create_witness_operation' is used to make witness object in the object database. Those objects are marked with the id '1.6.X' and this id needs to be provided when running Master Node as argument 'witness-id'.

Operation properties are:

- 'fee' - amount of fee to be payed (this is currently a free-of-charge operation)
- 'authority' - this is root account authority.
- 'witness_account' - existing account that we want to promote into a Master Node candidate.
- 'block_signing_key' - public key that is used for signing blocks
- 'url' - optional field used to specify url address of Witness Node
- 'comment' - optional field

### 6.3.2 Update Witness Operation

The operation 'update_witness_operation' is used to change the properties of an already existent witness object.

Operation properties are:

- ‘fee’ - amount of fee to be payed (this is currently a free-of-charge operation)
- ‘witness’ - the id of witness object in database
- ‘authority’ - this is root account authority
- ‘witness_account’ - optional field used to specify an existing account that we want to promote into a Master Node candidate
- ‘block_signing_key’ - optional field used to specify public key that is used for signing blocks
- ‘url’ - optional field used to specify the url address of Witness Node
- ‘comment’ - optional field

### 6.3.3 Remove witness operation

The operation ‘remove_witness_operation’ is used to remove existent witness object from the database.

Operation properties are:

- ‘fee’ - amount of fee to be payed (this is currently a free-of-charge operation)
- ‘witness’ - id of witness object in database that will be deleted
- ‘authority’ - this is root account authority
- ‘comment’ - optional field

### 6.3.4 Activate witness operation

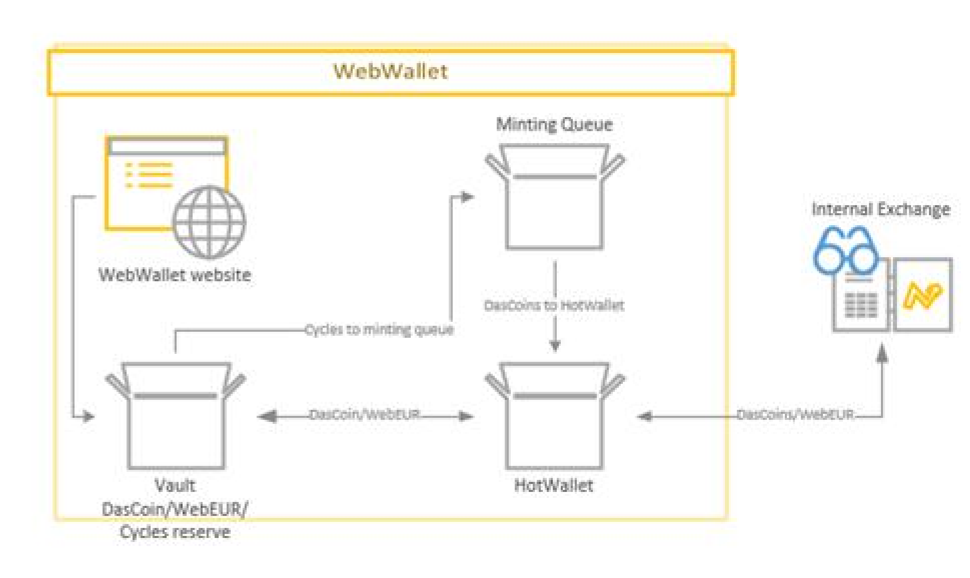
The operation ‘activate_witness_operation’ is used to add an existent witness object to active witness objects. Witness scheduler will reserve a slot for this witness in every signing round, and this witness (node instance) will be able to sign blocks on its slot in time. If the node instance is not running, the slot will be empty, and we will have missing blocks. Example: if we have 3 witness nodes active and only two of them running we will have every third slot in time missing a block.

Operation properties are:

- ‘fee’ - amount of fee to be payed (this is currently a free-of-charge operation)
- ‘witness’ - id of witness object in database that will be activated
- ‘authority’ - this is root account authority.
- ‘comment’ - optional field

# 7 WebWallet

WebWallet is a secure access point for users to access relevant data on the DasCoin Blockchain and interact with the global state by way of signing transactions. It is a cryptographic web-based wallet with the client front end running in the user's browser and the application server backend hosted on an internet web server. The WebWallet serves as the user's point of entry into the secure DasCoin Network. Each WebWallet account links the users to their vaults and wallets and is used to store and validate personal KYC and AML information. Users authenticated through WebWallet can access their relevant data from the global blockchain state – balances from vaults, license purchases, transaction history, etc.




WebWallet consists of a Vault and a HotWallet. The Vault is used for storing all assets that come from the NetLeaders site. Every NetLeaders account has a corresponding Vault. A WebWallet may contain a single or multiple Vaults, depending on how a user registers their account after they received their WebWallet invitation email.

Assets that are stored in the Vault are:

- DasCoin – can be received in your Vault after the minting process, or if you purchased them in the Exchange;
- Reserved WebEuros – your reserve Euro commission earnings from NetLeaders are automatically transferred to your Vault and are converted to the DasExchange currency of WebEuros;
- Cash WebEuros – when you transfer Euros from your NetLeaders balance, they go to your Vault and are converted to the DasExchange currency of WebEuros. You can also accumulate Cash WebEuros from selling DasCoins on the Internal Exchange;
- Cycles – Cycles acquired from licenses that can be manually submitted to the minting Queue from your Vault.

## 7.1 Security

Each Vault Account must be licensed for a person or entity to access DasNet. The license level and corresponding KYC levels also determine the daily amounts a Vault Account is eligible to transfer to a Wallet Account. A higher license level enables the person or entity to increase their level of authentication and therefore increases their access to more capacity within the system and higher withdrawal privileges. This way the DasCoin Blockchain and DasNet can be fully compliant with global regulations that require account holders to be identified and in good standing before engaging in commerce with other participants of DasNet. This type of authentication protocol results in a higher level of integrity among participants and is likely to lead to more acceptance within regulated jurisdictions throughout the world.

## 7.2 Validator

WebWallet relies on a cryptographic hardware store for managing keys and securely signing blockchain transactions. This proprietary hardware device is known as The Validator and has the ability to generate and store ECDSA private keys corresponding to the users' blockchain vaults. The Validator is also secured with a PIN/passphrase that prevents misuse and theft of keys. When signing blockchain transactions, WebWallet interfaces with The Validator hardware device. The user must first unlock their Validator before the required private ECDSA key can be transferred into the browser memory of the javascript client application upon confirmation of a transaction. No DasCoin transaction can be made without validation through this cryptographic hardware device. This system provides the highest level of safety and security in the authentication of transactions. The key is stored in the memory for the shortest possible time required to sign the transaction upon which it is purged from the client memory. The private key is never 'hot' – it never crosses the wire, not even in encrypted form.

You will need your Validator to:

- Withdraw Euros from your WebWallet account
- Send DasCoin to another WebWallet account
- Trade DasCoin via DasExchange
- Access your Reserve funds
- Set up your HotWallet and transfer funds to it

### 7.2.1 Account recovery

The user can back up a mnemonic consisting of 24 words used as entropy for generating the private key. In case of theft or loss of the hardware device, the user can use the mnemonic to reconstruct the original private key, thereby restoring access to their account.

## 7.3 Identity management

### 7.3.1 KYC

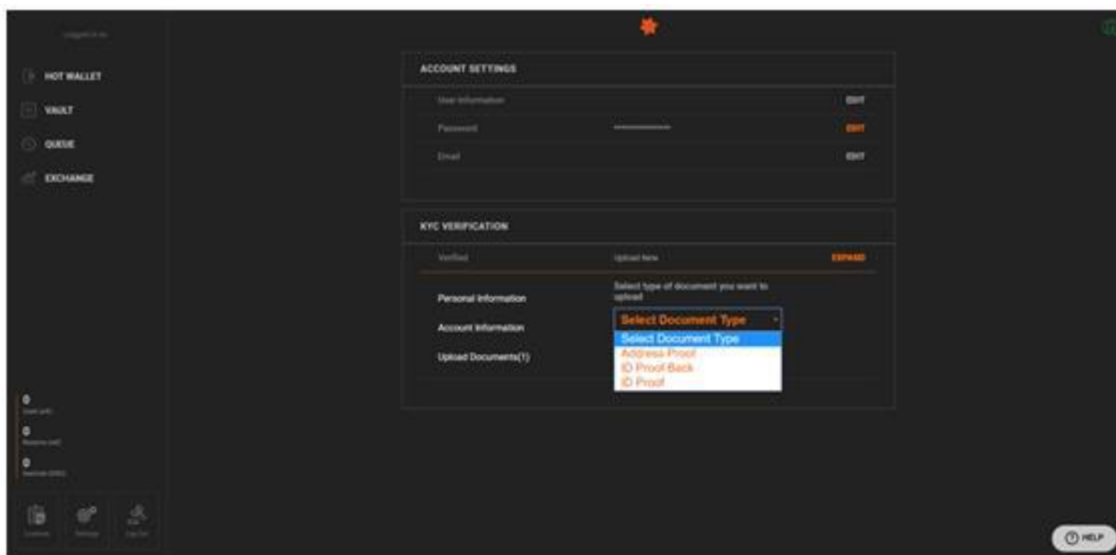
AML stands for Anti-Money Laundering. It is a phrase that applies to the procedures, laws and regulations designed to stop the practice of generating income through illegal actions. At DasNet, we verify your identity using bank standard "Know Your Customer" protocols that help safeguard DasCoin's position as the Currency of Trust.

Depending on your country of residence, the KYC verification process typically takes around 10 minutes. In certain cases, however, it can take up to 48h. During the process, you may be required to upload additional documents to verify your identity. If additional documents are required, we will contact you via email with instructions. We will also provide status updates during the process if needed. You can upload government-issued documents such as personal ID cards, driver's licenses and passports. As proof of address, you can upload government-issued documents on which your residential address is visible, or a credit card/bank statement less than three month old.

Before you can really enjoy all the benefits the platform provides, KYC verification is a mandatory step. Without KYC, customer will not receive access to WebWallet.

For successful KYC verification, you need to upload 3 document types:

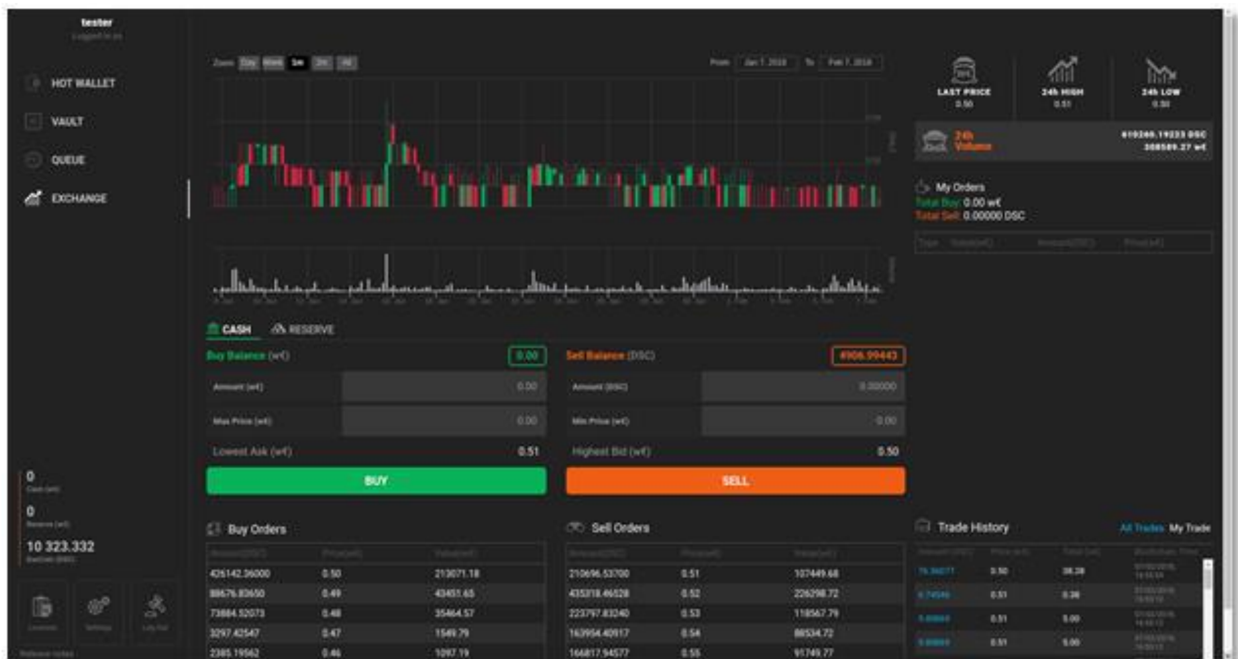
- Proof of address
- ID proof – Front
- ID Proof – Back



## 7.4 Sample UI

Here is a quick preview of the WebWallet interface, the structure of WebWallet consist of:

- The WebWallet website – a GUI representation of customers' wallets, as well as easy access to our DasExchange
- Vault – used for storing all of your assets coming from the NetLeaders website
- HotWallet – where you hold funds to trade and withdraw
- Minting Queue – used to mint DasCoins from Cycles
- Internal Exchange or DasExchange




One of the main parts of the WebWallet platform is our Internal Exchange where customers can sell or buy their DasCoins.

DasExchange is extremely easy to use and contains all the information that customer might need - from an overview of value fluctuation that can be set to daily, weekly, or monthly periods (or to a period extending from the moment our platform went live to the present day) to information like Trade History, Buy Orders and Sell Orders.

# 8 API

For a full list of API calls and details please see the Appendix or visit our regularly updated API website here: https://docs.dascoin.com

## 8.1 Websocket Calls & Notifications

**Call Format**

In DasCoin, Websocket calls are stateful and accessible via regular websocket connection. The JSON call has the following structure:

```
{
  "id":1,
  "method":"call",
  "params":[0, "get_accounts", [["1.2.0"]]]
}
```

The parameters params have the following structure:

```
[API-identifier, Method-to-Call, Call-Parameters]
```

In the example above, we query the database API which carries the identifier 0 in our example.

**Example Call With wscat**

Note: Examples in this section assume that you have a full node running and listening to port 9880, locally.

The following will show the usage of websocket connections. We make use of the wscat application available via npm:

```
npm install -g wscat
```

A non-restricted call against a full-node would take the form:

```
wscat -c ws://127.0.0.1:9880
> {"id":1, "method":"call", "params":[0, "get_accounts", [["1.2.0"]]]}
```

**Successful Calls**

The API will return a JSON formatted response carrying the same id as the request to distinguish subsequent calls.

```
{
  "id":1,
  "result":  ..data..
}
```

**Errors**

In case of an error, the resulting answer will carry an error attribute and a detailed description:

```
{
  "id":0,
  "ssid": ""
  "error": {
    "data": {
      "code": error-code,
      "name": " .. name of exception .."
      "message": " .. message of exception ..",
      "stack": [ .. stack trace .. ],
    },
    "code": 1,
  },
}
```

### 8.1.1 Requesting API access

The DasCoin full node offers a wide range of APIs that can be accessed via websockets. The procedure works as follows:

1. Login to the Full Node
2. Request access to an API
3. Obtain the API identifier
4. Call methods of a specific API by providing the identifier

Please find a list of available APIs below:

- Database API
- Account History API
- Network Broadcast API
- Network Nodes API

**Login**

The first thing we need to do is to log in:

```
> {"id":2,"method":"call", "params":[1, "login", ["",""]]}
< {"id":2,"result":true}
```

If you have restricted access, then you may be required to put your username and password into quotes, accordingly. Furthermore, you should verify, that the result gives positive confirmation about your login.

**Requesting Access to an API**

Most data can be queried from the Database API to which we register with the following call::

```
> {"id":2,"method":"call", "params":[1, "database",[]]}
```

**Obtain the API identifier**

After requesting access, the full node will either deny access or return an identifier to be used in future calls:

```
  < {"id":2,"result":2}
```

The result will be our identifier for the database API, in the following called DATABASE_API_ID!

**Call methods of a specific API by providing the identifier**

Now we can call any methods available to the database API via::

```
  > {"id":1,"method":"call", "params":[DATABASE_API_ID,"get_accounts",
[["1.2.0"]]]}
```

### 8.1.2 Database Notifications

In DasCoin, the websocket connection is used for notifications when objects in the database change or a particular event (such as filled orders) occurs.

**Available subscriptions**

***set_subscribe_callback***

```
set_subscribe_callback( int identifier, bool clear_filter )
```

To simplify development, a global subscription callback can be registered. Every notification initiated by the full node will carry a particular id as defined by the user with the identifier parameter.

***set_pending_transaction_callback***

```
set_pending_transaction_callback(int identifier)
```

Notifications for incoming unconfirmed transactions.

***set_block_applied_callback***

```
set_block_applied_callback(block_id)
```

Gives a notification whenever the block block_id is applied to the blockchain.

***subscribe_to_market***

```
subscribe_to_market(int identifier, asset_id a, asset_id b))
```

Subscribes to market changes in market a:b and sends notifications with id identifier.

***get_full_accounts***

```
get_full_accounts(array account_ids, bool subscribe)
```

Returns the full account object for the accounts in array account_ids and subscribes to changed to that account if subscribe is set to True.

**Subscription Format**

Let's first get a global subscription callback to distinguish our notifications from regular RPC calls::

```
  >
{"id":4,"method":"call","params":[DATABASE_API_ID,"set_subscribe_callback",
[SUBSCRIPTION_ID, true]]}
```

This call above will register SUBSCRIPTION_ID as the id for notifications. Now, whenever you get an object from the witness (e.g. via get_objects) you will automatically subscribe to any future changes of that object. After calling set_subscribe_callback the witness will start to send notices every time the object changes::

```
  < {
      "method": "notice"
      "params": [
        SUBSCRIPTION_ID,
        [[
          { "id": "2.1.0", ...  },
          { "id": ...  },
          { "id": ...  },
          { "id": ...  }
        ]]
      ],
    }
```

### 8.1.3 Example Session

Here is an example of a full session::

```
  > {"id":2,"method":"call","params":[1,"login",["",""]]}
  < {"id":2,"result":true}
  > {"id":3,"method":"call","params":[1,"database",[]]}
  < {"id":3,"result":2}
  > {"id":4,"method":"call","params":[1,"history",[]]}
  < {"id":4,"result":3}
  > {"id":5,"method":"call","params":[2,"set_subscribe_callback",[5,false]]}
  < {"id":5,"result":null}
  > {"id":6,"method":"call","params":[2,"get_objects",[["2.1.0"]]]}
```

(plenty of data coming in from this point on)

For the complete list of DasCoin API calls see *Appendix A.*

## 8.2 Operations

For a full list of Operations and details please see *Appendix B*.

# 9 Appendix A: Blockchain API

## 9.1 Database API

The database API is available from the full node via websockets. If you have not set up your websockets connection, please read this article.

#### 9.1.1.1 OBJECTS

fc::variants graphene::app::database_api::**get_objects**(*const* vector<object_id_type> &*ids*) *const*
Get the objects corresponding to the provided IDs. If any of the provided IDs does not map to an object, a null variant is returned in its position.

**Return**
The objects retrieved, in the order they are mentioned in ids
**Parameters**
ids: IDs of the objects to retrieve
If any of the provided IDs does not map to an object, a null variant is returned in its position.

#### 9.1.1.2 SUBSCRIPTIONS

void graphene::app::database_api::**set_subscribe_callback**(std::function<void(*const* variant&)> *cb*,
bool notify_remove_create)

void graphene::app::database_api::**set_pending_transaction_callback**(std::function<void(*const* variant&)> *cb*)

void graphene::app::database_api::**set_block_applied_callback**(std::function<void(*const* variant &block_id)> *cb*)

void graphene::app::database_api::**cancel_all_subscriptions**()

Stop receiving any notifications.
This unsubscribes from all subscribed markets and objects.

#### 9.1.1.3 BLOCKS AND TRANSACTIONS

optional<block_header> graphene::app::database_api::**get_block_header**(uint32_t *block_num*) *const*
Retrieve a block header.
**Return**
Header of the referenced block, or null if no matching block was found
**Parameters**
block_num: Height of the block whose header should be returned

---

optional<signed_block> graphene::app::database_api::**get_block**(uint32_t *block_num*) *const*
Retrieve a full, signed block.
**Return**
The referenced block, or null if no matching block was found
**Parameters**
block_num: Height of the block to be returned

---

vector< signed_block_with_num > graphene::app::database_api::**get_blocks** ( uint32_t *start_block_num*, uint32_t *count*) *const*

**Return**
An array of full, signed blocks starting from a specified height.
**Parameters**
start_block_num: Height of the block to be returned
count: Number of blocks to return

---

vector< signed_block_with_virtual_operations_and_num > graphene::app::database_api::**get_blocks_with_virtual_operations** ( uint32_t *start_block_num*, uint32_t *count*, std::vector< uint16_t > *virtual_operation_ids* ) *const*

Return an array of full, signed blocks that contains virtual operations starting from a specified height.

**Return**
Array of enumerated blocks
**Parameters**
start_block_num: Height of the starting block.
count: Number of blocks to return
virtual_operation_ids: Array of virtual operation ids that should be included in result returned

---

processed_transaction graphene::app::database_api::**get_transaction**(uint32_t *block_num*, uint32_t *trx_in_block*) *const*

Used to fetch an individual transaction.

---

optional<signed_transaction> graphene::app::database_api::**get_recent_transaction_by_id**( *const* transaction_id_type &*id*) *const*

If the transaction has not expired, this method will return the transaction for the given ID or it will return NULL if it is not known. Just because it is not known does not mean it wasn't included in the blockchain.

---

#### 9.1.1.4 GLOBALS

---

chain_property_object graphene::app::database_api::**get_chain_properties**() *const*

Retrieve the chain_property_object associated with the chain.

global_property_object graphene::app::database_api::**get_global_properties**() *const*

Retrieve the current global_property_object.

fc::variant_object graphene::app::database_api::**get_config**() *const*

Retrieve compile-time constants.

chain_id_type graphene::app::database_api::**get_chain_id**() *const*

Get the chain ID.

dynamic_global_property_object
graphene::app::database_api::**get_dynamic_global_properties**() *const*

Retrieve the current dynamic_global_property_object.

#### 9.1.1.5 KEYS

vector<vector<account_id_type>>
graphene::app::database_api::**get_key_references**(vector<public_key_type> *key*) *const*

#### 9.1.1.6 ACCOUNTS

vector<optional<account_object>> graphene::app::database_api::**get_accounts**(
*const* vector<account_id_type> &*account_ids*) *const*

Get a list of accounts by ID. This function has semantics identical to get_objects.

**Return**
The accounts corresponding to the provided IDs
**Parameters**
account_ids: IDs of the accounts to retrieve

std::map<string, full_account> graphene::app::database_api::**get_full_accounts**(
*const* vector<string> &*names_or_ids*, bool *subscribe*)

Fetch all objects relevant to the specified accounts and subscribe to updates. This function fetches all relevant objects for the given accounts, and subscribes to updates to the given accounts. If any of the strings in names_or_ids cannot be tied to an account, that input will be ignored. All other accounts will be retrieved and subscribed.

**Return**
Map of string from names_or_ids to the corresponding account

**Parameters**

callback: Function to call with updates

names_or_ids: Each item must be the name or ID of an account to retrieve

---

optional<account_object> graphene::app::database_api::**get_account_by_name**(string *name*) *const*

---

vector<account_id_type> graphene::app::database_api::**get_account_references**( account_id_type *account_id*) *const*

**Return**

all accounts that refer to the key or account id in their owner or active authorities.

---

vector<optional<account_object>> graphene::app::database_api::**lookup_account_names**( *const* vector<string> &*account_names*) *const*

Get a list of accounts by name. This function has semantics identical to get_objects.

**Return**

The accounts holding the provided names

**Parameters**

account_names: Names of the accounts to retrieve

---

map<string, account_id_type> graphene::app::database_api::**lookup_accounts**( *const* string &*lower_bound_name*, uint32_t *limit*) *const*

Get names and IDs for registered accounts.

**Return**

Map of account names to corresponding IDs

**Parameters**

lower_bound_name: Lower bound of the first name to return

limit: Maximum number of results to return must not exceed 1000

---

uint64_t graphene::app::database_api::**get_account_count**() *const*

Get the total number of accounts registered with the blockchain.

---

optional< vault_info_res > graphene::app::database_api::**get_vault_info**(account_id_type *vault_id*) *const*

Get vault information.

**Return**

vault_info_res (optional)

**Parameters**

vault_id

---

vector< acc_id_vault_info_res > graphene::app::database_api::**get_vaults_info**(

vector<account_id_type> *vault_ids*) *const*

Get vault information for a list of vaults.

**Return**
A JSON object containing a vault id and optional vault information (if vault exists).
**Parameters**
vault_ids: A list of vault ID's.

---

### 9.1.1.7 BALANCES

---

vector<asset> graphene::app::database_api::**get_account_balances**( account_id_type *id*, *const* flat_set<asset_id_type> &*assets*) *const*

Get an account's balances in various assets.

**Return**
Balances of the account
**Parameters**
id: ID of the account to get balances for
assets: IDs of the assets to get balances of; if empty, get all assets account has a balance in

---

vector<asset> graphene::app::database_api::**get_named_account_balances**( *const* std::string &*name*, *const*flat_set<asset_id_type> &*assets*) *const*

Semantically equivalent to get_account_balances, but takes a name instead of an ID.

---

vector<balance_object> graphene::app::database_api::**get_balance_objects**(*const* vector<address> &*addrs*) *const*

**Return**
all unclaimed balance objects for a set of addresses

---

vector<asset> graphene::app::database_api::**get_vested_balances**( *const* vector<balance_id_type> &*objs*) *const*

---

vector<vesting_balance_object> graphene::app::database_api::**get_vesting_balances**( account_id_type *account_id*) *const*

---

acc_id_share_t_res graphene::app::database_api::**get_free_cycle_balance**(account_id_type *account_id*) *const*

(Deprecated) Get a free cycle amount for account

**Return**

Number of issued free cycles on account
**Parameters**
account_id**:** account_ids ID of the account to retrieve

---

vector<acc_id_share_t_res>
graphene::app::database_api::**get_free_cycle_balances_for_accounts**(
vector<account_id_type> *ids*) *const*

(Deprecated) Get remaining amount of cycles.

**Return**
Vector of objects containing account id and cycle balance
**Parameters**
ids**:** Vector of account ids

---

acc_id_vec_cycle_agreement_res graphene::app::database_api::**get_all_cycle_balances**(
account_id_type *account_id*) *const*

(Deprecated) Get cycle amounts per cycle agreement for account

**Return**
Vector of cycle amounts and frequency locks on account
**Parameters**
account_id: ID of the account to retrieve

---

vector< acc_id_vec_cycle_agreement_res >
graphene::app::database_api::**get_all_cycle_balances_for_accounts**(
vector< account_id_type > *ids*) *const*

(Deprecated) Get cycle balances for list of accounts.

**Return**
Vector of objects containing account id, cycle amount and frequency lock
**Parameters**
ids**:** Vector of account ids

---

signed_transaction graphene::app::database_api::**purchase_cycle_asset**(string account, string amount_to_sell, string symbol_to_sell, double frequency, double amount_of_cycles_to_receive, bool broadcast = false)

Purchase cycles

**Parameters**
account: account name or id
amount_to_sell: amount of asset to sell
symbol_to_sell: symbol of asset to sell
frequency: frequency at which we buy

amount_of_cycles_to_receive: amount of cycles to receive by this buy

---

optional<cycle_price> graphene::wallet::wallet_api::**calculate_cycle_price**(share_type cycle_amount, asset_id_type asset_id) *const*

Calculates and returns the amount of asset one needs to pay to get the given amount of cycles

**Return**
cycle_price structure (optional)
**Parameters**
cycle_amount: Desired amount of cycles to get
asset_id_type: Asset to pay

---

acc_id_share_t_res graphene::app::database_api::**get_dascoin_balance**( account_id_type *id*) *const*

Get amount of DASCoin for on an account.

**Return**
An object containing id and balance of an account
**Parameters**
id: ID of the account to retrieve

---

#### *9.1.1.8 REWARD QUEUE*

vector< reward_queue_object > graphene::app::database_api::**get_reward_queue**() *const*

Return the entire reward queue.
**Return**
Vector of all reward queue objects.

---

vector< reward_queue_object > graphene::app::database_api::**get_reward_queue_by_page**( Uint32_t from, uint32_t amount) *const*

Return a portion of the reward queue.
**Parameters**
from: Starting page
amount: Number of pages to get
**Returns**
Vector which represent a portion of that queue

---

unit32_t graphene::app::database_api::**get_reward_queue_size**() *const*

Get the size of the DASCoin reward queue.
**Return**
Number of elements in the DASCoin queue.

---

acc_id_queue_subs_w_pos_res graphene::app::database_api::**get_queue_submissions_with_pos**( account_id_type account_id) *const*

Get all current submissions to reward queue by single account.
**Parameters**
account_id: id of account whose submissions should be returned
**Return**
All elements on DASCoin reward queue submitted by given account

---

vector<acc_id_queue_subs_w_pos_res> graphene::app::database_api::**get_queue_submissions_with_pos_for_accounts** (vector<account_id_type> ids) *const*

Get all current submissions to reward queue by multiple account.
**Parameters**
ids: vector of account ids
**Return**
All elements on DASCoin reward queue submitted by given accounts

---

#### *9.1.1.9 REQUESTS*

vector< issue_asset_request_object > graphene::app::database_api::**get_all_webasset_issue_requests**() *const*

Get all webasset issue request objects, sorted by expiration.
**Return**
Vector of webasset issue request objects.

---

vector< wire_out_holder_object > graphene::app::database_api::**get_all_wire_out_holders**() *const*

Get all wire out holder objects.
**Return**
Vector of wire out holder objects.

---

vector< wire_out_with_fee_holder_object > graphene::app::database_api::**get_all_wire_out_with_fee_holders**() *const*

Get all wire out with fee holder objects.
**Return**
Vector of wire out with fee holder objects.

---

#### *9.1.1.10 ASSETS*

vector<optional<asset_object>> graphene::app::database_api::**get_assets**(

*const* vector<asset_id_type> &*asset_ids*) *const*

Get a list of assets by ID.
This function has semantics identical to get_objects.
**Return**
The assets corresponding to the provided IDs
**Parameters**
asset_ids: IDs of the assets to retrieve

---

vector<asset_object> graphene::app::database_api::**list_assets**(
*const* string &*lower_bound_symbol*, uint32_t *limit*) *const*
Get assets alphabetically by symbol name.
**Return**
The assets found
**Parameters**
lower_bound_symbol: Lower bound of symbol names to retrieve
limit: Maximum number of assets to fetch (must not exceed 100)

---

vector<optional<asset_object>> graphene::app::database_api::**lookup_asset_symbols**(
*const* vector<string> &*symbols_or_ids*) *const*
Get a list of assets by symbol. This function has semantics identical to get_objects.
**Return**
The assets corresponding to the provided symbols or IDs
**Parameters**
asset_symbols: Symbols or stringified IDs of the assets to retrieve

---

optional< asset_object > graphene::app::database_api::**lookup_asset_symbol** (
const string &symbols_or_id) const

Get an asset by symbol.

**Return**
The asset corresponding to the provided symbol or ID
**Parameters**
asset_symbols: Symbols or stringified IDs of the assets to retrieve

---

uint64_t graphene::app::database_api::**get_account_count**() *const*

Get the total number of accounts registered with the blockchain.

---

bool graphene::app::database_api::**check_issued_asset**(
const string &*unique_id*, const string &*asset*) const

Check if an asset issue with the corresponding unique was completed on the chain.

---

bool graphene::app::database_api::check_issued_webeur(const string &*unique_id*) const

Check if a webeur issue with the corresponding unique was completed on the chain.

### *9.1.1.11 MARKETS / FEEDS*

order_book graphene::app::database_api::**get_order_book**(
*const* string &*base*, *const* string &*quote*, unsigned *limit* = 50) *const*

Returns the order book for the market base:quote.

**Return**
Order book of the market
**Parameters**

- base: String name of the first asset
- quote: String name of the second asset
- depth: of the order book. Up to depth of each asks and bids, capped at 50. Prioritizes most moderate of each

---

vector<limit_order_object> graphene::app::database_api::**get_limit_orders**(
asset_id_type *a*, asset_id_type *b*, uint32_t *limit*) *const*

Get limit orders in a given market.

**Return**
The limit orders, ordered from least price to greatest
**Parameters**

- a: ID of asset being sold
- b: ID of asset being purchased
- limit: Maximum number of orders to retrieve

---

vector<limit_order_object> graphene::app::database_api::**get_limit_orders**(
asset_id_type *a*, asset_id_type *b*, uint32_t *limit*) *const*

Get limit orders in a given market.

**Return**
The limit orders, ordered from least price to greatest
**Parameters**

- a: ID of asset being sold
- b: ID of asset being purchased
- limit: Maximum number of orders to retrieve

---

vector<limit_order_object> graphene::app::database_api::**get_limit_orders_for_account**(
asset_id_type *a*, asset_id_type *b*, uint32_t *limit*) *const*

Get limit orders in a given market.

**Return**

The limit orders, ordered from least price to greatest

**Parameters**

id: ID of the account to get limit orders for

- a: ID of asset being sold
- b: ID of asset being purchased
- limit: Maximum number of orders to retrieve

---

limit_orders_grouped_by_price
graphene::app::database_api::**get_limit_orders_grouped_by_price**(
asset_id_type *a*, asset_id_type *b,* uint32_t *limit*) *const*

Get limit orders in a given market grouped by price and divided in buy and sell vectors.

**Return**

The call orders, ordered from earliest to be called to latest

**Parameters**

- a: ID of asset being sold
- b: ID of asset being purchased
- limit: Maximum number of orders to retrieve

---

vector<force_settlement_object> graphene::app::database_api::**get_settle_orders**(
asset_id_type *a*, uint32_t *limit*) *const*

Get forced settlement orders in a given asset.

**Return**

The settle orders, ordered from earliest settlement date to latest

**Parameters**

- a: ID of asset being settled
- limit: Maximum number of orders to retrieve

---

vector<call_order_object> graphene::app::database_api::**get_margin_positions**(
*const* account_id_type &*id*) *const*

**Return**

all open margin positions for a given account id.

---

void graphene::app::database_api::**subscribe_to_market**(
std::function<void(*const* variant&)> *callback*, asset_id_type *a*, asset_id_type *b*, )

Request notification when the active orders in the market between two assets changes. Callback will be passed a variant containing a vector<pair<operation, operation_result>>. The vector will contain, in order, the operations which changed the market, and their results.

**Parameters**

- callback: Callback method which is called when the market changes
- a: First asset ID
- b: Second asset ID

---

void graphene::app::database_api::**unsubscribe_from_market**(asset_id_type *a*, asset_id_type *b*)

Unsubscribe from updates to a given market.

**Parameters**

- a: First asset ID
- b: Second asset ID

---

market_ticker graphene::app::database_api::**get_ticker**(*const* string &*base*, *const* string &*quote*) *const*

Returns the ticker for the market assetA:assetB.

**Return**

The market ticker for the past 24 hours.

**Parameters**

- a: String name of the first asset
- b: String name of the second asset

---

market_hi_lo_volume graphene::app::database_api::**get_24_hi_lo_volume**( *const* string &*base*, *const* string &*quote*) *const*

Returns the 24 hour high, low and volume for the market assetA:assetB.

**Return**

The market high, low and volume over the past 24 hours

**Parameters**

- a: String name of the first asset
- b: String name of the second asset

---

vector<market_trade> graphene::app::database_api::**get_trade_history**( *const* string &*base*, *const* string &*quote*, fc::time_point_sec *start*, fc::time_point_sec *stop*, unsigned *limit* = 100) *const*

Returns recent trades for the market assetA:assetB Note: Currentlt, timezone offsets are not supported. The time must be UTC.

**Return**

Recent transactions in the market

**Parameters**

- a: String name of the first asset
- b: String name of the second asset
- stop: Stop time as a UNIX timestamp
- limit: Number of transactions to retrieve, capped at 100
- start: Start time as a UNIX timestamp

---

vector<market_trade> graphene::app::database_api::**get_trade_history_by_sequence**( *const* string &*base*, *const* string &*quote*, fc::time_point_sec *start*, fc::time_point_sec *stop*, unsigned *limit* = 100) *const*

Returns recent trades for the market assetA:assetB Note: Currentlt, timezone offsets are not supported. The time must be UTC.

**Return**

Recent transactions in the market

**Parameters**

- a: String name of the first asset
- b: String name of the second asset
- stop: Stop time as a UNIX timestamp

- start: Start sequence as an Integer, the latest trade to retrieve
- limit: Number of transactions to retrieve, capped at 100

---

### 9.1.1.12 WITNESSES

vector<optional<witness_object>> graphene::app::database_api::**get_witnesses**(
*const* vector<witness_id_type> &*witness_ids*) *const*
Get a list of witnesses by ID. This function has semantics identical to get_objects
**Return**
The witnesses corresponding to the provided IDs
**Parameters**

- witness_ids: IDs of the witnesses to retrieve

---

fc::optional<witness_object> graphene::app::database_api::**get_witness_by_account**(
account_id_type *account*) *const*
Get the witness owned by a given account.
**Return**
The witness object, or null if the account does not have a witness
**Parameters**

- account: The ID of the account whose witness should be retrieved

---

map<string, witness_id_type> graphene::app::database_api::**lookup_witness_accounts**(
*const* string &*lower_bound_name*, uint32_t *limit*) *const*
Get names and IDs for registered witnesses.
**Return**
Map of witness names to corresponding IDs
**Parameters**

- lower_bound_name: Lower bound of the first name to return
- limit: Maximum number of results to return must not exceed 1000

---

uint64_t graphene::app::database_api::**get_witness_count**() *const*

Get the total number of witnesses registered with the blockchain.

### 9.1.1.13 COMMITTEE MEMBERS

vector<optional<committee_member_object>>
graphene::app::database_api::**get_committee_members**(
*const* vector<committee_member_id_type> &*committee_member_ids*) *const*
Get a list of committee_members by ID. This function has semantics identical to get_objects
**Return**
The committee_members corresponding to the provided IDs
**Parameters**

committee_member_ids: IDs of the committee_members to retrieve

---

fc::optional<committee_member_object>
graphene::app::database_api::**get_committee_member_by_account**(

account_id_type *account*) *const*
Get the committee_member owned by a given account.
**Return**
The committee_member object, or null if the account does not have a committee_member
**Parameters**
account: The ID of the account whose committee_member should be retrieved

---

map<string, committee_member_id_type>
graphene::app::database_api::**lookup_committee_member_accounts**(
*const* string &*lower_bound_name*, uint32_t *limit*) *const*
Get names and IDs for registered committee_members.
**Return**
Map of committee_member names to corresponding IDs
**Parameters**
lower_bound_name: Lower bound of the first name to return
limit: Maximum number of results to return must not exceed 1000

#### 9.1.1.14 WORKERS

vector<worker_object> graphene::app::database_api::**get_workers_by_account**(
account_id_type *account*) *const*

Return the worker objects associated with this account.

---

#### 9.1.1.15 VOTES

vector<variant> graphene::app::database_api::**lookup_vote_ids**(*const* vector<vote_id_type> &*votes*) *const*

Given a set of votes, return the objects they are voting for.
This will be a mixture of committee_member_object, witness_objects, and worker_objects.
The results will be in the same order as the votes. Null will be returned for any vote ids that are not found.

#### 9.1.1.16 AUTHORITY / VALIDATION

std::string graphene::app::database_api::**get_transaction_hex**(*const* signed_transaction &*trx*) *const*
Get a hexdump of the serialized binary form of a transaction.

---

set<public_key_type> graphene::app::database_api::**get_required_signatures**(*const* signed_transaction &*trx*, *const*flat_set<public_key_type> &*available_keys*) *const*
This API will take a partially signed transaction and a set of public keys that the owner has the ability to sign for and return the minimal subset of public keys that should add signatures to the transaction.

---

set<public_key_type> graphene::app::database_api::**get_potential_signatures**(*const* signed_transaction &*trx*) *const*

This method will return the set of all public keys that could possibly sign for a given transaction. This call can be used by wallets to filter their set of public keys to just the relevant subset prior to calling get_required_signatures to get the minimum subset.

---

set<address> graphene::app::database_api::**get_potential_address_signatures**(*const* signed_transaction &*trx*) *const*

---

bool graphene::app::database_api::**verify_authority**(*const* signed_transaction &*trx*) *const*
**Return**
true of the trx has all of the required signatures, otherwise throws an exception

---

bool graphene::app::database_api::**verify_account_authority**(
*const* string &*name_or_id*, *const* flat_set<public_key_type> &*signers*) *const*

**Return**
true if the signers have enough authority to authorize an account

---

processed_transaction graphene::app::database_api::**validate_transaction**(*const* signed_transaction &*trx*) *const*

Validates a transaction against the current state without broadcasting it on the network.

---

vector<fc::variant> graphene::app::database_api::**get_required_fees**(*const* vector<operation> &*ops*, asset_id_type *id*)*const*

For each operation calculate the required fee in the specified asset type. If the asset type does not have a valid core_exchange_rate.

---

###### *9.1.1.17 PROPOSED TRANSACTIONS*

vector<proposal_object> graphene::app::database_api::**get_proposed_transactions**( account_id_type *id*) *const*

**Return**
the set of proposed transactions relevant to the specified account id.

---

###### *9.1.1.18 BLINDED BALANCES*

vector<blinded_balance_object> graphene::app::database_api::**get_blinded_balances**(
*const* flat_set<commitment_type> &*commitments*) *const*

**Return**
the set of blinded balance objects by commitment ID

---

###### *9.1.1.19 LICENSES*

optional<license_type_object> graphene::app::database_api::**get_license_type**(

*license_type_id_type*) *const*

Get license type-ids found on the blockchain.

**Returns**
The license type-id if found.
**Parameters**
license_type_id_type-id: used on the block

---

vector<license_type_object> graphene::app::database_api::**get_license_types**() *const*

Get all license type-ids found on the blockchain.

**Returns**
Vector of license type-ids found

---

vector<pair<string, license_type_id_type>>
graphene::app::database_api::**get_license_type_names_ids**() const

Get all name/license type-ids found on the blockchain.

**Returns**
Vector of license name/type-ids pairs found

---

vector<license_types_grouped_by_kind_res>
graphene::app::database_api::**get_license_type_names_ids_grouped_by_kind**() *const*

Get all license type-ids grouped by kind found on the blockchain.

**Returns**
Vector of license type-ids found grouped by kind

---

---

vector<license_objects_grouped_by_kind_res>
graphene::app::database_api::**get_license_objects_grouped_by_kind**() *const*

Get all license objects grouped by kind found on the blockchain.

**Returns**
Vector of license objects found grouped by kind

---

vector<license_type_object>
graphene::app::database_api::**list_license_types_by_name**(
*const* string& *lower_bound_name*, uint32_t *limit*) *const*

Get license types active on the blockchain by name.

**Returns**
The license types found

**Parameters**

lower_bound_symbol: Lower bound of license type names to retrieve
limit: Maximum number of license types to fetch (must not exceed 100)

---

vector<license_type_object> graphene::app::database_api::**list_license_types_by_amount**( *const* uint32_t *lower_bound_amount*, uint32_t *limit*) *const*

Get license types active on the blockchain by amount.

**Returns**
The license types found.
**Parameters**

lower_bound_symbol: Lower bound of license type names to retrieve.
limit: Maximum number of license types to fetch (must not exceed 100).

---

vector<optional< license_type_object>>
graphene::app::database_api::**lookup_license_type_names**(
*const* vector<string> &*names_or_ids*) *const*

Get a list of license types by names. This function has semantics identical to get_objects

**Returns**
The assets corresponding to the provided symbols or IDs
**Parameters**

asset_symbols: Symbols or stringified IDs of the assets to retrieve

---

vector<optional<license_information_object>>
graphene::app::database_api::**get_license_information**(
*const* vector<account_id_type> &*account_ids*) *const*

Get a list of account issued license types. This function has semantics identical to get_objects.

**Returns**
Vector of issued license information objects
**Parameters**

account_ids: IDs of the accounts to retrieve

---

---

vector<upgrade_event_object> graphene::app::database_api::**get_upgrade_events**() *const*

Get a list of upgrade events.

**Returns**
A list of upgrade events, scheduled or executed

---

## 9.2 Account History API

The history_api class implements the RPC API for account history.

#### 9.2.1.1 Account History

vector<operation_history_object> graphene::app::history_api::**get_account_history** (account_id_type *account*, operation_history_id_type *stop* = operation_history_id_type (), unsigned *limit* = 100, operation_history_id_type *start* = operation_history_id_type ()) *const*

Get operations relevant to the specified account.

**Return**

A list of operations performed by account, ordered from most recent to oldest.

**Parameters**

- account: The account whose history should be queried
- stop: ID of the earliest operation to retrieve
- limit: Maximum number of operations to retrieve (must not exceed 100)
- start: ID of the most recent operation to retrieve

---

vector<operation_history_object>
graphene::app::history_api::**get_account_history_by_operation** (account_id_type *account*, flat_set< uint32_t > operation_types, operation_history_id_type *stop* = operation_history_id_type (), unsigned *limit* = 100, operation_history_id_type *start* = operation_history_id_type ()) *const*

Get operations relevant to the specified account filtering by operation type.

**Return**

A list of operations performed by account, ordered from most recent to oldest.

**Parameters**

- account: The account whose history should be queried
- operation_types: The IDs of the operation we want to get operations in the account( 0 = transfer , 1 = limit order create, ...)
- stop: ID of the earliest operation to retrieve
- limit: Maximum number of operations to retrieve (must not exceed 100)
- start: ID of the most recent operation to retrieve

---

vector< operation_history_object > graphene::app::history_api::**get_relative_account_history** (account_id_type *account*, uint32_t *stop* = 0, unsigned *limit* = 100,uint32_t *start* = 0) const

Get operations relevant to the specified account referenced by an event numbering specific to the account. The current number of operations for the account can be found in the account statistics (or use 0 for start).

**Return**

A list of operations performed by account, ordered from most recent to oldest.

**Parameters**

- account: The account whose history should be queried
- stop: Sequence number of earliest operation. 0 is default and will query 'limit' number of operations.
- limit: Maximum number of operations to retrieve (must not exceed 100)
- start: Sequence number of the most recent operation to retrieve. 0 is default, which will start querying from the most recent operation.

#### 9.2.1.2 Market History

vector<order_history_object> graphene::app::history_api::**get_fill_order_history**(asset_id_type *a*, asset_id_type *b*, uint32_t *limit*) *const*

---

vector<bucket_object> graphene::app::history_api::**get_market_history**(asset_id_type *a*, asset_id_type *b*, uint32_t *bucket_seconds*, fc::time_point_sec *start*, fc::time_point_sec *end*) *const*

---

flat_set<uint32_t> graphene::app::history_api::**get_market_history_buckets**() *const*

---

## 9.3 Crypto API

#### 9.3.1.1 Blinding and Un-Blinding

blind_signature graphene::app::crypto_api::**blind_sign**(*const* extended_private_key_type &*key*, *const* fc::ecc::blinded_hash &*hash*, int *i*)

---

signature_type graphene::app::crypto_api::**unblind_signature**(*const* extended_private_key_type &*key*, *const* extended_public_key_type &*bob*, *const* fc::ecc::blind_signature &*sig*, *const* fc::sha256 &*hash*, int *i*)

---

commitment_type graphene::app::crypto_api::**blind**(*const* fc::ecc::blind_factor_type &*blind*, uint64_t *value*)

---

blind_factor_type graphene::app::crypto_api::**blind_sum**(*const* std::vector<blind_factor_type> &*blinds_in*, uint32_t *non_neg*)

---

#### 9.3.1.2 Rage Proofs

range_proof_info graphene::app::crypto_api::**range_get_info**(*const* std::vector<char> &*proof*)

---

std::vector<char> graphene::app::crypto_api::**range_proof_sign**(uint64_t *min_value*, *const* commitment_type &*commit*, *const* blind_factor_type &*commit_blind*, *const* blind_factor_type &*nonce*, int8_t *base10_exp*, uint8_t *min_bits*, uint64_t *actual_value*)

---

#### 9.3.1.3 Verification

bool graphene::app::crypto_api::**verify_sum**(*const* std::vector<commitment_type> &*commits_in*, *const* std::vector<commitment_type> &*neg_commits_in*, int64_t *excess*)

---

verify_range_result graphene::app::crypto_api::**verify_range**(*const* fc::ecc::commitment_type &*commit*, *const* std::vector<char> &*proof*)

---

verify_range_proof_rewind_result graphene::app::crypto_api::**verify_range_proof_rewind**(*const* blind_factor_type &*nonce*, *const* fc::ecc::commitment_type &*commit*, *const* std::vector<char> &*proof*)

---

## 9.4 Network Broadcast API

The network_broadcast_api class allows broadcasting of transactions.

#### 9.4.1.1 Transactions

void graphene::app::network_broadcast_api::**broadcast_transaction**(*const* signed_transaction &*trx*)
Broadcast a transaction to the network.
The transaction will be checked for validity in the local database prior to broadcasting. If it fails to apply locally, an error will be thrown and the transaction will not be broadcast.
**Parameters**

- trx: The transaction to broadcast

---

void graphene::app::network_broadcast_api::**broadcast_transaction_with_callback**(confirmation_callback *cb*, *const* signed_transaction &*trx*)
This version of broadcast transaction registers a callback method that will be called when the transaction is included into a block. The callback method includes the transaction id, block number, and transaction number in the block.

---

#### 9.4.1.2 Block

void graphene::app::network_broadcast_api::**broadcast_block**(*const* signed_block &*block*)

## 9.5 Network Nodes API

The network_node_api class allows maintenance of p2p connections.

#### 9.5.1.1 Obtain Network Information

fc::variant_object graphene::app::network_node_api::**get_info**() *const*
Return general network information, such as p2p port.

---

std::vector<net::peer_status> graphene::app::network_node_api::**get_connected_peers**() *const*
Get status of all current connections to peers.

---

std::vector<net::potential_peer_record> graphene::app::network_node_api::**get_potential_peers**() *const*
Return list of potential peers.

---

fc::variant_object graphene::app::network_node_api::**get_advanced_node_parameters**() *const*
Get advanced node parameters, such as desired and max number of connections.

---

#### 9.5.1.2 Change Network Settings

void graphene::app::network_node_api::**add_node**(*const* fc::ip::endpoint &*ep*)
add_node Connect to a new peer
**Parameters**

- ep: The IP/Port of the peer to connect to

---

void graphene::app::network_node_api::**set_advanced_node_parameters**(*const* fc::variant_object &*params*)
Set advanced node parameters, such as desired and max number of connections.
**Parameters**

- params: a JSON object containing the name/value pairs for the parameters to set

## 9.6 Wallet API

### *9.6.1.1 General Calls*

***help***

```
string graphene::wallet::wallet_api::help() const
```

Returns a list of all commands supported by the wallet API. This lists each command, along with its arguments and return types. For more detailed help on a single command, use get_help()
**Return**
a multiline string suitable for displaying on a terminal

***gethelp***

```
string graphene::wallet::wallet_api::gethelp(const string &method) const
```

Returns detailed help on a single API command.
**Return**
a multi-line string suitable for displaying on a terminal
**Parameters**

- method: the name of the API command you want help with variant

***info***

```
graphene::wallet::wallet_api::info()
```

Returns JSON string with informations about current blockchain state
**Return**
JSON string

***about***

```
variant_object graphene::wallet::wallet_api::about() const
```

Returns info such as client version, git version of graphene/fc, version of boost, openssl.
**Return**
compile time info and client and dependencies versions

***network_add_nodes***

```
void graphene::wallet::wallet_api::network_add_nodes(const vector<string>
&nodes)
```

Connect to a new peer
**Parameters**

- nodes: list of the IP addresses and ports of new nodes

***network_get_connected_peers***

```
vector<variant> graphene::wallet::wallet_api::network_get_connected_peers()
```
Get status of all current connections to peers

### *9.6.1.2 Wallet Calls*

***is_new***

```
bool graphene::wallet::wallet_api::is_new() const
```
Checks whether the wallet has just been created and has not yet had a password set. Calling set_password() will transition the wallet to the locked state.
**Return**
true if the wallet is new

***is_locked***

```
bool graphene::wallet::wallet_api::is_locked() const
```
Checks whether the wallet is locked (is unable to use its private keys). This state can be changed by calling lock() or unlock().
**Return**
true if the wallet is locked

***lock***

```
void graphene::wallet::wallet_api::lock()
```
Locks the wallet immediately.

***unlock***

```
void graphene::wallet::wallet_api::unlock(string password)
```
Unlocks the wallet. The wallet remain unlocked until the lock is called or the program exits.
**Parameters**

- password: the password previously set with set_password()

***set_password***

```
void graphene::wallet::wallet_api::set_password(string password)
```
Sets a new password on the wallet. The wallet must be either ‘new’ or ‘unlocked’ to execute this command.
**Parameters**

- password: new password

***dump_private_keys***

```
map<public_key_type, string>
graphene::wallet::wallet_api::dump_private_keys()
```

Dumps all private keys owned by the wallet. The keys are printed in WIF format. You can import these keys into another wallet using import_key()

**Return**

a map containing the private keys, indexed by their public key

### *import_key*

```
bool graphene::wallet::wallet_api::import_key(string account_name_or_id,
string wif_key)
```

Imports the private key for an existing account. The private key must match either an owner key or an active key for the named account.

**See**

dump_private_keys()

**Return**

true if the key was imported

**Parameters**

- account_name_or_id: the account owning the key
- wif_key: the private key in WIF format

### *import_accounts*

```
map<string, bool> graphene::wallet::wallet_api::import_accounts(string
filename, string password)
```

### *import_account_keys*

```
bool graphene::wallet::wallet_api::import_account_keys(string filename,
string password, string src_account_name, string dest_account_name)
```

### *import_balance*

```
vector<signed_transaction>
graphene::wallet::wallet_api::import_balance(string account_name_or_id, const
vector<string> &wif_keys, bool broadcast)
```

This call will construct transaction(s) that will claim all balances controlled by wif_keys and deposit them into the given account.

### *suggest_brain_key*

```
brain_key_info graphene::wallet::wallet_api::suggest_brain_key() const
```

Suggests a safe brain key to use for creating your account. create_account_with_brain_key() requires you to specify a ‘brain key’, a long passphrase that provides enough entropy to generate cryptographic keys. This function will suggest a suitably random string that should be easy to write down (and, with effort, memorize).

**See**

create_account_with_brain_key()
**Return**
a suggested brain_key

### *get_transaction_id*

```
transaction_id_type graphene::wallet::wallet_api::get_transaction_id(const
signed_transaction &trx) const
```
This method is used to convert a JSON transaction to its transaction ID.

### *get_private_key*

```
string graphene::wallet::wallet_api::get_private_key(public_key_type pubkey)
const
```
Get the WIF private key corresponding to a public key. The private key must already be in the wallet.

### *load_wallet_file*

```
bool graphene::wallet::wallet_api::load_wallet_file(string wallet_filename =
"")
```
Loads a specified Graphene wallet. The current wallet is closed before the new wallet is loaded.
**Warning:** This does not change the filename that will be used for future wallet writes, so this may cause you to overwrite your original wallet unless you also call set_wallet_filename()
**Return**
true if the specified wallet is loaded
**Parameters**

- wallet_filename: the filename of the wallet JSON file to load. If wallet_filename is empty, it reloads the existing wallet file

### *get_wallet_filename*

```
string graphene::wallet::wallet_api::get_wallet_filename()
```
Returns the current wallet filename. This is the filename that will be used when automatically saving the wallet.
**See**
set_wallet_filename()
**Return**
the wallet filename

### *set_wallet_filename*

```
void graphene::wallet::wallet_api::set_wallet_filename(string
wallet_filename)
```
Sets the wallet filename used for future writes. This does not trigger a save, it only changes the default filename that will be used the next time a save is triggered.
Parameters

- wallet_filename: the new filename to use for future saves

***normalize_brain_key***

```
string graphene::wallet::wallet_api::normalize_brain_key(string s) const
```

Transforms a brain key to reduce the chance of errors when re-entering the key from memory. This takes a user-supplied brain key and normalizes it into the form used for generating private keys. In particular, this upper-cases all ASCII characters and collapses multiple spaces into one.

**Return**

the brain key in its normalized form

**Parameters**

- s: the brain key as supplied by the user

***save_wallet_file***

```
void graphene::wallet::wallet_api::save_wallet_file(string wallet_filename = "")
```

Saves the current wallet to the given filename.

**Warning:** This does not change the wallet filename that will be used for future writes, so think of this function as 'Save a Copy As…' instead of 'Save As…'. Use set_wallet_filename() to make the filename persist.

**Parameters**

- wallet_filename: the filename of the new wallet JSON file to create or overwrite. If wallet_filename is empty, save to the current filename.

#### 9.6.1.3 Account Calls

***list_my_accounts***

```
vector<account_object> graphene::wallet::wallet_api::list_my_accounts()
```

Lists all accounts controlled by this wallet. This returns a list of the full account objects for all accounts whose private keys we possess.

**Return**

a list of account objects

***list_accounts***

```
map<string, account_id_type>
graphene::wallet::wallet_api::list_accounts(const string &lowerbound,
uint32_t limit)
```

Lists all accounts registered in the blockchain. This returns a list of all account names and their account ids, sorted by account name. Use the lowerbound and limit parameters to page through the list. To retrieve all accounts, start by setting lowerbound to the empty string "", and then each iteration, pass the last account name returned as the lowerbound for the next list_accounts() call.

**Return**

a list of accounts mapping account names to account ids

**Parameters**

- lowerbound: the name of the first account to return. If the named account does not exist, the list will start at the account that comes after lowerbound
- limit: the maximum number of accounts to return (max: 1000)

***list_account_balances***

```
vector<asset> graphene::wallet::wallet_api::list_account_balances(const string &id)
```

List the balances of an account. Each account can have multiple balances, one for each type of asset owned by that account. The returned list will only contain assets for which the account has a nonzero balance

**Return**

a list of the given account balances

**Parameters**

- id: the name or id of the account whose balances you want

***register_account***

```
signed_transaction graphene::wallet::wallet_api::register_account(string name, public_key_type owner, public_key_type active, string registrar_account, string referrer_account, uint32_t referrer_percent, bool broadcast = false)
```

Registers a third party's account on the blockchain. This function is used to register an account for which you do not own the private keys. When acting as a registrar, an end user will generate their own private keys and send you the public keys. The registrar will use this function to register the account on behalf of the end user.

**See**

create_account_with_brain_key()

**Return**

the signed transaction registering the account

**Parameters**

- name: the name of the account, must be unique on the blockchain. Shorter names are more expensive to register; the rules are still in flux, but in general names of more than 8 characters with at least one digit will be cheap.
- owner: the owner key for the new account
- active: the active key for the new account
- registrar_account: the account which will pay the fee to register the user
- referrer_account: the account who is acting as a referrer, and may receive a portion of the user's transaction fees. This can be the same as the registrar_account if there is no referrer.
- referrer_percent: the percentage (0 - 100) of the new user's transaction fees not claimed by the blockchain that will be distributed to the referrer; the rest will be sent to the registrar. Will be multiplied by GRAPHENE_1_PERCENT when constructing the transaction.
- broadcast: true to broadcast the transaction on the network

***register_vault_account***

```
signed_transaction
graphene::wallet::wallet_api::register_vault_account(string name,
public_key_type owner, public_key_type active, bool broadcast = false)
```

Registers a third party's vault account on the blockchain. This function is used to register a vault account for which you do not own the private keys. When acting as a registrar, an end user will generate their own private keys and send you the public keys. The registrar will use this function to register the account on behalf of the end user.

**See**

register_account()

**Return**

the signed transaction registering vault account

**Parameters**

- name: the name of the account, must be unique on the blockchain. Shorter names are more expensive to register; the rules are still in flux, but in general names of more than 8 characters with at least one digit will be cheap.
- owner: the owner key for the new account
- active: the active key for the new account
- broadcast: true to broadcast the transaction on the network

### *tether_accounts*

```
signed_transaction graphene::wallet::wallet_api::tether_accounts(string
wallet, string vault, bool broadcast = false)
```

Tethers a wallet account to a vault account on the blockchain. This function is used to tether two accounts (a wallet and a vault)

**Return**

the signed transaction tethering two accounts

**Parameters**

- wallet: the name or id of the wallet account to tether
- vault: the name or id of the vault account to tether
- broadcast: true to broadcast the transaction on the network

### *upgrade_account*

```
signed_transaction graphene::wallet::wallet_api::upgrade_account(string name,
bool broadcast)
```

Upgrades an account to prime status. This makes the account holder a 'lifetime member'.

**Return**

the signed transaction upgrading the account

**Parameters**

- name: the name or id of the account to upgrade
- broadcast: true to broadcast the transaction on the network

### *create_account_with_brain_key*

```
signed_transaction
graphene::wallet::wallet_api::create_account_with_brain_key(string brain_key,
```

```
string account_name, string registrar_account, string referrer_account, bool
broadcast = false)
```

Creates a new account and registers it on the blockchain.

**See**

suggest_brain_key()

register_account()

**Return**

the signed transaction registering the account

**Parameters**

- brain_key: the brain key used for generating the account's private keys
- account_name: the name of the account, must be unique on the blockchain. Shorter names are more expensive to register; the rules are still in flux, but in general names of more than 8 characters with at least one digit will be cheap.
- registrar_account: the account which will pay the fee to register the user
- referrer_account: the account who is acting as a referrer, and may receive a portion of the user's transaction fees. This can be the same as the registrar_account if there is no referrer.
- broadcast: true to broadcast the transaction on the network

***create_account***

```
signed_transaction graphene::wallet::wallet_api::create_account(account_kind
kind, string name, public_key_type owner, public_key_type active, bool
broadcast = false)
```

Registers a third party's account on the blockchain. This function is used to register an account for which you do not own the private keys. When acting as a registrar, an end user will generate their own private keys and send you the public keys. The registrar will use this function to register the account on behalf of the end user.

**See**

create_account_with_brain_key()

**Return**

the signed transaction registering the account

**Parameters**

- kind: the kind of the account, i.e. vault
- name: the name of the account, must be unique on the blockchain. Shorter names are more expensive to register; the rules are still in flux, but in general names of more than 8 characters with at least one digit will be cheap.
- owner: the owner key for the new account
- active: the active key for the new account
- broadcast: true to broadcast the transaction on the network

***transfer***

```
signed_transaction graphene::wallet::wallet_api::transfer(string from, string
to, string amount, string asset_symbol, string memo, bool broadcast = false)
```

Transfer an amount from one account to another.

**Return**

the signed transaction transferring funds

**Parameters**

- from: the name or id of the account sending the funds
- to: the name or id of the account receiving the funds
- amount: the amount to send (in nominal units to send half of a BTS, specify 0.5)
- asset_symbol: the symbol or id of the asset to send
- memo: a memo to attach to the transaction. The memo will be encrypted in the transaction and readable for the receiver. There is no length limit other than the limit imposed by maximum transaction size, but transaction increase with transaction size
- broadcast: true to broadcast the transaction on the network

*transfer2*

```
pair<transaction_id_type, signed_transaction>
graphene::wallet::wallet_api::transfer2(string from, string to, string
amount, string asset_symbol, string memo)
```

This method works just like transfer, except it always broadcasts and returns the transaction ID along with the signed transaction.

*whitelist_account*

```
signed_transaction graphene::wallet::wallet_api::whitelist_account(string
authorizing_account, string account_to_list,
account_whitelist_operation::account_listing new_listing_status, bool
broadcast = false)
```

Whitelist and blacklist accounts, primarily for transacting in whitelisted assets. Accounts can freely specify opinions about other accounts, in the form of either whitelisting or blacklisting them. This information is used in chain validation only to determine whether an account is authorized to transact in an asset type which enforces a whitelist, but third parties can use this information for other uses as well, as long as it does not conflict with the use of whitelisted assets. An asset which enforces a whitelist specifies a list of accounts to maintain its whitelist, and a list of accounts to maintain its blacklist. In order for a given account A to hold and transact in a whitelisted asset S, A must be whitelisted by at least one of S's whitelist_authorities and blacklisted by none of S's blacklist_authorities. If A receives a balance of S, and is later removed from the whitelist(s) which allowed it to hold S, or added to any blacklist S specifies as authoritative, A's balance of S will be frozen until A's authorization is reinstated.

**Return**

the signed transaction changing the whitelisting status

**Parameters**

- authorizing_account: the account who is doing the whitelisting
- account_to_list: the account being whitelisted
- new_listing_status: the new whitelisting status
- broadcast: true to broadcast the transaction on the network

*get_vesting_balances*

```
vector<vesting_balance_object_with_info>
graphene::wallet::wallet_api::get_vesting_balances(string account_name)
```

Get information about a vesting balance object.

**Parameters**

- account_name: An account name, account ID, or vesting balance object ID.

*withdraw_vesting*

```
signed_transaction graphene::wallet::wallet_api::withdraw_vesting(string
witness_name, string amount, string asset_symbol, bool broadcast = false)
```

Withdraw a vesting balance.

**Parameters**

- witness_name: The account name of the witness, also accepts account ID or vesting balance ID type.
- amount: The amount to withdraw.
- asset_symbol: The symbol of the asset to withdraw.
- broadcast: true if you wish to broadcast the transaction

*get_account*

```
account_object graphene::wallet::wallet_api::get_account(string
account_name_or_id) const
```

Returns information about the given account.

**Return**

the public account data stored in the blockchain

**Parameters**

- account_name_or_id: the name or id of the account to provide information about

*get_account_id*

```
account_id_type graphene::wallet::wallet_api::get_account_id(string
account_name_or_id) const
```

Lookup the id of a named account.

**Return**

the id of the named account

**Parameters**

- account_name_or_id: the name of the account to look up

*get_account_history*

```
vector<operation_detail>
graphene::wallet::wallet_api::get_account_history(string name, int limit)
const
```

Returns the most recent operations on the named account. This returns a list of operation history objects, which describe activity on the account.

**Return**

a list of operation_history_objects

**Parameters**

- name: the name or id of the account
- limit: the number of entries to return (starting from the most recent)

***approve_proposal***

```
signed_transaction graphene::wallet::wallet_api::approve_proposal(const
string &fee_paying_account, const string &proposal_id, const approval_delta
&delta, bool broadcast)
```

Approve or disapprove a proposal.

**Return**

the signed version of the transaction

**Parameters**

- fee_paying_account: The account paying the fee for the op.
- proposal_id: The proposal to modify.
- delta: Members contain approvals to create or remove. In JSON you can leave empty members undefined.
- broadcast: true if you wish to broadcast the transaction

***get_account_cycle_balance***

```
acc_id_share_t_res
graphene::wallet::wallet_api::get_account_cycle_balance(const string&
account) const
```

Get the amount of cycles in the account.

**Return**

Cycle balance of the account.

**Parameters**

- account: Account name or stringified id.

***get_full_cycle_balances***

```
acc_id_vec_cycle_agreement_res
graphene::wallet::wallet_api::get_full_cycle_balances(const string& account)
const
```

Deprecated

***purchase_cycle_asset***

```
signed_transaction graphene::wallet::wallet_api::purchase_cycle_asset(string
account, string amount_to_sell, string symbol_to_sell, double frequency,
double amount_of_cycles_to_receive, bool broadcast = false)
```

Purchase cycles.

**Parameters**

- account: account name or id
- amount_to_sell: amount of asset to sell
- symbol_to_sell: symbol of asset to sell
- frequency: frequency at which we buy
- amount_of_cycles_to_receive: amount of cycles to receive by this buy

***calculate_cycle_price***

```
optional<cycle_price>
graphene::wallet::wallet_api::calculate_cycle_price(share_type cycle_amount,
asset_id_type asset_id) const
```

Calculates and returns the amount of asset one needs to pay to get the given amount of cycles
**Return**
cycle_price structure (optional)
**Parameters**

- cycle_amount: Desired amount of cycles to get
- asset_id_type: Asset to pay

***get_dascoin_balance***

```
acc_id_share_t_res graphene::wallet::wallet_api::get_dascoin_balance(const
string& account) const
```

Get amount of DasCoin for on an account.
**Return**
An object containing dascoin balance of an account
**Parameters**

- account: Account name or stringified id.

#### 9.6.1.4 Wire out Calls

***wire_out***

```
signed_transaction graphene::wallet::wallet_api::wire_out(const string&
account, share_type amount, bool broadcast) const
```

Wire out some WebAsset
**Return**
the signed transaction wiring out webasset
**Parameters**

- account: Account name or stringified id
- amount: the amount to wire out
- broadcast: true to broadcast the transaction on the network

***wire_out_with_fee***

```
signed_transaction graphene::wallet::wallet_api::wire_out_with_fee(const
string& account, share_type amount, const string& currency_of_choice, const
string& to_address, const string& memo, bool broadcast) const
```

Wire out with fee some WebAsset
**Return**
the signed transaction wiring out webasset
**Parameters**

- account: Account name or stringified id
- amount: the amount to wire out

- currency_of_choice: Currency of choice (string abbreviation) in which user wants wire out
- to_address: Destination blockchain address to which the amount needs to be wired
- memo: Optional note
- broadcast: true to broadcast the transaction on the network

#### 9.6.1.5 Trading Calls

***sell_asset***

```
signed_transaction graphene::wallet::wallet_api::sell_asset(string
seller_account, string amount_to_sell, string symbol_to_sell, string
min_to_receive, string symbol_to_receive, uint32_t timeout_sec = 0, bool
fill_or_kill = false, bool broadcast = false)
```

Place a limit order attempting to sell one asset for another. Buying and selling are the same operation on Graphene; if you want to buy DASC with USD, you should sell USD for DASC. The blockchain will attempt to sell the symbol_to_sell for as much symbol_to_receive as possible, as long as the price is at least min_to_receive / amount_to_sell. In addition to the transaction fees, market fees will apply as specified by the issuer of both the selling asset and the receiving asset as a percentage of the amount exchanged. If either the selling asset or the receiving asset is whitelist restricted, the order will only be created if the seller is on the whitelist of the restricted asset type. Market orders are matched in the order they are included in the blockchain.

**Return**

the signed transaction selling the funds

**Parameters**

- seller_account: the account providing the asset being sold, and which will receive the proceeds of the sale.
- amount_to_sell: the amount of the asset being sold to sell (in nominal units)
- symbol_to_sell: the name or id of the asset to sell
- min_to_receive: the minimum amount you are willing to receive in return for selling the entire amount_to_sell
- symbol_to_receive: the name or id of the asset you wish to receive
- timeout_sec: if the order does not fill immediately, this is the length of time the order will remain on the order books before it is cancelled and the un-spent funds are returned to the seller's account
- fill_or_kill: if true, the order will only be included in the blockchain if it is filled immediately; if false, an open order will be left on the books to fill any amount that cannot be filled immediately.
- broadcast: true to broadcast the transaction on the network

***sell***

```
signed_transaction graphene::wallet::wallet_api::sell(string seller_account,
string base, string quote, string rate, double amount, bool broadcast =
false)
```

Place a limit order attempting to sell one asset for another.
This API call abstracts away some of the details of the sell_asset() call to be more user friendly.
All orders placed with sell never timeout and will not be killed if they cannot be filled

immediately. If you wish for one of these parameters to be different, then sell_asset() should be used instead.
**Return**
the signed transaction selling the funds
**Parameters**

- seller_account: the account providing the asset being sold, and which will receive the proceeds of the sale.
- base: the name or id of the asset to sell.
- quote: the name or id of the asset to receive.
- rate: The rate in base:quote at which you want to sell.
- amount: the amount of base you want to sell.
- broadcast: true to broadcast the transaction on the network

***buy***

```
signed_transaction graphene::wallet::wallet_api::buy(string buyer_account,
string base, string quote, double rate, double amount, bool broadcast =
false)
```

Place a limit order attempting to buy one asset with another.
This API call abstracts away some of the details of the sell_asset() call to be more user friendly. All orders placed with buy never timeout and will not be killed if they cannot be filled immediately. If you wish for one of these parameters to be different, then sell_asset() should be used instead
**Return**
the signed transaction buying the funds
**Parameters**

- buyer_account: the account buying the asset for another asset
- base: the name or id of the asset to buy.
- quote: the name or id of the asset being offered as payment.
- rate: the rate in base:quote at which you want to buy.
- amount: the amount of base you want to buy.
- broadcast: true to broadcast the transaction on the network

***borrow_asset***

```
signed_transaction graphene::wallet::wallet_api::borrow_asset(string
borrower_name, string amount_to_borrow, string asset_symbol, string
amount_of_collateral, bool broadcast = false)
```

Borrow an asset or update the debt/collateral ratio for the loan. This is the first step in shorting an asset. Call sell_asset() to complete the short.
**Return**
the signed transaction borrowing the asset
**Parameters**

- borrower_name: the name or id of the account associated with the transaction.
- amount_to_borrow: the amount of the asset being borrowed. Make this value negative to pay back debt.
- asset_symbol: the symbol or id of the asset being borrowed.

- amount_of_collateral: the amount of the backing asset to add to your collateral position. Make this negative to claim back some of your collateral. The backing asset is defined in the bitasset_options for the asset being borrowed.
- broadcast: true to broadcast the transaction on the network

***cancel_order***

```
signed_transaction graphene::wallet::wallet_api::cancel_order(object_id_type
order_id, bool broadcast = false)
```

Cancel an existing order
**Return**
the signed transaction canceling the order
**Parameters**

- order_id: the id of order to be cancelled
- broadcast: true to broadcast the transaction on the network

***settle_asset***

```
signed_transaction graphene::wallet::wallet_api::settle_asset(string
account_to_settle, string amount_to_settle, string symbol, bool broadcast =
false)
```

Schedules a market-issued asset for automatic settlement. Holders of market-issued assets may request a forced settlement for some amount of their asset. This means that the specified sum will be locked by the chain and held for the settlement period, after which time the chain will choose a margin position holder and buy the settled asset using the margin collateral. The price of this sale will be based on the feed price for the market-issued asset being settled. The exact settlement price will be the feed price at the time of settlement with an offset in favor of the margin position, where the offset is a blockchain parameter set in the global_property_object.
**Return**
the signed transaction settling the named asset
**Parameters**

- account_to_settle: the name or id of the account owning the asset
- amount_to_settle: the amount of the named asset to schedule for settlement
- symbol: the name or id of the asset to settlement on
- broadcast: true to broadcast the transaction on the network

***get_market_history***

```
vector<bucket_object> graphene::wallet::wallet_api::get_market_history(string
symbol, string symbol2, uint32_t bucket, fc::time_point_sec start,
fc::time_point_sec end) const
```

***get_limit_orders***

```
vector<limit_order_object>
graphene::wallet::wallet_api::get_limit_orders(string a, string b, uint32_t
limit) const
```

***get_call_orders***

```
vector<call_order_object>
graphene::wallet::wallet_api::get_call_orders(string a, uint32_t limit) const
```

***get_settle_orders***

```
vector<force_settlement_object>
graphene::wallet::wallet_api::get_settle_orders(string a, uint32_t limit)
const
```

#### 9.6.1.6 Asset Calls

***list_assets***

```
vector<asset_object> graphene::wallet::wallet_api::list_assets(const string
&lowerbound, uint32_t limit) const
```

Lists all assets registered on the blockchain. To list all assets, pass the empty string "" for the lowerbound to start at the beginning of the list, and iterate as necessary.

**Return**

the list of asset objects, ordered by symbol

**Parameters**

- lowerbound: the symbol of the first asset to include in the list.
- limit: the maximum number of assets to return (max: 100)

***create_asset***

```
signed_transaction graphene::wallet::wallet_api::create_asset(string issuer,
string symbol, uint8_t precision, asset_options common,
fc::optional<bitasset_options> bitasset_opts, bool broadcast = false)
```

Creates a new user-issued or market-issued asset. Many options can be changed later using update_asset(). Right now this function is difficult to use because you must provide raw JSON data structures for the options objects, and those include prices and asset ids.

**Return**

the signed transaction creating a new asset

**Parameters**

- issuer: the name or id of the account who will pay the fee and become the issuer of the new asset. This can be updated later
- symbol: the ticker symbol of the new asset
- precision: the number of digits of precision to the right of the decimal point, must be less than or equal to 12
- common: asset options required for all new assets. Note that core_exchange_rate technically needs to store the asset ID of this new asset. Since this ID is not known at the time this operation is created, create this price as though the new asset has instance ID 1, and the chain will overwrite it with the new asset's ID.
- bitasset_opts: options specific to BitAssets. This may be null unless the market_issued flag is set in common.flags
- broadcast: true to broadcast the transaction on the network

***update_asset***

```
signed_transaction graphene::wallet::wallet_api::update_asset(string symbol,
optional<string> new_issuer, asset_options new_options, bool broadcast =
false)
```

Update the core options on an asset. There are a number of options which all assets in the network use. These options are enumerated in the asset_object::asset_options struct. This command is used to update these options for an existing asset.

Note: *This operation cannot be used to update BitAsset-specific options. For these options, update_bitasset() instead.*

**Return**

the signed transaction updating the asset

**Parameters**

- symbol: the name or id of the asset to update
- new_issuer: if changing the asset's issuer, the name or id of the new issuer. null if you wish to remain the issuer of the asset
- new_options: the new asset_options object, which will entirely replace the existing options.
- broadcast: true to broadcast the transaction on the network

***update_bitasset***

```
signed_transaction graphene::wallet::wallet_api::update_bitasset(string
symbol, bitasset_options new_options, bool broadcast = false)
```

Update the options specific to a BitAsset. BitAssets have some options which are not relevant to other asset types. This operation is used to update those options an an existing BitAsset.

**See**

update_asset()

**Return**

the signed transaction updating the bitasset

**Parameters**

- symbol: the name or id of the asset to update, which must be a market-issued asset
- new_options: the new bitasset_options object, which will entirely replace the existing options.
- broadcast: true to broadcast the transaction on the network

***update_asset_feed_producers***

```
signed_transaction
graphene::wallet::wallet_api::update_asset_feed_producers(string symbol,
flat_set<string> new_feed_producers, bool broadcast = false)
```

Update the set of feed-producing accounts for a BitAsset. BitAssets have price feeds selected by taking the median values of recommendations from a set of feed producers. This command is used to specify which accounts may produce feeds for a given BitAsset.

**Return**

the signed transaction updating the bitasset's feed producers

**Parameters**

- symbol: the name or id of the asset to update

- new_feed_producers: a list of account names or ids which are authorized to produce feeds for the asset. this list will completely replace the existing list
- broadcast: true to broadcast the transaction on the network

***publish_asset_feed***

```
signed_transaction graphene::wallet::wallet_api::publish_asset_feed(string publishing_account, string symbol, price_feed feed, bool broadcast = false)
```

Publishes a price feed for the named asset. Price feed providers use this command to publish their price feeds for market-issued assets. A price feed is used to tune the market for a particular market-issued asset. For each value in the feed, the median across all committee_member feeds for that asset is calculated and the market for the asset is configured with the median of that value. The feed object in this command contains three prices: a call price limit, a short price limit, and a settlement price. The call limit price is structured as (collateral asset) / (debt asset) and the short limit price is structured as (asset for sale) / (collateral asset). Note that the asset IDs are opposite to each other, so if we're publishing a feed for USD, the call limit price will be CORE/USD and the short limit price will be USD/CORE. The settlement price may be flipped either direction, as long as it is a ratio between the market-issued asset and its collateral.

**Return**

the signed transaction updating the price feed for the given asset

**Parameters**

- publishing_account: the account publishing the price feed
- symbol: the name or id of the asset whose feed we're publishing
- feed: the price_feed object containing the three prices making up the feed
- broadcast: true to broadcast the transaction on the network

***issue_asset***

```
signed_transaction graphene::wallet::wallet_api::issue_asset(string to_account, string amount, string symbol, string memo, bool broadcast = false)
```

Issue new shares of an asset.

**Return**

the signed transaction issuing the new shares

**Parameters**

- to_account: the name or id of the account to receive the new shares
- amount: the amount to issue, in nominal units
- symbol: the ticker symbol of the asset to issue
- memo: a memo to include in the transaction, readable by the recipient
- broadcast: true to broadcast the transaction on the network

***issue_webasset***

```
signed_transaction graphene::wallet::wallet_api::issue_webasset(string to_account, string amount, string reserved, bool broadcast = false)
```

Issue webasset to an account's balance.

**Return**

the signed transaction issuing webasset

**Parameters**

- to_account: the name or id of the account to receive the new shares
- amount: the amount to issue, in nominal units
- reserved: reserved amount to issue, in nominal units
- broadcast: true to broadcast the transaction on the network

***get_asset***

```
asset_object graphene::wallet::wallet_api::get_asset(string asset_name_or_id)
const
```

Returns information about the given asset.

**Return**

the information about the asset stored in the blockchain

**Parameters**

- asset_name_or_id: the symbol or id of the asset in question

***get_asset_id***

```
asset_id_type graphene::wallet::wallet_api::get_asset_id(string
asset_name_or_id) const
```

Lookup the id of a named asset.

**Return**

the id of the given asset

**Parameters**

- asset_name_or_id: the symbol or id of the asset in question

***get_bitasset_data***

```
asset_bitasset_data_object
graphene::wallet::wallet_api::get_bitasset_data(string asset_name_or_id)
const
```

Returns the BitAsset-specific data for a given asset. Market-issued assets's behavior are determined both by their "BitAsset Data" and their basic asset data, as returned by get_asset().

**Return**

the BitAsset-specific data for this asset

**Parameters**

- asset_name_or_id: the symbol or id of the BitAsset in question

***fund_asset_fee_pool***

```
signed_transaction graphene::wallet::wallet_api::fund_asset_fee_pool(string
from, string symbol, string amount, bool broadcast = false)
```

Pay into the fee pool for the given asset. User-issued assets can optionally have a pool of the core asset which is automatically used to pay transaction fees for any transaction using that asset (using the asset's core exchange rate). This command allows anyone to deposit the core asset into this fee pool.

**Return**

the signed transaction funding the fee pool

**Parameters**

- from: the name or id of the account sending the core asset
- symbol: the name or id of the asset whose fee pool you wish to fund
- amount: the amount of the core asset to deposit
- broadcast: true to broadcast the transaction on the network

***reserve_asset***

```
signed_transaction graphene::wallet::wallet_api::reserve_asset(string from,
string amount, string symbol, bool broadcast = false)
```

Burns the given user-issued asset. This command burns the user-issued asset to reduce the amount in circulation. Note: *you cannot burn market-issued assets.*

**Return**

the signed transaction burning the asset

**Parameters**

- from: the account containing the asset you wish to burn
- amount: the amount to burn, in nominal units
- symbol: the name or id of the asset to burn
- broadcast: true to broadcast the transaction on the network

***global_settle_asset***

```
signed_transaction graphene::wallet::wallet_api::global_settle_asset(string
symbol, price settle_price, bool broadcast = false)
```

Forces a global settling of the given asset (black swan or prediction markets). In order to use this operation, asset_to_settle must have the global_settle flag set. When this operation is executed all balances are converted into the backing asset at the settle_price and all open margin positions are called at the settle price. If this asset is used as backing for other bitassets, those bitassets will be force settled at their current feed price. Note: *this operation is used only by the asset issuer, settle_asset() may be used by any user owning the asset*

**Return**

the signed transaction settling the named asset

**Parameters**

- symbol: the name or id of the asset to force settlement on
- settle_price: the price at which to settle
- broadcast: true to broadcast the transaction on the network

#### *9.6.1.7 Licence Calls*

***issue_license***

```
signed_transaction graphene::wallet::wallet_api::issue_license(const string&
issuer, const string& account, const string& license, share_type
bonus_percentage, frequency_type frequency, bool broadcast = false);
```

Issue a license to an account. This will create a license request object that can be denied by the license authentication authority.

**Return**

The signed version of the transaction

**Parameters**

- issuer: this MUST be the license issuing chain authority.
- account: the account that will benefit the license.
- license: the id of the license that will be granted to the account.
- bonus_percentage: bonus percentage of license base cycles to be issued. Value must be greater than -100.
- frequency: frequency lock for this license.
- broadcast: true if you wish to broadcast the transaction.

***get_license_types***

```
vector<license_type_object> graphene::wallet::wallet_api::get_license_types()
const
```

Get all license type ids found on the blockchain

**Returns**

Vector of license type ids

***get_license_type_names_ids***

```
vector<pair<string, license_type_id_type>>
graphene::wallet::wallet_api::get_license_type_names_ids() const;
```

Get names and license type ids found on the blockchain.

**Return**

Vector of license name/type-ids pairs

***get_license_information***

```
vector<optional<license_information_object>>
graphene::wallet::wallet_api::get_license_information(const
vector<account_id_type>& account_ids) const;
```

Get a list of account issued license types. This function has semantics identical to get_objects.

**Return**

Vector of issued license information objects

**Parameters**

- account_ids: IDs of the accounts to retrieve.

#### *9.6.1.8 Reward Queue Calls*

***update_queue_parameters***

```
signed_transaction
graphene::wallet::wallet_api::update_queue_parameters(optional<bool>
enable_dascoin_queue, optional<uint32_t> reward_interval_time_seconds,
optional<share_type> dascoin_reward_amount, bool broadcast) const;
```

Update various reward queue parameters

**Return**

signed transaction updating the queue

**Parameters**

- enable_dascoin_queue: true if minting is enabled
- reward_interval_time_seconds: the time interval between DasCoin reward events
- dascoin_reward_amount: the amount of DasCoins produced on the DasCoin reward event
- broadcast: true if you wish to broadcast the transaction

### *get_order_book*

```
order_book graphene::wallet::wallet_api::get_order_book(const string& base, const string& quote, unsigned limit = 50)
```

Returns the order book for the market base:quote.

**Return**

Order book of the market

**Parameters**

- base: name of the first asset
- quote: name of the second asset
- limit: of the order book. Up to limit of each asks and bids, capped at 50. Prioritizes most moderate of each

### *get_reward_queue_size*

```
uint32_t graphene::wallet::wallet_api::get_reward_queue_size() const
```

Gives the size of the DASCoin reward queue

**Return**

Number of elements in the DASCoin queue

### *get_reward_queue*

```
vector<reward_queue_object> graphene::wallet::wallet_api::get_reward_queue() const
```

Gives the entire reward queue.

**Return**

Vector of all reward queue objects

### *get_reward_queue_by_page*

```
vector<reward_queue_object> graphene::wallet::wallet_api::get_reward_queue_by_page(uint32_t from, uint32_t amount) const
```

Returns a part of the reward queue.

**Return**

Vector of reward queue objects

**Parameters**

- from: Starting page
- amount: Number of pages to get

### *get_queue_submissions_with_pos*

```
acc_id_queue_subs_w_pos_res get_queue_submissions_with_pos(account_id_type account_id) const
```
Get all current submissions to reward queue by account id
**Return** All elements on DasCoin reward queue submitted by given account
**Parameters**

- account_id: id of account whose submissions should be returned

###### 9.6.1.9 Requests Inspection

***get_all_webasset_issue_requests***

```
vector<issue_asset_request_object>
graphene::app::database_api::get_all_webasset_issue_requests() const
```
Get all webasset issue request objects, sorted by expiration.
**Return**
Vector of webasset issue request objects.

***get_all_wire_out_holders***

```
vector<wire_out_holder_object>
graphene::app::database_api::get_all_wire_out_holders() const
```
Get all wire out holder objects.
**Return**
Vector of wire out holder objects.

***get_all_wire_out_with_fee_holder***

```
vector<wire_out_with_fee_holder_object>
graphene::app::database_api::get_all_wire_out_with_fee_holders() const
```
Get all wire out with fee holder objects.
**Return**
Vector of wire out with fee holder objects.

###### 9.6.1.10 Governance

***create_committee_member***

```
signed_transaction
graphene::wallet::wallet_api::create_committee_member(string owner_account, string url, bool broadcast = false)
```
Creates a committee_member object owned by the given account. An account can have at most one committee_member object.
**Return**
the signed transaction registering a committee_member
**Parameters**

- owner_account: the name or id of the account which is creating the committee_member
- url: a URL to include in the committee_member record in the blockchain. Clients may display this when showing a list of committee_members. May be blank.

- broadcast: true to broadcast the transaction on the network

***get_witness***

```
witness_object graphene::wallet::wallet_api::get_witness(string
owner_account)
```

Returns information about the given witness.

**Return**

the information about the witness stored in the blockchain

**Parameters**

- owner_account: the name or id of the witness account owner, or the id of the witness

***get_committee_member***

```
committee_member_object
graphene::wallet::wallet_api::get_committee_member(string owner_account)
```

Returns information about the given committee_member.

**Return**

the information about the committee_member stored in the blockchain

**Parameters**

- owner_account: the name or id of the committee_member account owner, or the id of the committee_member

***list_witnesses***

```
map<string, witness_id_type>
graphene::wallet::wallet_api::list_witnesses(const string &lowerbound,
uint32_t limit)
```

Lists all witnesses registered in the blockchain. This returns a list of all account names that own witnesses, and the associated witness id, sorted by name. This lists witnesses whether they are currently voted in or not. Use the lowerbound and limit parameters to page through the list. To retrieve all witnesses, start by setting lowerbound to the empty string "", and then each iteration, pass the last witness name returned as the lowerbound for the next list_witnesss() call.

**Return**

a list of witnesses mapping witness names to witness ids

**Parameters**

- lowerbound: the name of the first witness to return. If the named witness does not exist, the list will start at the witness that comes after lowerbound
- limit: the maximum number of witnesses to return (max: 1000)

***list_committee_members***

```
map<string, committee_member_id_type>
graphene::wallet::wallet_api::list_committee_members(const string
&lowerbound, uint32_t limit)
```

Lists all committee_members registered in the blockchain. This returns a list of all account names that own committee_members, and the associated committee_member id, sorted by name. This lists committee_members whether they are currently voted in or not. Use the lowerbound and limit parameters to page through the list. To retrieve all committee_members, start by setting lowerbound to the empty string "", and then each iteration, pass the last committee_member name returned as the lowerbound for the next list_committee_members() call.

**Return**

a list of committee_members mapping committee_member names to committee_member ids

**Parameters**

- lowerbound: the name of the first committee_member to return. If the named committee_member does not exist, the list will start at the committee_member that comes after lowerbound
- limit: the maximum number of committee_members to return (max: 1000)

***create_witness***

```
signed_transaction graphene::wallet::wallet_api::create_witness(string owner_account, string url, bool broadcast = false)
```

Creates a witness object owned by the given account. An account can have at most one witness object.

**Return**

the signed transaction registering a witness

**Parameters**

- owner_account: the name or id of the account which is creating the witness
- url: a URL to include in the witness record in the blockchain. Clients may display this when showing a list of witnesses. May be blank.
- broadcast: true to broadcast the transaction on the network

***update_witness***

```
signed_transaction graphene::wallet::wallet_api::update_witness(string witness_name, string url, string block_signing_key, bool broadcast = false)
```

Update a witness object owned by the given account.

**Return**

signed transaction

**Parameters**

- witness: The name of the witness's owner account. Also accepts the ID of the owner account or the ID of the witness.
- url: Same as for create_witness. The empty string makes it remain the same.
- block_signing_key: The new block signing public key. The empty string makes it remain the same.
- broadcast: true if you wish to broadcast the transaction.

***create_worker***

```
signed_transaction graphene::wallet::wallet_api::create_worker(string owner_account, time_point_sec work_begin_date, time_point_sec work_end_date,
```

```
share_type daily_pay, string name, string url, variant worker_settings, bool
broadcast = false)
```

Create a worker object.

**Parameters**

- owner_account: The account which owns the worker and will be paid
- work_begin_date: When the work begins
- work_end_date: When the work ends
- daily_pay: Amount of pay per day (NOT per maint interval)
- name: Any text
- url: Any text
- worker_settings: {“type” : “burn”|”refund”|”vesting”, “pay_vesting_period_days” : x}
- broadcast: true if you wish to broadcast the transaction.

***update_worker_votes***

```
signed_transaction graphene::wallet::wallet_api::update_worker_votes(string
account, worker_vote_delta delta, bool broadcast = false)
```

Update your votes for a worker

**Parameters**

- account: The account which will pay the fee and update votes.
- worker_vote_delta: {“vote_for” : […], “vote_against” : […], “vote_abstain” : […]}
- broadcast: true if you wish to broadcast the transaction.

***vote_for_committee_member***

```
signed_transaction
graphene::wallet::wallet_api::vote_for_committee_member(string
voting_account, string committee_member, bool approve, bool broadcast =
false)
```

Vote for a given committee_member. An account can publish a list of all committee_memberes they approve of. This command allows you to add or remove committee_memberes from this list. Each account’s vote is weighted according to the number of shares of the core asset owned by that account at the time the votes are tallied. Note: *you cannot vote against a committee_member, you can only vote for the committee_member or not vote for the committee_member.*

**Return**

the signed transaction changing your vote for the given committee_member

**Parameters**

- voting_account: the name or id of the account who is voting with their shares
- committee_member: the name or id of the committee_member’ owner account
- approve: true if you wish to vote in favor of that committee_member, false to remove your vote in favor of that committee_member
- broadcast: true if you wish to broadcast the transaction

***vote_for_witness***

```
signed_transaction graphene::wallet::wallet_api::vote_for_witness(string voting_account, string witness, bool approve, bool broadcast = false)
```

Vote for a given witness. An account can publish a list of all witnesses they approve of. This command allows you to add or remove witnesses from this list. Each account's vote is weighted according to the number of shares of the core asset owned by that account at the time the votes are tallied. Note: *you cannot vote against a witness, you can only vote for the witness or not vote for the witness.*

**Return**

the signed transaction changing your vote for the given witness

**Parameters**

- voting_account: the name or id of the account who is voting with their shares
- witness: the name or id of the witness' owner account
- approve: true if you wish to vote in favor of that witness, false to remove your vote in favor of that witness
- broadcast: true if you wish to broadcast the transaction

***set_voting_proxy***

```
signed_transaction graphene::wallet::wallet_api::set_voting_proxy(string account_to_modify, optional<string> voting_account, bool broadcast = false)
```

Set the voting proxy for an account. If a user does not wish to take an active part in voting, they can choose to allow another account to vote their stake. Setting a vote proxy does not remove your previous votes from the blockchain, they remain there but are ignored. If you later null out your vote proxy, your previous votes will take effect again. This setting can be changed at any time.

**Return**

the signed transaction changing your vote proxy settings

**Parameters**

- account_to_modify: the name or id of the account to update
- voting_account: the name or id of an account authorized to vote account_to_modify's shares, or null to vote your own shares
- broadcast: true if you wish to broadcast the transaction

***set_desired_witness_and_committee_member_count***

```
signed_transaction
graphene::wallet::wallet_api::set_desired_witness_and_committee_member_count(
string account_to_modify, uint16_t desired_number_of_witnesses, uint16_t
desired_number_of_committee_members, bool broadcast = false)
```

Set your vote for the number of witnesses and committee_members in the system. Each account can voice their opinion on how many committee_members and how many witnesses there should be in the active committee_member/active witness list. These are independent of each other. You must vote your approval of at least as many committee_members or witnesses as you claim there should be (you can't say that there should be 20 committee_members but only vote for 10). There are maximum values for each set in the blockchain parameters (currently defaulting to 1001). This setting can be changed at any time. If your account has a voting proxy set, your preferences will be ignored.

**Return**
the signed transaction changing your vote proxy settings
**Parameters**

- account_to_modify: the name or id of the account to update
- number_of_committee_members: the number
- broadcast: true if you wish to broadcast the transaction

***propose_parameter_change***

```
signed_transaction
graphene::wallet::wallet_api::propose_parameter_change(const string
&proposing_account, fc::time_point_sec expiration_time, const variant_object
&changed_values, bool broadcast = false)
```

Creates a transaction to propose a parameter change. Multiple parameters can be specified if an atomic change is desired.
**Return**
the signed version of the transaction
**Parameters**

- proposing_account: The account paying the fee to propose the tx
- expiration_time: Timestamp specifying when the proposal will either take effect or expire.
- changed_values: The values to change; all other chain parameters are filled in with default values
- broadcast: true if you wish to broadcast the transaction

***propose_fee_change***

```
signed_transaction graphene::wallet::wallet_api::propose_fee_change(const
string &proposing_account, fc::time_point_sec expiration_time, const
variant_object &changed_values, bool broadcast = false)
```

Propose a fee change.
**Return**
the signed version of the transaction
**Parameters**

- proposing_account: The account paying the fee to propose the tx
- expiration_time: Timestamp specifying when the proposal will either take effect or expire.
- changed_values: Map of operation type to new fee. Operations may be specified by name or ID. The “scale” key changes the scale. All other operations will maintain current values.
- broadcast: true if you wish to broadcast the transaction

##### *9.6.1.11 Privacy Mode*

***set_key_label***

```
bool graphene::wallet::wallet_api::set_key_label(public_key_type key, string label)
```

These methods are used for stealth transfers This method can be used to set the label for a public key. Note: *No two keys can have the same label.*

**Return**

true if the label was set, otherwise false

***get_key_label***

```
string graphene::wallet::wallet_api::get_key_label(public_key_type key) const
```

***get_public_key***

```
public_key_type graphene::wallet::wallet_api::get_public_key(string label) const
```

**Return**

the public key associated with the given label

***get_blind_accounts***

```
map<string, public_key_type> graphene::wallet::wallet_api::get_blind_accounts() const
```

**Return**

all blind accounts

***get_my_blind_accounts***

```
map<string, public_key_type> graphene::wallet::wallet_api::get_my_blind_accounts() const
```

**Return**

all blind accounts for which this wallet has the private key

***get_blind_balances***

```
vector<asset> graphene::wallet::wallet_api::get_blind_balances(string key_or_label)
```

**Return**

the total balance of all blinded commitments that can be claimed by the given account key or label

***create_blind_account***

```
public_key_type graphene::wallet::wallet_api::create_blind_account(string
label, string brain_key)
```

Generates a new blind account for the given brain key and assigns it the given label.

***transfer_to_blind***

```
blind_confirmation graphene::wallet::wallet_api::transfer_to_blind(string
from_account_id_or_name, string asset_symbol, vector<pair<string, string>>
to_amounts, bool broadcast = false)
```

Transfers a public balance from to one or more blinded balances using a stealth transfer.

**Parameters**

- to_amounts: map from key or label to amount

***transfer_from_blind***

```
blind_confirmation graphene::wallet::wallet_api::transfer_from_blind(string
from_blind_account_key_or_label, string to_account_id_or_name, string amount,
string asset_symbol, bool broadcast = false)
```

Transfers funds from a set of blinded balances to a public account balance.

***blind_transfer***

```
blind_confirmation graphene::wallet::wallet_api::blind_transfer(string from_key_or_label, string
to_key_or_label, string amount, string symbol, bool broadcast = false)
```

Used to transfer from one set of blinded balances to another

***blind_history***

```
vector<blind_receipt> graphene::wallet::wallet_api::blind_history(string
key_or_account)
```

**Return**

all blind receipts to/form a particular account

***receive_blind_transfer***

```
blind_receipt graphene::wallet::wallet_api::receive_blind_transfer(string
confirmation_receipt, string opt_from, string opt_memo)
```

Given a confirmation receipt, this method will parse it for a blinded balance and confirm that it exists in the blockchain. If it exists then it will report the amount received and who sent it.

**Parameters**

- confirmation_receipt: a base58 encoded stealth confirmation
- opt_from: if not empty and the sender is a unknown public key, then the unknown public key will be given the label opt_from

##### *9.6.1.12 Blockchain Inspection*

***get_block***

```
optional<signed_block_with_info>
graphene::wallet::wallet_api::get_block(uint32_t num)
```

***get_account_count***

```
uint64_t graphene::wallet::wallet_api::get_account_count() const
```

Returns the number of accounts registered on the blockchain

***get_global_properties***

```
global_property_object graphene::wallet::wallet_api::get_global_properties()
const
```

Returns the block chain's slowly-changing settings. This object contains all of the properties of the blockchain that are fixed or that change only once per maintenance interval (daily) such as the current list of witnesses, committee_members, block interval, etc.
**See**
get_dynamic_global_properties() for frequently changing properties

***get_dynamic_global_properties***

```
dynamic_global_property_object
graphene::wallet::wallet_api::get_dynamic_global_properties() const
```

Returns the block chain's rapidly-changing properties. The returned object contains information that changes every block interval such as the head block number, the next witness, etc.
**See**
get_global_properties() for less-frequently changing properties

***get_object***

```
variant graphene::wallet::wallet_api::get_object(object_id_type id) const
```

Returns the blockchain object corresponding to the given id. This generic function can be used to retrieve any object from the blockchain that is assigned an ID. Certain types of objects have specialized convenience functions to return their objects e.g., assets have get_asset(), accounts have get_account(), but this function will work for any object.
**Return**
the requested object
**Parameters**

- id: the id of the object to return

#### *9.6.1.13 Transaction Builder*

***begin_builder_transaction***

```
transaction_handle_type
graphene::wallet::wallet_api::begin_builder_transaction()
```

***add_operation_to_builder_transaction***

```
void
graphene::wallet::wallet_api::add_operation_to_builder_transaction(transactio
n_handle_type transaction_handle, const operation &op)
```

***replace_operation_in_builder_transaction***

```
void
graphene::wallet::wallet_api::replace_operation_in_builder_transaction(transa
ction_handle_type handle, unsigned operation_index, const operation &new_op)
```

***set_fees_on_builder_transaction***

```
asset
graphene::wallet::wallet_api::set_fees_on_builder_transaction(transaction_han
dle_type handle, string fee_asset = GRAPHENE_SYMBOL)
```

***preview_builder_transaction***

```
transaction
graphene::wallet::wallet_api::preview_builder_transaction(transaction_handle_
type handle)
```

***sign_builder_transaction***

```
signed_transaction
graphene::wallet::wallet_api::sign_builder_transaction(transaction_handle_typ
e transaction_handle, bool broadcast = true)
```

***propose_builder_transaction***

```
signed_transaction
graphene::wallet::wallet_api::propose_builder_transaction(transaction_handle_
type handle, time_point_sec expiration = time_point::now()+fc::minutes(1),
uint32_t review_period_seconds = 0, bool broadcast = true)
```

***propose_builder_transaction2***

```
signed_transaction
graphene::wallet::wallet_api::propose_builder_transaction2(transaction_handle
```

```
_type handle, string account_name_or_id, time_point_sec expiration =
time_point::now()+fc::minutes(1), uint32_t review_period_seconds = 0, bool
broadcast = true)
```

***remove_builder_transaction***

```
void
graphene::wallet::wallet_api::remove_builder_transaction(transaction_handle_t
ype handle)
```

***serialize_transaction***

```
string graphene::wallet::wallet_api::serialize_transaction(signed_transaction
tx) const
```

Converts a signed_transaction in JSON form to its binary representation.

**Return**

the binary form of the transaction. It will not be hex encoded, this returns a raw string that may have null characters embedded in it

**Parameters**

- tx: the transaction to serialize

***sign_transaction***

```
signed_transaction
graphene::wallet::wallet_api::sign_transaction(signed_transaction tx, bool
broadcast = false)
```

Signs a transaction. Given a fully-formed transaction that is only lacking signatures, this signs the transaction with the necessary keys and optionally broadcasts the transaction

**Return**

the signed version of the transaction

**Parameters**

- tx: the unsigned transaction
- broadcast: true if you wish to broadcast the transaction

***sign_transaction_with_keys***

```
signed_transaction
graphene::wallet::wallet_api::sign_transaction(signed_transaction tx,
std::vector<string> wif_keys, bool broadcast = false)
```

Signs a transaction. Given a fully-formed transaction that is only lacking signatures and a list of keys, this signs the transaction and optionally broadcasts the transaction

**Return**

the signed version of the transaction

**Parameters**

- tx: the unsigned transaction
- wif_keys: list of keys
- broadcast: true if you wish to broadcast the transaction

***get_prototype_operation***

```
operation graphene::wallet::wallet_api::get_prototype_operation(string
operation_type)
```

Returns an uninitialized object representing a given blockchain operation. This returns a default-initialized object of the given type; it can be used during early development of the wallet when we don't yet have custom commands for creating all of the operations the blockchain supports. Any operation the blockchain supports can be created using the transaction builder's add_operation_to_builder_transaction() , but to do that from the CLI you need to know what the JSON form of the operation looks like. This will give you a template you can fill in. It's better than nothing.

**Return**

a default-constructed operation of the given type

**Parameters**

- operation_type: the type of operation to return, must be one of the operations defined in graphene/chain/operations.hpp (e.g., "global_parameters_update_operation")

# 10 Appendix B: Operations

Here is a full list of our operations and their details.

***transfer_operation***

Transfer an amount from one account to another. Fees are paid by the "from" account

**Parameters**

- asset fee
- account_id_type from: Account to transfer asset from.
- account_id_type to: Account to transfer asset to.
- asset amount: The amount of asset to transfer from from to to.
- optional<memo_data> memo: User provided data encrypted to the memo key of the "to" account.
- extensions_type extensions

**Precondition**

- amount.amount > 0
- fee.amount >= 0
- from != to

**Postcondition**

- from account's balance will be reduced by fee and amount
- to account's balance will be increased by amount

***limit_order_create_operation***

instructs the blockchain to attempt to sell one asset for another. The blockchain will atempt to sell amount_to_sell.asset_id for as much min_to_receive.asset_id as possible. The fee will be paid by the seller's account. Market fees will apply as specified by the issuer of both the selling asset and the receiving asset as a percentage of the amount exchanged. If either the selling asset or the receiving asset is white list restricted, the order will only be created if the seller is on the white list of the restricted asset type. Market orders are matched in the order they are included in the blockchain.

**Parameters**

- asset fee
- account_id_type seller
- asset amount_to_sell
- asset min_to_receive
- share_type reserved_amount
- optional<account_id_type> account_to_credit
- time_point_sec expiration = time_point_sec::maximum()
- bool fill_or_kill = false: If this flag is set the entire order must be filled or the operation is rejected
- extensions_type extensions

***limit_order_cancel_operation***

Used to cancel an existing limit order. Both fee_pay_account and the account to receive the proceeds must be the same as order->seller.

**Returns**

the amount actually refunded

**Parameters**

- asset fee
- limit_order_id_type order
- account_id_type fee_paying_account
- extensions_type extensions

***call_order_update_operation***

This operation can be used to add collateral, cover, and adjust the margin call price for a particular user. For prediction markets the collateral and debt must always be equal. This operation will fail if it would trigger a margin call that couldn't be filled. If the margin call hits the call price limit then it will fail if the call price is above the settlement price.
Note: this operation can be used to force a market order using the collateral without requiring outside funds.

**Parameters**

- asset fee
- account_id_type funding_account: pays fee, collateral, and cover
- asset delta_collateral: the amount of collateral to add to the margin position
- asset delta_debt: the amount of the debt to be paid off, may be negative to issue new debt
- extensions_type extensions

***account_create_operation***

Create an account

**Parameters**

- asset fee
- uint8_t kind: The account kind: wallet, vault, special...
- account_id_type registrar: This MUST BE the current registrar chain authority.
- account_id_type referrer: This account receives a portion of the fee split between registrar and referrer. Must be a member.
- uint16_t referrer_percent = 0
- string name
- authority owner
- authority active
- account_options options
- extension<ext> extensions

***account_update_operation***

Update an existing account. This operation is used to update an existing account. It can be used to update the authorities, or adjust the options on the account. See account_object::options_type for the options which may be updated.

**Parameters**

- asset fee
- account_id_type account: The account to update.

- optional<authority> owner: New owner authority. If set, this operation requires owner authority to execute.
- optional<authority> active: New active authority. This can be updated by the current active authority.
- optional<account_options> new_options: New account options.
- extension<ext_account_update_operation> extensions

***account_whitelist_operation***

This operation is used to whitelist and blacklist accounts, primarily for transacting in whitelisted assets.
Accounts can freely specify opinions about other accounts, in the form of either whitelisting or blacklisting them. This information is used in chain validation only to determine whether an account is authorized to transact in an asset type which enforces a whitelist, but third parties can use this information for other uses as well, as long as it does not conflict with the use of whitelisted assets.
An asset which enforces a whitelist specifies a list of accounts to maintain its whitelist, and a list of accounts to maintain its blacklist. In order for a given account A to hold and transact in a whitelisted asset S, A must be whitelisted by at least one of S's whitelist_authorities and blacklisted by none of S's blacklist_authorities. If A receives a balance of S, and is later removed from the whitelist(s) which allowed it to hold S, or added to any blacklist S specifies as authoritative, A's balance of S will be frozen until A's authorization is reinstated.
This operation requires authorizing_account's signature, but not account_to_list's. The fee is paid by authorizing_account.
**Parameters**

- asset fee: Paid by authorizing_account.
- account_id_type authorizing_account: The account which is specifying an opinion of another account.
- account_id_type account_to_list: The account being opined about.
- uint8_t new_listing = no_listing
- extensions_type extensions

***account_upgrade_operation***

Manage an account's membership status. This operation is used to upgrade an account to a member, or renew its subscription. If an account which is an unexpired annual subscription member publishes this operation with upgrade_to_lifetime_member set to false, the account's membership expiration date will be pushed backward one year. If a basic account publishes it with upgrade_to_lifetime_member set to false, the account will be upgraded to a subscription member with an expiration date one year after the processing time of this operation.
Any account may use this operation to become a lifetime member by setting upgrade_to_lifetime_member to true. Once an account has become a lifetime member, it may not use this operation anymore.
**Parameters**

- asset fee
- account_id_type account_to_upgrade: The account to upgrade; must not already be a lifetime member.
- bool upgrade_to_lifetime_member = false: If true, the account will be upgraded to a lifetime member; otherwise, it will add a year to the subscription.

- extensions_type extensions

***account_transfer_operation***

Transfers the account to another account while clearing the white list. In theory an account can be transferred by simply updating the authorities, but that kind of transfer lacks semantic meaning and is more often done to rotate keys without transferring ownership. This operation is used to indicate the legal transfer of title to this account and a break in the operation history.
The account_id's owner/active/voting/memo authority should be set to new_owner
This operation will clear the account's whitelist statuses, but not the blacklist statuses.
**Parameters**

- asset fee
- account_id_type account_id
- account_id_type new_owner
- extensions_type extensions

***asset_create_operation***

Creates asset
**Parameters**

- asset fee
- account_id_type issuer: This account must sign and pay the fee for this operation. Later, this account may update the asset.
- string symbol: The ticker symbol of this asset.
- uint8_t precision = 0: Number of digits to the right of decimal point, must be less than or equal to 12.
- asset_options common_options
- optional<bitasset_options> bitasset_opts
- bool is_prediction_market = false: For BitAssets, set this to true if the asset implements a Prediction Market; false otherwise.
- extensions_type extensions

***asset_update_operation***

Update options common to all assets. There are a number of options which all assets in the network use. These options are enumerated in the asset_options struct. This operation is used to update these options for an existing asset.
Note: This operation cannot be used to update BitAsset-specific options. For these options, use asset_update_bitasset_operation instead.
**Parameters**

- asset fee
- account_id_type issuer
- asset_id_type asset_to_update
- optional<account_id_type> new_issuer: If the asset is to be given a new issuer, specify his ID here.
- asset_options new_options
- extensions_type extensions

**Precondition**

- issuer SHALL be an existing account and MUST match asset_object::issuer on asset_to_update
- fee SHALL be nonnegative, and issuer MUST have a sufficient balance to pay it
- new_options SHALL be internally consistent, as verified by validate()

**Postcondition**

- asset_to_update will have options matching those of new_options

### *asset_update_bitasset_operation*

Update options specific to BitAssets. BitAssets have some options which are not relevant to other asset types. This operation is used to update those options an an existing BitAsset.

**Parameters**

- asset fee
- account_id_type issuer
- asset_id_type asset_to_update
- bitasset_options new_options
- extensions_type extensions

**Precondition**

- issuer MUST be an existing account and MUST match asset_object::issuer on asset_to_update
- asset_to_update MUST be a BitAsset, i.e. asset_object::is_market_issued() returns true
- fee MUST be nonnegative, and issuer MUST have a sufficient balance to pay it
- new_options SHALL be internally consistent, as verified by validate()

**Postcondition**

- asset_to_update will have BitAsset-specific options matching those of new_options

### *asset_update_feed_producers_operation*

Update the set of feed-producing accounts for a BitAsset. BitAssets have price feeds selected by taking the median values of recommendations from a set of feed producers. This operation is used to specify which accounts may produce feeds for a given BitAsset.

**Parameters**

- asset fee
- account_id_type issuer
- asset_id_type asset_to_update
- flat_set<account_id_type> new_feed_producers
- extensions_type extensions

**Precondition**

- issuer MUST be an existing account, and MUST match asset_object::issuer on asset_to_update
- issuer MUST NOT be the committee account
- asset_to_update MUST be a BitAsset, i.e. asset_object::is_market_issued() returns true
- fee MUST be nonnegative, and issuer MUST have a sufficient balance to pay it
- Cardinality of new_feed_producers MUST NOT exceed chain_parameters::maximum_asset_feed_publishers

**Postcondition**

- asset_to_update will have a set of feed producers matching new_feed_producers
- All valid feeds supplied by feed producers in new_feed_producers, which were already feed producers prior to execution of this operation, will be preserved

***asset_issue_operation***

Issues asset to an account

**Parameters**

- asset fee
- account_id_type issuer: Must be asset_to_issue->asset_id->issuer.
- asset asset_to_issue
- account_id_type issue_to_account
- optional<memo_data> memo
- extensions_type extensions

***asset_reserve_operation***

used to take an asset out of circulation, returning to the issuer. Note: You cannot use this operation on market-issued assets.

**Parameters**

- asset fee
- account_id_type payer
- asset amount_to_reserve
- extensions_type extensions

***asset_fund_fee_pool_operation***

**Parameters**

- asset fee: core asset
- account_id_type from_account
- asset_id_type asset_id
- share_type amount: core asset
- extensions_type extensions

***asset_settle_operation***

Schedules a market-issued asset for automatic settlement. Holders of market-issued assests may request a forced settlement for some amount of their asset. This means that the specified sum will be locked by the chain and held for the settlement period, after which time the chain will choose a margin posision holder and buy the settled asset using the margin's collateral. The price of this sale will be based on the feed price for the market-issued asset being settled. The exact settlement price will be the feed price at the time of settlement with an offset in favor of the margin position, where the offset is a blockchain parameter set in the global_property_object.

The fee is paid by account, and account must authorize this operation

**Parameters**

- asset fee
- account_id_type account: Account requesting the force settlement. This account pays the fee.
- asset amount: Amount of asset to force settle. This must be a market-issued asset.
- extensions_type extensions

***asset_global_settle_operation***

allows global settling of bitassets (black swan or prediction markets). In order to use this operation, asset_to_settle must have the global_settle flag set. When this operation is executed all balances are converted into the backing asset at the settle_price and all open margin positions are called at the settle price. If this asset is used as backing for other bitassets, those bitassets will be force settled at their current feed price.

**Parameters**

- asset fee
- account_id_type issuer: must equal asset_to_settle->issuer
- asset_id_type asset_to_settle
- price settle_price
- extensions_type extensions

***asset_publish_feed_operation***

Publish price feeds for market-issued assets. Price feed providers use this operation to publish their price feeds for market-issued assets. A price feed is used to tune the market for a particular market-issued asset. For each value in the feed, the median across all committee_member feeds for that asset is calculated and the market for the asset is configured with the median of that value.

The feed in the operation contains three prices: a call price limit, a short price limit, and a settlement price. The call limit price is structured as (collateral asset) / (debt asset) and the short limit price is structured as (asset for sale) / (collateral asset). Note that the asset IDs are opposite to each other, so if we're publishing a feed for USD, the call limit price will be CORE/USD and the short limit price will be USD/CORE. The settlement price may be flipped either direction, as long as it is a ratio between the market-issued asset and its collateral.

**Parameters**

- asset fee: paid for by publisher
- account_id_type publisher
- asset_id_type asset_id: asset for which the feed is published
- price_feed feed
- extensions_type extensions

***witness_create_operation***

Create a witness object, as a bid to hold a witness position on the network. Accounts which wish to become witnesses may use this operation to create a witness object which stakeholders may vote on to approve its position as a witness.

**Parameters**

- asset fee
- account_id_type witness_account: The account which owns the witness. This account pays the fee for this operation
- string url
- public_key_type block_signing_key

***witness_update_operation***

Update a witness object's URL and block signing key
**Parameters**

- asset fee: paid for by publisher
- witness_id_type witness: The witness object to update.
- account_id_type witness_account: The account which owns the witness. This account pays the fee for this operation.
- optional<string> new_url: The new URL.
- optional<public_key_type> new_signing_key: The new block signing key.

***proposal_create_operation***

The proposal_create_operation creates a transaction proposal, for use in multi-sig scenarios. Creates a transaction proposal. The operations which compose the transaction are listed in order in proposed_ops, and expiration_time specifies the time by which the proposal must be accepted or it will fail permanently. The expiration_time cannot be farther in the future than the maximum expiration time set in the global properties object.
**Parameters**

- asset fee
- account_id_type fee_paying_account
- vector<op_wrapper> proposed_ops
- time_point_sec expiration_time
- optional<uint32_t> review_period_seconds
- extensions_type extensions

***proposal_update_operation***

This operation allows accounts to add or revoke approval of a proposed transaction. Signatures sufficient to satisfy the authority of each account in approvals are required on the transaction containing this operation.
If an account with a multi-signature authority is listed in approvals_to_add or approvals_to_remove, either all required signatures to satisfy that account's authority must be provided in the transaction containing this operation, or a secondary proposal must be created which contains this operation. Note: If the proposal requires only an account's active authority, the account must not update adding its owner authority's approval. This is considered an error. An owner approval may only be added if the proposal requires the owner's authority.
If an account's owner and active authority are both required, only the owner authority may approve. An attempt to add or remove active authority approval to such a proposal will fail.
**Parameters**

- account_id_type fee_paying_account
- asset fee
- proposal_id_type proposal
- flat_set<account_id_type> active_approvals_to_add
- flat_set<account_id_type> active_approvals_to_remove
- flat_set<account_id_type> owner_approvals_to_add
- flat_set<account_id_type> owner_approvals_to_remove
- flat_set<public_key_type> key_approvals_to_add
- flat_set<public_key_type> key_approvals_to_remove

- extensions_type extensions

*proposal_delete_operation*

The proposal_delete_operation deletes an existing transaction proposal. This operation allows the early veto of a proposed transaction. It may be used by any account which is a required authority on the proposed transaction, when that account's holder feels the proposal is ill-advised and he decides he will never approve of it and wishes to put an end to all discussion of the issue. Because he is a required authority, he could simply refuse to add his approval, but this would leave the topic open for debate until the proposal expires. Using this operation, he can prevent any further breath from being wasted on such an absurd proposal.

**Parameters**

- account_id_type fee_paying_account
- bool using_owner_authority = false
- asset fee
- proposal_id_type proposal
- extensions_type extensions

*withdraw_permission_create_operation*

Create a new withdrawal permission. This operation creates a withdrawal permission, which allows some authorized account to withdraw from an authorizing account. This operation is primarily useful for scheduling recurring payments.
Withdrawal permissions define withdrawal periods, which is a span of time during which the authorized account may make a withdrawal. Any number of withdrawals may be made so long as the total amount withdrawn per period does not exceed the limit for any given period.
Withdrawal permissions authorize only a specific pairing, i.e. a permission only authorizes one specified authorized account to withdraw from one specified authorizing account. Withdrawals are limited and may not exceed the withdrawal limit. The withdrawal must be made in the same asset as the limit; attempts with withdraw any other asset type will be rejected.
The fee for this operation is paid by withdraw_from_account, and this account is required to authorize this operation.

**Parameters**

- asset fee
- account_id_type withdraw_from_account: The account authorizing withdrawals from its balances
- account_id_type authorized_account: The account authorized to make withdrawals from withdraw_from_account.
- asset withdrawal_limit: The maximum amount authorized_account is allowed to withdraw in a given withdrawal period.
- uint32_t withdrawal_period_sec = 0: Length of the withdrawal period in seconds.
- uint32_t periods_until_expiration = 0: The number of withdrawal periods this permission is valid for.
- time_point_sec period_start_time: Time at which the first withdrawal period begins; must be in the future.

***withdraw_permission_update_operation***

Update an existing withdraw permission. This operation is used to update the settings for an existing withdrawal permission. The accounts to withdraw to and from may never be updated. The fields which may be updated are the withdrawal limit (both amount and asset type may be updated), the withdrawal period length, the remaining number of periods until expiration, and the starting time of the new period.
Fee is paid by withdraw_from_account, which is required to authorize this operation
**Parameters**

- asset fee
- account_id_type withdraw_from_account: This account pays the fee. Must match permission_to_update->withdraw_from_account
- account_id_type authorized_account: The account authorized to make withdrawals. Must match permission_to_update->authorized_account
- withdraw_permission_id_type permission_to_update: ID of the permission which is being updated
- asset withdrawal_limit: New maximum amount the withdrawer is allowed to charge per withdrawal period
- uint32_t withdrawal_period_sec = 0: New length of the period between withdrawals.
- time_point_sec period_start_time: New beginning of the next withdrawal period; must be in the future
- uint32_t periods_until_expiration = 0: The new number of withdrawal periods for which this permission will be valid

***withdraw_permission_claim_operation***

Withdraw from an account which has published a withdrawal permission. This operation is used to withdraw from an account which has authorized such a withdrawal. It may be executed at most once per withdrawal period for the given permission. On execution, amount_to_withdraw is transferred from withdraw_from_account to withdraw_to_account, assuming amount_to_withdraw is within the withdrawal limit. The withdrawal permission will be updated to note that the withdrawal for the current period has occurred, and further withdrawals will not be permitted until the next withdrawal period, assuming the permission has not expired. This operation may be executed at any time within the current withdrawal period.
Fee is paid by withdraw_to_account, which is required to authorize this operation
**Parameters**

- asset fee: Paid by withdraw_to_account
- withdraw_permission_id_type withdraw_permission: ID of the permission authorizing this withdrawal
- account_id_type withdraw_from_account: Must match withdraw_permission->withdraw_from_account
- account_id_type withdraw_to_account: Must match withdraw_permision->authorized_account
- asset amount_to_withdraw: Amount to withdraw. Must not exceed withdraw_permission->withdrawal_limit.
- optional<memo_data> memo: Memo for withdraw_from_account. Should generally be encrypted with withdraw_from_account->memo_key.

***withdraw_permission_delete_operation***

Delete an existing withdrawal permission. This operation cancels a withdrawal permission, thus preventing any future withdrawals using that permission. Fee is paid by withdraw_from_account, which is required to authorize this operation

**Parameters**

- asset fee
- account_id_type withdraw_from_account: Must match withdrawal_permission->withdraw_from_account. This account pays the fee
- account_id_type authorized_account: The account previously authorized to make withdrawals. Must match withdrawal_permission->authorized_account.

- withdraw_permission_id_type withdrawal_permission: ID of the permission to be revoked

***committee_member_create_operation***

Create a committee_member object, as a bid to hold a committee_member seat on the network. Accounts which wish to become committee_members may use this operation to create a committee_member object which stakeholders may vote on to approve its position as a committee_member.

**Parameters**

- asset fee
- account_id_type committee_member_account: The account which owns the committee_member. This account pays the fee for this operation
- string url

***committee_member_update_operation***

Update a committee_member object. Currently the only field which can be updated is the url field.

**Parameters**

- asset fee
- committee_member_id_type committee_member: The committee member to update
- account_id_type committee_member_account: The account which owns the committee_member. This account pays the fee for this operation
- optional<string> new_url

***committee_member_update_global_parameters_operation***

Used by committee_members to update the global parameters of the blockchain. This operation allows the committee_members to update the global parameters on the blockchain. These control various tunable aspects of the chain, including block and maintenance intervals, maximum data sizes, the fees charged by the network, etc.
This operation may only be used in a proposed transaction, and a proposed transaction which contains this operation must have a review period specified in the current global parameters before it may be accepted.

**Parameters**

- asset fee
- chain_parameters new_parameters

***vesting_balance_create_operation***

Create a vesting balance. The chain allows a user to create a vesting balance. Normally, vesting balances are created automatically as part of cashback and worker operations. This operation allows vesting balances to be created manually as well.
Manual creation of vesting balances can be used by a stakeholder to publicly demonstrate that they are committed to the chain. It can also be used as a building block to create transactions that function like public debt. Finally, it is useful for testing vesting balance functionality.
**Returns**
ID of newly created vesting_balance_object
**Parameters**

- asset fee
- account_id_type creator: Who provides funds initially
- account_id_type owner: Who is able to withdraw the balance
- asset amount
- vesting_policy_initializer policy

***vesting_balance_withdraw_operation***

Withdraw from a vesting balance. Withdrawal from a not-completely-mature vesting balance will result in paying fees
**Parameters**

- asset fee
- vesting_balance_id_type vesting_balance
- account_id_type owner: Must be vesting_balance.owner
- asset amount

***worker_create_operation***

Create a new worker object
**Parameters**

- asset fee
- account_id_type owner
- time_point_sec work_begin_date
- time_point_sec work_end_date
- share_type daily_pay
- string name
- string url
- worker_initializer initializer: This should be set to the initializer appropriate for the type of worker to be created

***custom_operation***

provides a generic way to add higher level protocols on top of witness consensus. There is no validation for this operation other than that required auths are valid and a fee is paid that is appropriate for the data contained.

**Parameters**

- asset fee
- account_id_type payer
- flat_set<account_id_type> required_auths
- uint16_t id = 0
- vector<char> data

***assert_operation***

assert that some conditions are true. This operation performs no changes to the database state, but can but used to verify pre or post conditions for other operations.

**Parameters**

- asset fee
- account_id_type fee_paying_account
- vector<predicate> predicates
- flat_set<account_id_type> required_auths
- extensions_type extensions

***balance_claim_operation***

Claim a balance in a balanc_object. This operation is used to claim the balance in a given balance_object. If the balance object contains a vesting balance, total_claimed must not exceed balance_object::available at the time of evaluation. If the object contains a non-vesting balance, total_claimed must be the full balance of the object.

**Parameters**

- asset fee
- account_id_type deposit_to_account
- balance_id_type balance_to_claim
- public_key_type balance_owner_key
- asset total_claimed

***override_transfer_operation***

Allows the issuer of an asset to transfer an asset from any account to any account if they have override_authority.

**Parameters**

- asset fee
- account_id_type issuer
- account_id_type from: Account to transfer asset from
- account_id_type to: Account to transfer asset to
- asset amount: The amount of asset to transfer from from to to
- optional<memo_data> memo: User provided data encrypted to the memo key of the "to" account

- extensions_type extensions

**Precondition**

- amount.asset_id->issuer == issuer
- issuer != from because this is pointless, use a normal transfer operation

***transfer_to_blind_operation***

Converts public account balance to a blinded or stealth balance.

**Parameters**

- asset fee
- asset amount
- account_id_type from
- blind_factor_type blinding_factor
- vector<blind_output> outputs

***blind_transfer_operation***

Transfers from blind to blind. There are two ways to transfer value while maintaining privacy:

- account to account with amount kept secret
- stealth transfers with amount sender/receiver kept secret

When doing account to account transfers, everyone with access to the memo key can see the amounts, but they will not have access to the funds.
When using stealth transfers the same key is used for control and reading the memo.
This operation is more expensive than a normal transfer and has a fee proportional to the size of the operation.
All assets in a blind transfer must be of the same type: fee.asset_id The fee_payer is the temp account and can be funded from the blinded values.
Using this operation you can transfer from an account and/or blinded balances to an account and/or blinded balances.
Stealth Transfers:
Assuming Receiver has key pair R,r and has shared public key R with Sender Assuming Sender has key pair S,s Generate one time key pair O,o as s.child(nonce) where nonce can be inferred from transaction Calculate secret V = o*R blinding_factor = sha256(V) memo is encrypted via aes of V owner = R.child(sha256(blinding_factor))
Sender gives Receiver output ID to complete the payment.
This process can also be used to send money to a cold wallet without having to pre-register any accounts.
Outputs are assigned the same IDs as the inputs until no more input IDs are available, in which case a the return value will be the first ID allocated for an output. Additional output IDs are allocated sequentially thereafter. If there are fewer outputs than inputs then the input IDs are freed and never used again.

**Parameters**

- asset fee
- vector<blind_input> inputs
- vector<blind_output> outputs

***transfer_from_blind_operation***

Converts blinded/stealth balance to a public account balance.
**Parameters**

- asset fee
- asset amount
- account_id_type to
- blind_factor_type blinding_factor
- vector<blind_input> inputs

***asset_claim_fees_operation***

used to transfer accumulated fees back to the issuer's balance
**Parameters**

- asset fee
- account_id_type issuer
- asset amount_to_claim
- extensions_type extensions

**Preconditions**

- amount_to_claim.asset_id->issuer must == issuer

***board_update_chain_authority_operation***

Used by board members to update chain authorities. This operation allows the committee members to update a chain authority in the global properties object on the blockchain. The kind must match the number assigned to said authority. This operation may only be used in a proposed transaction, and a proposed transaction which contains this operation must have a review period specified in the current global parameters before it may be accepted.
**Parameters**

- asset fee
- string kind
- account_id_type account
- account_id_type committee_member_account

***update_queue_parameters_operation***

**Parameters**

- asset fee
- account_id_type issuer
- optional<bool> enable_dascoin_queue
- optional<uint32_t> reward_interval_time_seconds
- optional<share_type> dascoin_reward_amount
- extensions_type extensions

***create_license_type_operation***

Create a new type of license. Must be signed by the current license_administration authority.
**Parameters**

- asset fee
- account_id_type admin
- string name
- share_type amount
- string kind
- upgrade_multiplier_type balance_multipliers
- upgrade_multiplier_type requeue_multipliers
- upgrade_multiplier_type return_multipliers
- share_type eur_limit

***issue_license_operation***

Request a license to be granted an account. Grant a license to an account. This operation must be signed by the current license_issuer authority.

**Parameters**

- asset fee
- account_id_type issuer
- account_id_type account
- license_type_id_type license
- share_type bonus_percentage
- frequency_type frequency_lock
- time_point_sec activated_at
- extensions_type extensions

***tether_accounts_operation***

tethers a vault and wallet account together.

**Parameters**

- asset fee
- account_id_type wallet_account
- account_id_type vault_account
- extensions_type extensions

***asset_create_issue_request_operation***

For dual authority issued assets, create an asset issue request that can be denied by the asset authenticator. Note: You cannot use this operation on single issuer assets.

**Parameters**

- asset fee
- account_id_type issuer
- account_id_type receiver
- share_type amount
- asset_id_type asset_id
- share_type reserved_amount
- string unique_id
- string comment
- extensions_type extensions

***asset_deny_issue_request_operation***

As the asset authenticator on a dual authentication issuing asset, deny an asset issue request.
**Parameters**

- asset fee
- account_id_type authenticator
- issue_asset_request_id_type request
- extensions_type extensions

***wire_out_operation***

**Parameters**

- asset fee
- account_id_type account
- asset asset_to_wire
- string memo
- extensions_type extensions

***wire_out_complete_operation***

**Parameters**

- asset fee
- account_id_type wire_out_handler
- wire_out_holder_id_type holder_object_id
- extensions_type extensions

***wire_out_reject_operation***

**Parameters**

- asset fee
- account_id_type wire_out_handler
- wire_out_holder_id_type holder_object_id
- extensions_type extensions

***transfer_vault_to_wallet_operation***

Transfers assets from a tethered vault to its parent wallet, with limits enforced.
**Parameters**

- asset fee
- account_id_type from_vault
- account_id_type to_wallet
- asset asset_to_transfer
- share_type reserved_to_transfer
- extensions_type extensions

***transfer_wallet_to_vault_operation***

Transfers assets from a tethered vault to its parent wallet. NO LIMITS are enforced.

**Parameters**

- asset fee
- account_id_type from_wallet
- account_id_type to_vault
- asset asset_to_transfer
- share_type reserved_to_transfer
- extensions_type extensions

***submit_reserve_cycles_to_queue_operation***

Request to issue cycles to an account. An authorized cycle issuing authority can request to issue a certain amount of cycles. An independent authorized cycle authentication authority must inspect and approve this request.

**Parameters**

- asset fee
- account_id_type issuer
- account_id_type account
- share_type amount
- frequency_type frequency_lock
- string comment
- extensions_type extensions

***submit_cycles_to_queue_operation***

Submit cycles to the DasCoin distribution queue. A user can submit their cycles to the dascoin distribution queue where they await to be minted.

- asset fee
- account_id_type account
- share_type amount
- frequency_type frequency
- string comment
- extensions_type extensions

***change_public_keys_operation***

**Parameters**

- asset fee
- account_id_type account
- optional<authority> active
- optional<authority> owner
- extensions_type extensions

***update_global_frequency_operation***

**Parameters**

- asset fee
- account_id_type authority
- frequency_type frequency
- string comment
- extensions_type extensions

***issue_free_cycles_operation***

**Parameters**

- asset fee
- account_id_type authority
- uint8_t origin
- account_id_type account
- share_type amount
- string comment
- extensions_type extensions

***edit_license_type_operation***

**Parameters**

- asset fee
- account_id_type authority
- license_type_id_type license_type
- optional<string> name
- optional<share_type> amount
- optional<share_type> eur_limit

***update_euro_limit_operation***

Allows the authority to disable or enable the euro limit to an account.

**Parameters**

- asset fee
- account_id_type authority
- account_id_type account
- bool disable_limit
- optional<share_type> eur_limit
- string comment
- extensions_type extensions

***submit_cycles_to_queue_by_license_operation***

Submit cycles by license to the DasCoin distribution queue. A user can submit their cycles to the dascoin distribution queue where they await to be minted.

**Parameters**

- asset fee

- account_id_type account
- share_type amount
- license_type_id_type license_type
- frequency_type frequency_lock
- string comment
- extensions_type extensions

***update_license_operation***

Update a license issued to an account. Update a license issued to an account. This operation must be signed by the current license_issuer authority.

**Parameters**

- asset fee
- account_id_type authority
- account_id_type account
- license_type_id_type license
- optional<share_type> bonus_percentage
- optional<frequency_type> frequency_lock
- optional<time_point_sec> activated_at
- extensions_type extensions

***Issue_cycles_to_license_operation***

**Parameters**

- asset fee
- account_id_type authority
- account_id_type account
- license_type_id_type license
- share_type amount
- string origin
- string comment
- extensions_type extensions

***remove_root_authority_operation***

**Parameters**

- asset fee
- account_id_type root_account: Root account whose authority we will revoke. This account pays the fee for this operation.
- string comment

***create_witness_operation***

**Parameters**

- asset fee
- account_id_type authority: Root account authority. This account pays the fee for this operation.

- account_id_type witness_account: Existing account that we want to promote into a master node candidate.
- public_key_type block_signing_key: Public key that is used for signing blocks.
- string url
- string comment

***update_witness_operation***

**Parameters**
- asset fee
- witness_id_type witness
- account_id_type authority: Root account authority. This account pays the fee for this operation
- optional<account_id_type> witness_account: Existing account that we want to promote into a master node candidate.
- optional<public_key_type> block_signing_key: Public key that is used for signing blocks.
- optional<string> url
- optional<string> comment

***remove_witness_operation***

**Parameters**
- asset fee
- witness_id_type witness
- account_id_type authority: Root account authority. This account pays the fee for this operation
- optional<string> comment

***activate_witness_operation***

**Parameters**
- asset fee
- witness_id_type witness
- account_id_type authority: Root account authority. This account pays the fee for this operation
- optional<string> comment

***deactivate_witness_operation***

**Parameters**
- asset fee
- witness_id_type witness
- account_id_type authority: Root account authority. This account pays the fee for this operation
- optional<string> comment

***create_upgrade_event_operation***

**Parameters**

- asset fee
- account_id_type upgrade_creator
- time_point_sec execution_time
- optional<time_point_sec> cutoff_time
- vector<time_point_sec> subsequent_execution_times
- string comment
- extensions_type extensions

***update_upgrade_event_operation***

**Parameters**

- asset fee
- account_id_type upgrade_creator
- upgrade_event_id_type upgrade_event_id
- optional<time_point_sec> execution_time
- optional<time_point_sec> cutoff_time
- optional<vector<time_point_sec>> subsequent_execution_times
- optional<string> comment
- extensions_type extensions

***delete_upgrade_event_operation***

**Parameters**

- asset fee
- account_id_type upgrade_creator
- upgrade_event_id_type upgrade_event_id
- extensions_type extensions

***remove_vault_limit_operation***

Allows the authority to remove limits on all vaults in system

**Parameters**

- asset fee
- account_id_type authority
- string comment
- extensions_type extensions

***change_operation_fee_operation***

Request to change fee for particular operation

**Parameters**

- asset fee
- account_id_type issuer
- uint64_t new_fee
- unsigned op_num

- string comment
- extensions_type extensions

***change_fee_pool_account_operation***

Request to change fee for particular operation
**Parameters**
- asset fee
- account_id_type issuer
- account_id_type fee_pool_account_id
- string comment
- extensions_type extensions

***purchase_cycle_asset_operation***

**Parameters**
- asset fee
- account_id_type wallet_id
- share_type amount
- frequency_type frequency
- share_type expected_amount
- extensions_type extensions

***transfer_cycles_from_licence_to_wallet_operation***

**Parameters**
- asset fee
- account_id_type vault_id
- license_type_id_type license_id
- share_type amount
- account_id_type wallet_id
- extensions_type extensions

***wire_out_with_fee_operation***

**Parameters**
- asset fee
- account_id_type account
- asset asset_to_wire
- string currency_of_choice
- string to_address
- string memo
- extensions_type extensions

***wire_out_with_fee_complete_operation***

**Parameters**
- asset fee
- account_id_type wire_out_handler
- wire_out_with_fee_holder_id_type holder_object_id
- extensions_type extensions

***wire_out_with_fee_reject_operation***

**Parameters**
- asset fee
- account_id_type wire_out_handler
- wire_out_with_fee_holder_id_type holder_object_id
- extensions_type extensions

***set_starting_cycle_asset_amount_operation***

Sets global value for starting amount of cycles on new accounts. Changes the value of global property starting_cycle_asset_amount, that represents a number of cycles that is given to each new wallet or custodian account.

**Parameters**
- asset fee
- account_id_type issuer: Operation issuer, must be root authority
- uint32_t new_amount = DASCOIN_DEFAULT_STARTING_CYCLE_ASSET_AMOUNT: A value to set the amount to
- extensions_type extensions;

DasCoin™