
Graphene Documentation

Release

Fabian Schuh

Jan 26, 2018

Contents

1 Recent Updates	3
2 Blockchain Specific Guides	5
2.1 BitShares 2.0	5
2.2 MUSE	137
3 Integration Guide	155
3.1 Integration Guide	155
4 API Guide	239
5 Development	241
5.1 Development Guide	241
5.2 Testnets	341
6 Contribute	361
6.1 How to Contribute	361

Caution: Please note that the content of this web page is out-dated and currently under review by the BitShares Blockchain Foundation. If you need factual information regarding the BitShares Blockchain, or BTS, please contact spokesperson@bitshares.foundation.

The developers of BitShares formed Cryptonomex to monetize the technology, experience, reputation and good will they accumulated during their first two years of development and operations. Much of that technology is embodied in Graphene™, an industrial strength software platform for deploying third generation cryptographically secure decentralized ledgers known as block chains.

Graphene based systems have orders of magnitude better performance than first-generation Bitcoin-derived systems or even the second generation “Bitcoin 2.0” systems that constitute our current closest competitors. Graphene based systems go beyond mere “checkbook” style payments to offer a broad range of financial services distinguished by their transparency and inherent incorruptibility.

This page documents the **Graphene** technology built by [Cryptonomex](#). You can see Graphene as a toolkit for real-time blockchains. We separated the documentation into smaller parts for convenience and for the sake of easy location of relevant information.

If you are here to find out more about **BitShares** and want to get started right away, we recommend that you take a look at the [*Getting Started*](#) guide!

CHAPTER 1

Recent Updates

- 17/04/04 Stress Test results available in *Whitepapers*
- 16/06/29 *Distributed Access to the BitShares Decentralised Exchange*
- 16/04/07 *Getting Started*
- 16/03/17 *Traders, Supporting Libraries*
- 16/03/15 *Investor Guide, Claim your Investment*, bitshares/migration/legacy-blockchain
- 16/03/02 *Referral Program*
- 16/03/01 *How to Run and Use the Cli-Wallet, Transferring Funds using the cli-wallet, Voting, Voting*
- 16/02/13 Huge improvements in the *API Guide*
- 16/02/08 *Committee, How to Approve/Disapprove a Committee Proposal, Vesting Balances*
- 16/02/01 *Wallet Merchant Protocol*, Added search to the navigation
- 16/01/19 *Testnets, Creating a Prediction Market*, bitshares/user/eba
- 16/01/13 *Update/Change an existing UIA, Creating a UIA manually, Creating a new UIA, Network and Wallet Configuration*, integration/tutorials/index
- 16/01/12 *Assets/Tokens, Creating a UIA manually Creating a MPA manually, Assets FAQ, Privatized BitAssets, Publishing a Feed, Prediction Markets, Creating a Prediction Market, Closing/Settling a Prediction Market*

CHAPTER 2

Blockchain Specific Guides

The Graphene Technology has been applied to several blockchain already. BitShares 2.0 has been the first application of Graphene technology and you will be able to find almost everything feature implemented in BitShares 2.0. Further blockchains will be added independently.

BitShares 2.0 is a Financial Smart Contracts platform that enables trading of digital assets and has market-pegged assets that track the value of their underlying asset (e.g. bitUSD tracking the U.S. dollar).

2.1 BitShares 2.0

BitShares is a technology supported by next generation entrepreneurs, investors, and developers with a common interest in finding free market solutions by leveraging the power of globally decentralized consensus and decision making. Consensus technology has the power to do for economics what the internet did for information. It can harness the combined power of all humanity to coordinate the discovery and aggregation of real-time knowledge, previously unobtainable. This knowledge can be used to more effectively coordinate the allocation of resources toward their most productive and valuable use.

2.1.1 What is BitShares

We have compiled some articles that give a brief overview over how BitShares started, evolved and why it is where it is today. We recognize that BitShares has changed significantly over the last years and now feel confident to have a promising and very stable software that will improve steadily at a pace that partners and customers can follow easily.

If you are totally new to BitShares and want to get started right away, we recommend that you take a look at the [Getting Started](#) guide!

What is BitShares

BitShares is a technology supported by next generation entrepreneurs, investors, and developers with a common interest in finding free market solutions by leveraging the power of globally decentralized consensus and decision making. Consensus technology has the power to do for economics what the internet did for information. It can harness the

combined power of all humanity to coordinate the discovery and aggregation of real-time knowledge, previously unobtainable. This knowledge can be used to more effectively coordinate the allocation of resources toward their most productive and valuable use.

Bitcoin is the first fully autonomous system to utilize distributed consensus technology to create a more efficient and reliable global payment network. The core innovation of Bitcoin is the Blockchain, a cryptographically secured public ledger of all accounts on the Bitcoin network that facilitates the transfer of value from one individual directly to another. For the first time in history, financial transactions over the internet no longer require a middle man to act as a trustworthy, confidential fiduciary.

BitShares looks to extend the innovation of the blockchain to all industries that rely upon the internet to provide their services. Whether its banking, stock exchanges, lotteries, voting, music, auctions or many others, a digital public ledger allows for the creation of distributed autonomous companies (or DACs) that provide better quality services at a fraction of the cost incurred by their more traditional, centralized counterparts. The advent of DACs ushers in a new paradigm in organizational structure in which companies can run without any human management and under the control of an incorruptible set of business rules. These rules are encoded in publicly auditable open source software distributed across the computers of the companies' shareholders, who effortlessly secure the company from arbitrary control.

BitShares does for business what bitcoin did for money by utilizing distributed consensus technology to create companies that are inherently global, transparent, trustworthy, efficient and most importantly profitable.

BitShares has went through many changes and has done its best to stay on top of blockchain technology. Towards the end of 2014 some of the DACs were merged and the X was dropped from "BitShares X" to become simply BitShares (BTS).

Background

BitShares X was first introduced in a White Paper titled "A Peer-to-Peer Polymorphic Digital Asset Exchange" by Daniel Larimer, Charles Hoskinson, and Stan Larimer. Shortly after authoring the White Paper, the project was founded by Daniel Larimer of Invictus Innovations after receiving funding from Chinese venture capital firm BitFund.PE. Charles Hoskinson, founder of the Bitcoin Education Project, was a co-founder of the original project but has since left the team. The BitShares X project received a lot of attention in August 2013 when it was covered by CoinDesk and subsequently announced to the the BitcoinTalk forums on August 22nd 2013 as a project announcement. The project generated a good amount of buzz around the proposal, though the original scope and timelines have since modified.

Consensus Technology

Consensus is the mechanism by which organizations of people decide upon unitary rational action. While not considered technology in the traditional sense, consensus "technology" is the basis of democratic governance and the coordination of free market activity first coined by Adam Smith as the "Invisible Hand." The process of consensus decision-making allows for all participants to consent upon a resolution of action even if not the favored course of action for each individual participant. Bitcoin was the first system to integrate a fully decentralized consensus method with the modern technology of the internet and peer-to-peer networks in order to more efficiently facilitate the transfer of value through electronic communication. The proof-of-work structure that secures and maintains the Bitcoin network is one manner of organizing individuals who do not necessarily trust one another to act in the best interest of all participants of the network. The BitShares ecosystem employs Delegated Proof of Stake in order to find efficient solutions to distributed consensus decision making.

Distributed Autonomous Companies

Distributed Autonomous Companies (DAC) run without any human involvement under the control of an incorruptible set of business rules. (That's why they must be distributed and autonomous.) These rules are implemented as publicly auditable open source software distributed across the computers of their stakeholders. You become a stakeholder by buying "stock" in the company or being paid in that stock to provide services for the company. This stock may entitle you to a share of its "profits", participation in its growth, and/or a say in how it is run.

Community

The BitShares community is a global network of people who all share the same goal of creating and participating in various Distributed Autonomous Companies. The community mainly revolves around the BitShares Team and third parties who use Graphene (the toolkit that makes BitShares possible) to create their own Distributed Autonomous Companies. The main discussions in the BitShares community takes place openly at BitSharestalk.org.

The History of BitShares

The History of BitShares as laid out by Stan Larimer

"Here are the first parts of a planned series of blog articles that will appear on the bitshares.org website as soon as I am sufficiently pleased with them. This series is intended to be a resource for newbies and a summary of all that has been accomplished in the past year. Think of this as the Official Year One Newsletter."

- [Part One Source](#)
- [Part Two Source](#)
- [Part Three Source](#)

The History of BitShares Part One

Today is the one-year anniversary of the invention of BitShares. On the second day of June 2013, a rogue entrepreneur named Dan "Bytemaster" Larimer was struck on the head by a proverbial migrating coconut. When he regained consciousness, he realized that he had invented bit-USD, the key insight that makes BitShares possible. Before sundown he had reworked his original BitShares whitepaper and published the whole new concept at bitcointalk.org:

- [Creating a Fiat/Bitcoin Exchange without Fiat Deposits](#)

Over the next five weeks, Bytemaster engaged in a series of vigorous forum discussions defending and refining the concept. There he met Charles Hoskinson who helped to vet the idea and develop a business plan. Charles presented the plan to Li Xiaolai in China who agreed to fund the development. On American Independence Day, the Fourth of July 2013, Invictus Innovations was incorporated in the state of Virginia.

The next several months were spent bootstrapping the company and publishing articles, many of which may be found on the bitsharestalk.org thread at [Advice, Tutorials, and General References for Newbies](#).

In September, the concept of a Distributed Autonomous Company (DAC) was invented and introduced to the world in two groundbreaking LetsTalkBitcoin.com articles:

- [Overpaying for Security](#)
- [Bitcoin and the Three Laws of Robotics](#)

Invictus introduced the BitShares Vision to the world via presentations by Hoskinson and Larimer at the Atlanta Bitcoin Conference in October 2013. It is here that the plans for Keyhotee were first introduced – an integrated

multi-wallet, communication, and DAC interface application intended to defend privacy and help spread knowledge of BitShares technologies outside the crypto-currency community.

Hoskinson and Larimer parted company at this point. They each agreed to keep their reasons confidential and there is no bad blood from our point of view. The only official statement on the subject was made by CEO Bo Shen to end a minor forum firestorm here:

BitSharestalk

It is our opinion that Charles Hoskinson is the best dealmaker we have ever seen, and we miss his vision and talent for recruiting allies. No doubt he will help make his new Ethereum team very successful.

Despite this loss, all of this activity was beginning to create a buzz that would soon explode on the scene with a sequence of revolutionary innovations at roughly monthly intervals that continue to this very day.

The first was ProtoShares...

The History of BitShares Part Two

November's Innovation – BitShares PTS

BitShares PTS (formerly called ProtoShares) was developed for an entirely different reason than what its paradigm-shattering role became shortly after launch. In fact, every one of the subsequent breathtaking innovations came about from reacting to opportunities and lessons learned from the previous month's breakthroughs and, um, screw-ups. Necessity is the mother of invention. I wish we could say we had it all planned out in advance, but no business plan survives first contact with the market. We are merely blessed opportunists.

Initially, we were just looking for a way serve people interested in our first objective, BitShares X. A way for them to start mining and trading it early. BitShares X was first viewed as a coin backed by a built-in business that gives it more worth than the speculative value of a meme or some alternative technical implementation. In this first case, that integral unmanned business was a decentralized bank and exchange.

Yep. Your coins would now contain a bank, not the other way around.

Needless to say, this kind of second-generation crypto-company takes a lot longer to build and early adopters were growing impatient. So our plan was to just offer a plain old Bitcoin clone whose coins would be a BitShares X prototype - upgradable into BitShares X "bitshares" when it was ready to launch. We would simply initialize the first 10% of the bitshares to match all the public keys of PTS holders, giving them instant matching control of the same number of bitshares in BitShares X. This is how the concept of a protocoins was born.

We envisioned many exciting uses for protocoins. For example, they could be used as A way to separate investing in an idea from investing in one or more implementations of the idea. An incentive for competitors to cooperate on building an implementation because they could all be common stakeholders in the idea. A way to vet an idea and attract venture capitalists based upon prediction market evidence that the idea has value. A way for developers to invest in an idea and raise funding by generating growth from showing more and more evidence that they will successfully implement the idea in a way that benefits investors in the idea. A way for someone with a good idea but lacking the ability to implement it to share it and benefit from its ultimate implementation by somebody else. A way for an entire community to participate in "pre-mining" in a way that might be deemed fair (e.g. for unmanned businesses that must start out with enough currency to operate and enough credibility to get market depth on exchanges from Day One.) A more graceful "soft fork" way to upgrade to version two of a DAC by instantiating the new in parallel with the old and let the owners (shareholders) not just the employees (miners) decide when and if value transitions from the old to the new. A way to build a community and get them to cooperate on the implementation because they all have a stake in the idea. So you see, right off the bat we are talking about two assets: PTS and BTS. Before long, we would be talking about entire families of such assets. Second-generation crypto-currencies that we began to call crypto-equities because the coins also seemed like "shares" in the underlying unmanned business that gave the currency value - BitShares.

Since then, we have come to prefer the inverse of this dual metaphor:

Bitcoin is a type of crypto-company that implements a coin not BitShares as a type of crypto-coin that implements a company.

Of course, BitShares are something very different than shares in a government-created and therefore government-regulated organization. We are speaking metaphorically to help people understand how they work and what gives them value. They can still be viewed as ordinary altcoins (ok, incredibly powerful ordinary altcoins) as far as their underlying technology is concerned.

Charles Evans explored this dual metaphor in this delightful blog article:

A BitRose by Any Other Name. <http://bitshares.org/a-bitrose-by-any-other-name/>

We offered a bounty for an experienced coin designer to build the PTS protocol for us. A developer known as FreeTrade answered the call. It took him about a month to clone it from the Bitcoin library. Then, while we were still evaluating his code, another independent entrepreneur known as Super3 downloaded the open-source from FreeTrade's library and started it running. On November 5, 2013 Super3 went down in history as the miner of the first protocol block in crypto-equity history!

POW! The rest of the world (who had been eagerly awaiting the launch based on the several months we had been writing about it) jumped on it with everything they had. It took just a few days before the competition became so intense that people had a hard time mining solo with their individual computers. They started joining pools that several enterprising businessmen quickly set up and then everyone started renting cloud computers to remain competitive. By the end of the third week, there were hundreds of thousands of mining nodes competing. Several independent coin exchanges jumped in and listed PTS, driving it immediately into the top ten of the over 100 coins listed on coinmarketcap.com at the time.

So you see, we really don't own PTS. It was launched by the industry for the industry. We just described what ought to exist, and a decentralized industry of entrepreneurs produced it practically overnight.

Of course, that moon shot may have had something to do with one small suggestion we made literally at the last minute: we decided to recommend PTS be the basis for more than just BitShares X. PTS should also be used to initialize all of the other second-generation assets we had been writing about. Mine once for a whole family of assets. Why should you have to keep mining over and over again to get a "fair" distribution?

In fact, we recommended that other developers do the same thing. Suddenly BitShares PTS was backed by more than thin air. More than just one unmanned business. More than just one company's product line of unmanned businesses. It could well become backed by a good portion of the unmanned business industry!

BitShares PTS was valuable because as a universal prototype it was upgradable to multiple future releases like BitShares X.

Just like a good deal on Microsoft Office 1.0 might get you free upgrades on Word, Excel, PowerPoint and all the rest ... for as long as you both shall live!

To a community willing to speculate on any altcoin with a cute name, that was all it took. Now there was something tangible to speculate on. Soon crypto-currency speculators would be demanding to know every new asset's business case.

Imagine that! We had almost accidentally changed the crypto-currency industry forever.

It was just our opening shot.

The History of BitShares Part Three

December's Innovation – TAPOS and the End of Mining

In the weeks that followed it became increasingly obvious that the whole paradigm of mining on which the crypto-currency industry is founded was horribly flawed. While generally billed as a "fair" lottery for wide distribution of a new currency, it was clear that the ordinary guy was still at a disadvantage. Technically savvy people could use and

optimize the tools - others could not install their wallet. Wealthy individuals could rent computers by the thousands - others had no computer at all. Only a very small percentage of the general population was benefitting - sucking up the lion's share of the coins and then reselling them on the market at a profit.

Now, there's nothing wrong with using your brains or wealth to earn a profit while contributing to society (like, say, developing a new technology), but as far as the general public was concerned, this small elite group of individuals were effectively just selling the currency into existence. Most of the general population had to buy them from the market anyway!

And even those elite few only got to keep a small percentage of what the market was willing to pay for the currency. They were required to destroy most of what they received from the market doing the electronic equivalent of digging holes and filling them back in. The whole industry was ein bisschen poco loco.

"No, wait!", the Bitcoin-trained community protested, "burning the seed capital is the price we must pay for securing the network!"

Except the network was not really being secured. Economies of scale dictate that hashing power will always migrate toward specialized capital-intensive organizations ultimately killing the very decentralization that mining was supposed to ensure. Today, most Bitcoin mining power is concentrated in the hands of a half-dozen individuals with just two of them controlling over 51%. And they proudly collaborate "for the good of the network."

Bytemaster recognized that Bitcoin could be viewed as an unprofitable company and its coins as stock in that company. Stock value was generally rising because demand for its services (efficient private money transmission) exceeded supply. But, meanwhile it was bleeding red ink. 100% of its transaction fees were going to pay its employees (the miners). But that still wasn't enough. It had to print more money (up to 12% annual inflation) also to pay its employees. So Bitcoin is a company with annual losses near 12%. (And the employees were only getting to keep a few percent of the money being wasted on them.)

He decided that eliminating those employees was a key objective that would inevitably lead to a whole new generation of profitable crypto-businesses. Assets based on destructive mining would go the way of the dinosaur, unable to compete with profitable business models of second generation assets that could afford to pay dividends and interest to their holders. It was just a matter of time.

So a month after the ProtoShares revolution, around December 1, Bytemaster fired his second shot heard round the world: all his future designs would replace Proof of Work mining with a Proof of Stake derivative.

Transactions as Proof of Stake ([TAPoS](#)) and the End of Mining . An algorithm that was lightweight enough to run invisibly on anyone's computer, for free! Mining was dead. Next generation crypto-assets would be profitable. They would be valuable because they returned a yield, rather than for superficial speculative reasons.

There were merely a few technical wrinkles to iron out...

History of Funding

Also see, Summary of Key Facts for Invictus Stakeholders

When Invictus of VA was formed under Charles Hoskinson's term as CEO, our purpose was to create a company that would achieve all the objectives of Mr. Li as our primary investor.

(Since shortly after our founding, Mr. Li Xiaolai has held a subscription agreement that entitles him to buy 25% of our shares for a fixed price payable in increments spread out over the first year. Mr. Li also acquired an additional 1% from Charles Hoskinson in a separate purchase. This means that his total stake in Invictus is 26% of which he has completed payments on 21% as scheduled. His final payment for the last 5% is on hold pending completion of a restructuring forced by discovery of certain applicable U.S. regulations. All these shares will be equally treated.)

We had three nested tasks:

Build and launch BitShares X Build a company to Build and launch BitShares X. Build a decentralized industry in which this company could build and launch BitShares X (and many more).

Part of our task was to research the legal requirements to accomplish all of these goals.

In the process of studying the requirements in the United States we ran into a number of issues and uncertainties. In particular, there are strict rules about who can own shares of a U.S. corporation.

We recommended to Mr. Li that he ask an attorney he trusts to start over and create a company that would be able to meet all of the goals and honor all of his commitments. It has taken six months to work out all the details, after consulting with Li's attorney and multiple U.S law firms.

We will soon be ready to release a public statement about the details, but the bottom line is that Invictus Innovations Incorporated, LTD in Hong Kong is the company we intended to create in Virginia, except with the ability to meet the needs of Asian investors better than we can here.

So, you can think of it as relocating the Virginia company, but legally they are two independent companies with independent management aiming to meet Mr. Li's goals and obligations 100%.

The Virginia company now only handles small tasks associated with American payroll and payment processing. Further details on this decomposition into independent businesses optimized to comply with all regulations in their domains will be forthcoming.

The Great Consolidation

In the late part of 2014 it became obvious that Bytemaster had to lend his energies to other projects. People had donated AGS funds with the expectation of future DACs. With the decreasing funding due to dropping BTC prices and the requirements of Dan Larimer, the Great Consolidation occurred. Follow My Vote and DNS were merged into BTS so that all developers could be brought to work directly on one product instead of DACs all competing for users.

One outcome of this was also the addition of paying on the blockchain. Previously BitShares was a purely deflationary blockchain with dividends paid out by the burning of transaction fees. (Less currency in existence gives more value to those remaining.) With a pressing need to be the most innovative crypto-currency out there, it was determined that the Delegates needed to start paying. So the cap on Bitshares was raised to be slowly paid out similar to the inflation in Bitcoin. The rate was made to be kept under the current level of Bitcoin inflation, but delivering direct and meaningful value. Timeline of BitShares by forum announcements

- Momentum Proof of Work Introduced on BTT - October 18 2013
 - <https://bitcointalk.org/index.php?topic=313479.0>
 - <http://static.squarespace.com/static/51fb043ee4b0608e46483caf/t/52654716e4b01acd1ac8a085/1382369046208/MomentumProofOfWork.pdf> (White Paper)
 - <https://bitsharestalk.org/index.php?topic=962.msg9752#msg9752>
- Keyhotee ID Preorder - November 3, 2013
 - <https://bitsharestalk.org/index.php?topic=2.msg2#msg2>
- Mining of Bitshares PTS (Protoshares) - November 5, 2013
 - <https://bitsharestalk.org/index.php?topic=4.msg4#msg4>
- Transactions as Proof of Stake - November 30, 2013
 - <https://bitsharestalk.org/index.php?topic=1138.msg12010#msg12010>
 - <http://the-iland.net/static/downloads/TransactionsAsProofOfStake.pdf>
 - <https://bitsharestalk.org/index.php?topic=1138.msg11968#msg11968>
 - <https://bitsharestalk.org/index.php?topic=1138.msg12967#msg12967>
- Consensus + TaPoS

- <https://bitsharestalk.org/index.php?topic=1138.msg29905#msg29905>
- <https://bitsharestalk.org/index.php?topic=3588.msg45119#msg45119>
- The Inception of DPOS - December 8, 2013
 - <https://bitsharestalk.org/index.php?topic=1138.msg13602#msg13602>
 - <https://bitsharestalk.org/index.php?topic=1138.msg14784#msg14784>
- The Inception of AGS - December 14, 2013
 - <https://bitsharestalk.org/index.php?topic=1397.msg14794#msg14794>
- Official AGS Announcement - December 25, 2013
 - <https://bitsharestalk.org/index.php?topic=2644.msg32817#msg32817>
- February 28 Snapshot Announced - January 26, 2014
 - <https://bitsharestalk.org/index.php?topic=2591.45>
- Bitshares X Whitepaper - February 14th, 2014
 - https://docs.google.com/document/d/1RLcjSXWuU9vBJzzqLEXVACSCdn8zXKTTJRN_LfoCjNY/edit?pli=1#
- TaPos with a Trustee - March 28, 2014
 - <https://bitsharestalk.org/index.php?topic=3865.msg48605#msg48605>
- BitShares X released by DACsunlimited, July 19th, 2014
 - <https://bitsharestalk.org/index.php?topic=5750.0>

In addition there are numerous threads discussing The Great Consolidation.

Delegated Proof of Stake

Delegated Proof of Stake (DPOS) is a new method of securing a crypto-currency's network. DPOS attempts to solve the problems of both Bitcoin's traditional Proof of Work system, and the Proof of Stake system of Peercoin and NXT. DPOS implements a layer of technological democracy to offset the negative effects of centralization.

Background

Delegated proof of stake mitigates the potential negative impacts of centralization through the use of witnesses (formally called *delegates*). A total of N witnesses sign the blocks and are voted on by those using the network with every transaction that gets made. By using a decentralized voting process, DPOS is by design more democratic than comparable systems. Rather than eliminating the need for trust all together, DPOS has safeguards in place to ensure that those trusted with signing blocks on behalf of the network are doing so correctly and without bias. Additionally, each block signed must have a verification that the block before it was signed by a trusted node. DPOS eliminates the need to wait until a certain number of untrusted nodes have verified a transaction before it can be confirmed.

This reduced need for confirmation produces an increase in speed of transaction times. By intentionally placing trust with the most trustworthy of potential block signers, as decided by the network, no artificial encumbrance need be imposed to slow down the block signing process. DPOS allows for many more transactions to be included in a block than either proof of work or proof of stake systems. DPOS technology allows cryptocurrency technology to transact at a level where it can compete with the centralized clearinghouses like Visa and Mastercard. Such clearinghouses administer the most popular forms of electronic payment systems in the world.

In a delegated proof of stake system centralization still occurs, but it is controlled. Unlike other methods of securing cryptocurrency networks, every client in a DPOS system has the ability to decide who is trusted rather than trust

concentrating in the hands of those with the most resources. DPOS allows the network to reap some of the major advantages of centralization, while still maintaining some calculated measure of decentralization. This system is enforced by a fair election process where anyone could potentially become a delegated representative of the majority of users.

Rationale Behind DPOS

- Give shareholders a way to delegate their vote to a key (one that doesn't control coins 'so they can mine')
- Maximize the dividends shareholders earn
- Minimize the amount paid to secure the network
- Maximize the performance of the network
- Minimize the cost of running the network (bandwidth, CPU, etc)

Shareholders are in Control

The fundamental feature of DPOS is that shareholders remain in control. If they remain in control then it is decentralized. As flawed as voting can be, when it comes to shared ownership of a company it is the only viable way. Fortunately if you do not like who is running the company you can sell and this market feedback causes shareholders to vote more rationally than citizens.

Every shareholder gets to vote for someone to sign blocks in their stead (a representative if you will). Anyone who can gain 1% or more of the votes can join the board. The representatives become a "board of directors" which take turns in a round-robin manner, signing blocks. If one of the directors misses their turn, clients will automatically switch their vote away from them. Eventually these directors will be voted off the board and someone else will join. Board members are paid a small token to make it worth their time ensuring uptime and an incentive to campaign. They also post a small bond equal to 100x the average pay they receive for producing a single block. To make a profit a director must have greater than 99% uptime.

Pooled Mining as Delegated Proof of Work

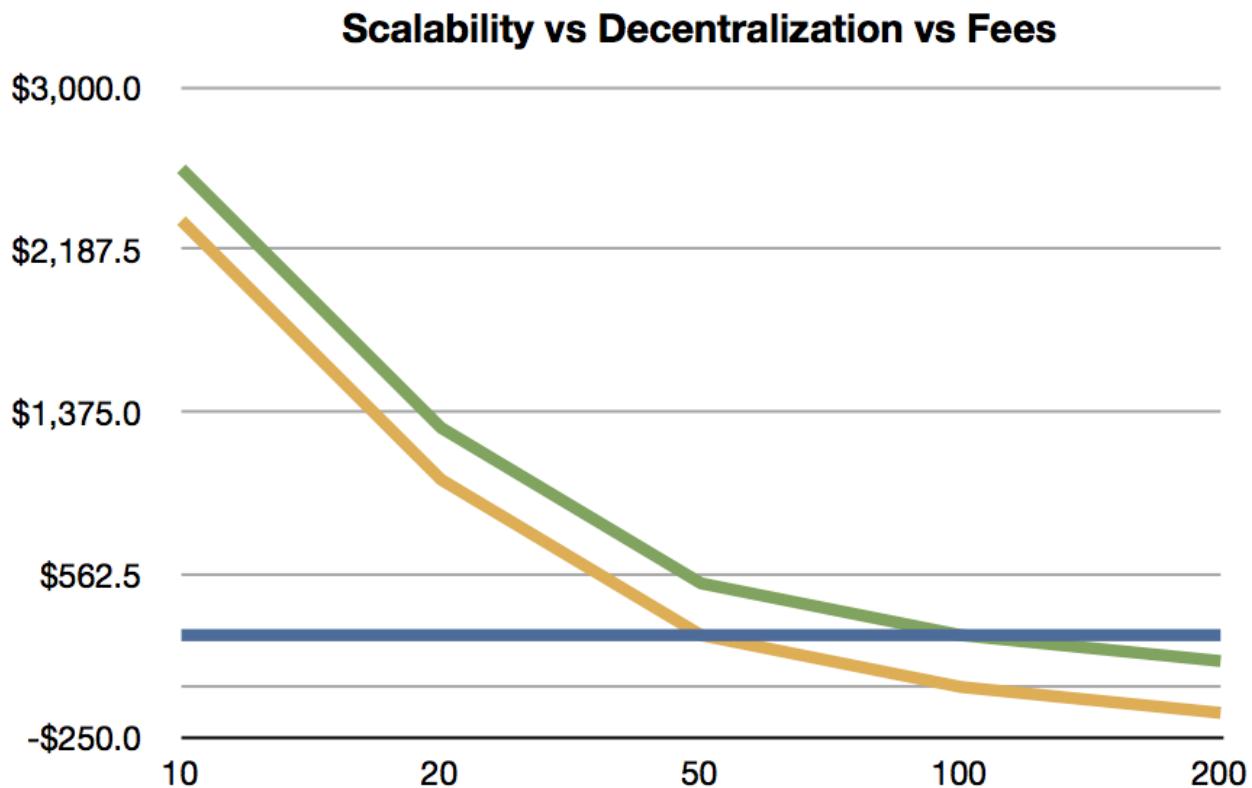
So how is this different than Bitcoin? With Bitcoin, users must pick a mining pool and each pool generally has 10% or more of the hash power. The operator of these pools is like a representative of the clients pointed at the pool. Bitcoin expects the users to switch pools to keep power from becoming too centralized, but collectively five major pools control the network and manual user intervention is expected if one of the pools is compromised. If a pool goes down then the block production rate slows proportionally until it comes back up. Which pool one mines with becomes a matter of politics.

Reasons to not randomly select representatives from all users

- High probability they are not online.
- Attackers would gain control proportional to their stake, without any peer review.
- Without any mining at all, the generation of a random number in a decentralized manner is impossible and thus an attacker could control the random number generation.

Scalability

Assuming a fixed validation cost per transaction and a fixed fee per transaction, there is a limit to the amount of decentralization that can take place. Assuming the validation cost exactly equals the fee, a network is completely centralized and can only afford one validator. Assuming the fee is 100x the cost of validation, the network can support 100 validators.



Systems like Nxt and Peercoin will have excessive fees if they intend to allow everyone to be a validator and earn fees at scale. What this means for Nxt and Peercoin is that anyone with less than 1% stake cannot validate profitably unless their fees are higher than our DPOS chain. If these chains assume 100 delegates is too centralized and start promoting they have 1000 validators, then their fees must be 10x those of DPOS. If such a chain grew to be the size of Bitcoin (\$10 B) then only those with \$1M worth of coin could validate profitably and most would consider that an elite club. If they reduce the minimum stake to be a validator to \$1000, then their fees would be 10,000 times higher than DPOS.

Developers of DPOS assume that everyone with less than the amount required to validate won't participate. Also assumed is a "reasonable" distribution of wealth. It's clear that unless alternate chains have unusually high fees, there will only be a handful of people with enough stake to validate profitably.

In conclusion, the only way for POS to work efficiently is to delegate. In the case of Nxt, they can pool their stake by some means and ultimately this will end up like DPOS prior to approval voting with a variable number of delegates. Delegates wouldn't actually receive any income as with mining pools because the validation expenses will consume the vast majority of the transaction fees.

The end result is that decentralization has a cost proportional to the number of validators and that costs do not disappear. At scale, these costs will centralize any system that does not support delegation. This kind of centralization should be designed as part of the system from the beginning so that it can be properly managed and controlled by the users, instead of evolving in some ad hoc manner as an unintended consequence.

Role of Delegates

- A witness is an authority that is allowed to produce and broadcast blocks.
- Producing a block consists of collecting transactions of the P2P network and signing it with the witness' signing private key.
- A witness' spot in the round is assigned randomly at the end of the previous block

How to become a delegate

Howto Become an Active Witness

Voting Algorithm

How do I get “votes?”

- Persuade others to give upvotes to your witness
- When another user gives an upvote to your (and possibly other) delegates
- A user can give an upvote for more than one witness. As a result all upvoted witness get a vote
- Convince proxies (that vote on behalf of their followers) to vote for you

Why use only upvotes?

- Giving only upvotes, and allowing multiple votes per share, is called **Approval Voting**, and comes with several advantages over the old *delegation* voting.
- No downvotes are needed, which not only simplifies usability but also reduces code and complexity.

How are ‘votes’ counted?

Once every *maintenance interval*, all votes are recounted and the corresponding result takes effect.

Is there an anti-vote?

Not any more. After discovering [emski's attack](#) the developers decided to use **Approval Voting**.

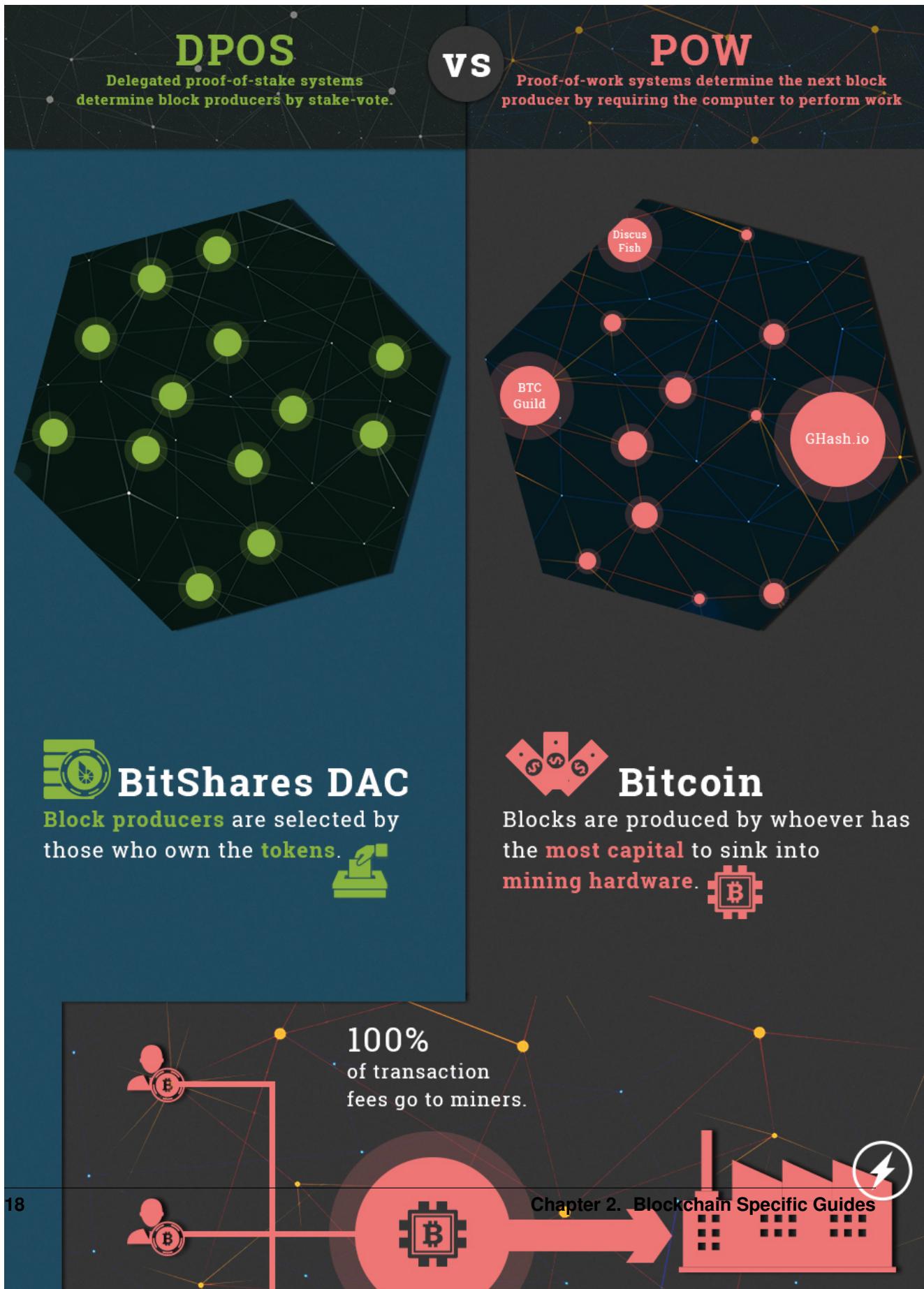
Disincentives for Attacks

- By choosing not to produce a block, a witness risks getting fired and they lose guaranteed profits in the future.
- A dishonest delegate would only fail to produce a block if they were sure to win something from it
- If a lottery only payed out 50% to a jackpot (giving the other 50% to charity) then the most this dishonest delegate could do is break even.
- Witnesses can't sign invalid blocks as the block needs confirmation by the other witnesses as well

How many witnesses are securing the network

This is totally in the hands of the shareholders. If the majority votes for 50 witnesses, then 50 witnesses will be used. If the shareholders only vote for 20, so be it. The minimum possible witness count is 11.

DPOS Infographic



Sources and Discussions

- <https://bitsharestalk.org/index.php?topic=5164.msg67657#msg67657>
- <https://bitsharestalk.org/index.php?topic=5205.0>
- https://github.com/BitShares/bitshares_toolkit/wiki/Delegated-Proof-of-Stake
- <https://bitsharestalk.org/index.php?topic=4984.0>
- <https://bitsharestalk.org/index.php?topic=4927.0>
- <https://bitsharestalk.org/index.php?topic=4869.0>
- <https://bitsharestalk.org/index.php?topic=4853.0>
- <https://bitsharestalk.org/index.php?topic=4836.0>
- <https://bitsharestalk.org/index.php?topic=4714.0>

Whitepapers

Under Construction

During ongoing quality assurance and documentation improvements, we have realized unfavorable use of terminology in the published whitepapers.

As a consequence and to reduce misunderstandings, we have taken down the old whitepapers in order to properly review them under legal guidance.

2.1.2 Guides

Since BitShares is a software with a variety of features and possibilities, not only for developers but also for end-users and merchants, we have collected several guides for each group that should help think things through.

Installation

This section describes the installation procedure and the build process for those that want to compile from the source.

Downloads

Precompile Executables

Compiled executables are available for [download from github](#).

Sources

The sources are located at [github](#) and can be downloaded with *git*.

```
git clone https://github.com/bitshares/bitshares-core
```

Since the repository makes use of so called *submodules* which are repositories on their own, we need to refresh those.

```
git submodule update --init --recursive
```

Building from Sources

Downloading the sources

The sources can be downloaded from github as described [here](#).

Dependencies

Development Toolkit

The following dependencies were necessary for a clean install of Ubuntu 14.04 LTS:

```
sudo apt-get install gcc-4.9 g++-4.9 cmake make \
    libbz2-dev libdb++-dev libdb-dev \
    libssl-dev openssl libreadline-dev \
    autoconf libtool git libcurl4-openssl-dev
```

Boost (optional)

BitShares-core now supports Boost up to version 1.63.0. If you have a newer version installed, this is how you can install boost 1.63.0 individually:

```
export BOOST_ROOT=$HOME/opt/boost_1_63_0
sudo apt-get update
sudo apt-get install autotools-dev build-essential \
    g++ libbz2-dev libicu-dev python-dev
wget -c 'http://sourceforge.net/projects/boost/files/boost/1.63.0/boost_1_63_0.tar.
bz2/download' \
    -O boost_1_63_0.tar.bz2
sha256sum boost_1_63_0.tar.bz2
# "beae2529f759f6b3bf3f4969a19c2e9d6f0c503edcb2de4a61d1428519fcb3b0"
tar xjf boost_1_63_0.tar.bz2
cd boost_1_63_0/
./bootstrap.sh "--prefix=$BOOST_ROOT"
./b2 install
```

Building BitShares

After downloading the BitShares sources according to [the download page](#), we can run `cmake` for configuration and compile with `make`:

```
cmake -DBOOST_ROOT="$BOOST_ROOT" -DCMAKE_BUILD_TYPE=Release .
make
```

Note that the environmental variable `$BOOST_ROOT` points to your local install directory of boost if you have installed it manually.

Updating BitShares

A quick tutorial on updating your BitShares binaries can be found [here](#).

Distribution Specific Settings

Ubuntu 14.04

As g++-4.9 isn't available in 14.04 LTS, you need to do this first:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
```

If you get build failures due to abi incompatibilities, just use gcc 4.9

```
CC=gcc-4.9 CXX=g++-4.9 cmake .
```

Ubuntu 15.04

Ubuntu 15.04 uses gcc 5, which has the c++11 ABI as default, but the boost libraries were compiled with the cxx11 ABI (this is an issue in many distros). If you get build failures due to abi incompatibilities, just use gcc 4.9:

```
CC=gcc-4.9 CXX=g++-4.9 cmake .
```

Upgrading

Recompiling from Sources

For upgrading from source you only need to execute:

```
git fetch
git checkout <version>
git submodule update --init --recursive
cmake .
make
```

Migrating from BitShares 1.0 to BitShares 2.0

This migration tutorial is relevant only to those customers and investors that have participated in BitShares 1.0. We show improvements, new features and give assistance for claiming your funds in BitShares 2.0.

What is New in BitShares 2.0

- **Votable Network Parameters:** BitShares 2.0 will allow its shareholders to fine-tune any parameter available to the protocol. This includes, block size, block interval, but also the payment for block producers and transaction fees.

- **Flexible and Dynamic Access Control:** BitShares 2.0 allows customers and participants a flexible and dynamic access to its funds or account handle. A so called *Authority* can consist of a flat hierarchy similar to *multi-signature* in Bitcoin, but could also support tree hierarchies never to be seen before. Read more about this about [dynamic account permissions](#).
- **Transferable Account Names:** Since Control over Funds is separated from the control over an account, we can have transferable account names that are registered on the blockchain. Named accounts allows for much easier transfers because no cryptic strings needs to be handed out. Read more about [transferable named accounts](#).
- **On-Chain Proposed Transactions:** In traditional crypto currencies, a multi-signature transaction has to be transferred to its corresponding signers on separated communication channels (off-chain). BitShares 2.0 allows to propose transactions on the chain and have the signers be notified for their required signature automatically. No more manual communications are required.
- **New Full-Node/Client Concept:** We recognize the hassles some people had when synchronizing the BitShares 1.0 blockchain with the heavy-weighted BitShares full client. In order to offer more comfort and a faster trading experience, we decided to separate the user-interface from the block syncing core component that connects to the peer-to-peer network. Of course, both are open source and a full node can run easily, we understand that some users mainly prefer to use the frontend not bothering about the blockchain.
- **Referral Program:** PayPal and Dwolla showed the success of referral programs, a program that could easily and cheaper be implemented in a decentralized software protocol. Hence we took our chance and implemented a blockchain based referral program. From every transaction fee, paid by a customer you referred, you will get a fraction. Of course, this fraction can be tuned by shareholders! Read more about the [referral program](#).
- **Recurring & Scheduled Payments:** We wanted to offer a way to have our rent payed automatically. So we implemented it in the blockchain. In BitShares 2.0, participants are capable of allowing others to withdraw funds from your account. Of course, you can define a daily/weekly or monthly limit. Read more about [recurring and scheduled payments](#).
- **Additional Privatized BitAssets:** In contrast to Market Pegged Assets (also known as BitAssets) that have a price feed published by witnesses that have approval of shareholders, a *privatized* bitasset allows to create market pegged assets that have an individual set of price feed publishers that do not need shareholders' approval. Hence, everyone can create a privatized bitAsset to track an individual value, such as indices, or binary predictions.

What has Changed since BitShares 0.9?

- **BitAssets are Collateralized Loans from a Counterpartyrisk-Free Smart Contract:** Our research has identified an improved mechanism to achieve a solid *peg* of bitAssets to its underlay. BitAssets like the bitUSD in BitShares 2.0 will always trade for *at least* the value of its underlying asset, i.e. \$1. We have summarized the economical analysis and incentives for market participants here: [bitAssets 2.0](#)
- **Faster Blocks:** Initially, the BitShares 2.0 blockchain will come with 3 seconds block interval with the option to reduce down to 1 second if shareholders agree.
- **Industrial Performance:** BitShares 2.0 can support massive load and works well beyond 100k transactions per second. Find out how we achieve [industrial performance and scalability](#).
- **New Reactive UI:** The BitShares 1.0 user interface was powerful but lacking in responsiveness and performance. For Bitshares 2.0 we've reimplemented the whole wallet using the React.js framework developed by Facebook, which is well-known for having excellent performance. The new BitShares UI is an entirely browser-based wallet, with private keys maintained in the browser. We expect a flourishing ecosystem of forked and tweaked wallets based off of our UI.
- **Accounts must be registered:** In BitShares 2.0 we have separated *authorities* from transaction partners. Hence, if Alice wants to send funds to Bob, it may be required that only Celine signs for that transaction. Also, BitShares 2.0 has a [referral program](#). Both features combined make it necessary that participants *register* an account on the blockchain.

- **No more Hierarchies in Account Names:** In BitShares 1, there have been hierarchies in account names. Namely, you could only create a sub-account `home.wallet` if you also owned `wallet`. In BitShares 2.0, these hierarchies no longer exist and to register `home.wallet` you don't need to own `wallet`.
- **Explicit Privacy:** The *TITAN* technology in BitShares 2.0 slowed down blockchain processing significantly. Because of this and because TITAN did not really offer good privacy, we eliminated TITAN as a default transaction feature. Hence: **Account transactions are public now as well.** However, since we recognize the value of financial privacy, we offer *blinded* transactions that hide the transferred *amount*, and *stealth* transactions that hide the sender and receiver. A combination of both is also possible.
- **Prices are Fractions:** To circumvent rounding errors, all prices in BitShares 2.0 are represented as fractions.
- **Delegates are now Witnesses and Payed Positions are now Budget Items:** Since we have separated the business part from the block producing part, we now call block producers (formerly known as *delegates*) witnesses, while the additional payed position for workers are called budget items.

Blockchain Upgrade

BitShares 2.0 will be initialized with what is called a *Genesis Block*. That genesis block will be constructed from the balances of BitShares 1.0. BitShares 1.0 offers many features that need to be migrated into BitShares 2.0. To simplify the process and reduce the risk of errors, the following conditions will be met:

- **Funds:**
 - **BTS Tokens:** All BTS balances will be migrated 1:1. The supply not change!
 - **User-Issued-Assets:** All UAI tokens will be migrated 1:1
 - **BitAssets:** Because the new chain is a simple migration and should retain all the same “perceived value”, all BitAssets and short positions are migrated 1:1.
- **Account Names:**

Under BitShares 2.0, accounts are transferable and have different prices based upon the “quality” of the account name. Any “premium” names registered on or after 2015-06-08 (US Eastern time) will be given the prefix “`bts-`” or similar after the migration. All account names registered on or after 2015-06-18 (US Eastern time) will be prefixed with “`bts-`” unless they were registered using the BitShares Faucet.

 - **Premium Name:** No numbers and has vowels
 - **Cheap Name:** Has numbers or no vowels

All other account names will be migrated with their corresponding owner/active keys.
- **Open Orders:** Open orders (except open short positions) will **not** migrate and the funds will be credited to the corresponding owners.
- **Open Shorts:** Short orders will be migrated to BitShares 2.0 on a 1:1 ratio. You collateral will be imported as a separated account (e.g. `usd-collateral-holder-124`) under your control.
- **Transaction History:** Transaction histories of BitShares 1.0 will be inaccessible in BitShares 2.0.
- **Vesting Balances:** Vesting balances will migrate under the existing terms, if two or more vesting balances were partially claimed as part of the same transaction prior to the snapshot the vesting balances may be merged into a single balance.
- **Unclaimed Delegate Pay:** Delegates that did not claim their pay prior to the snapshot will be able to claim their pay by importing their corresponding keys similar to any other balance.
- **Assets:** User issued assets and market pegged assets will migrated with their corresponding issuer and holders.
- **Deprecated Features:** Some features have turned out to be unreliable or impractical and will thus deprecate:

- **Wall Messages** will not be migrated as the feature is now deprecated
- Asset **description information** is no longer part of the blockchain state and will not be migrated
- Account **public data** is deprecated and is no longer part of the blockchain state
- BitShares URL scheme: *bts://* will be deprecated due to migration to hosted web wallets

Exporting Your Wallet

In this tutorial, we will export your wallet including all keys to access your accounts and funds, into a single JSON-formatted file. Note that your private keys will be encrypted and you will be required to provide the corresponding pass phrase when importing your funds into BitShares 2.0.

BitShares 1.0 Full Client

Since the snapshot has taken place already, all you need to do now to get access to your funds in BitShares 2.0 is described in the following.

Firstly, you need to upgrade your BitShares client to version 0.9.3c. To do the upgrade you need to:

- download the installation file from the [bitshares webpage](#)
- uninstall your previous version of the BitShares client
- install the new version

Synchronize your Wallet

Note: If you see all your funds in your wallet, you can safely skip this paragraph.

We now need to sync with the blockchain. This is only necessary if you think that since the last time you did the syncing there have been some new transactions involving any of your accounts.

legacy-blockchain

You can see the syncing progress from the status bar or from the `info` command in the console (account list->advanced settings->console). After having *synced* the blockchain, your wallet will automatically attempt to rescan the blockchain for new transactions. Depending on the amount of accounts in your wallet, this step should only take very few minutes.

Since BitShares 0.9.3c, we have a Graphene compatible Export Keys function that can be accessed in two ways:

- by accessing it in the main menu
- by issuing a command in the console.

Export via the main menu

Just select File Menu -> Export Wallet and you'll be asked to select a file location where the keys will be exported.

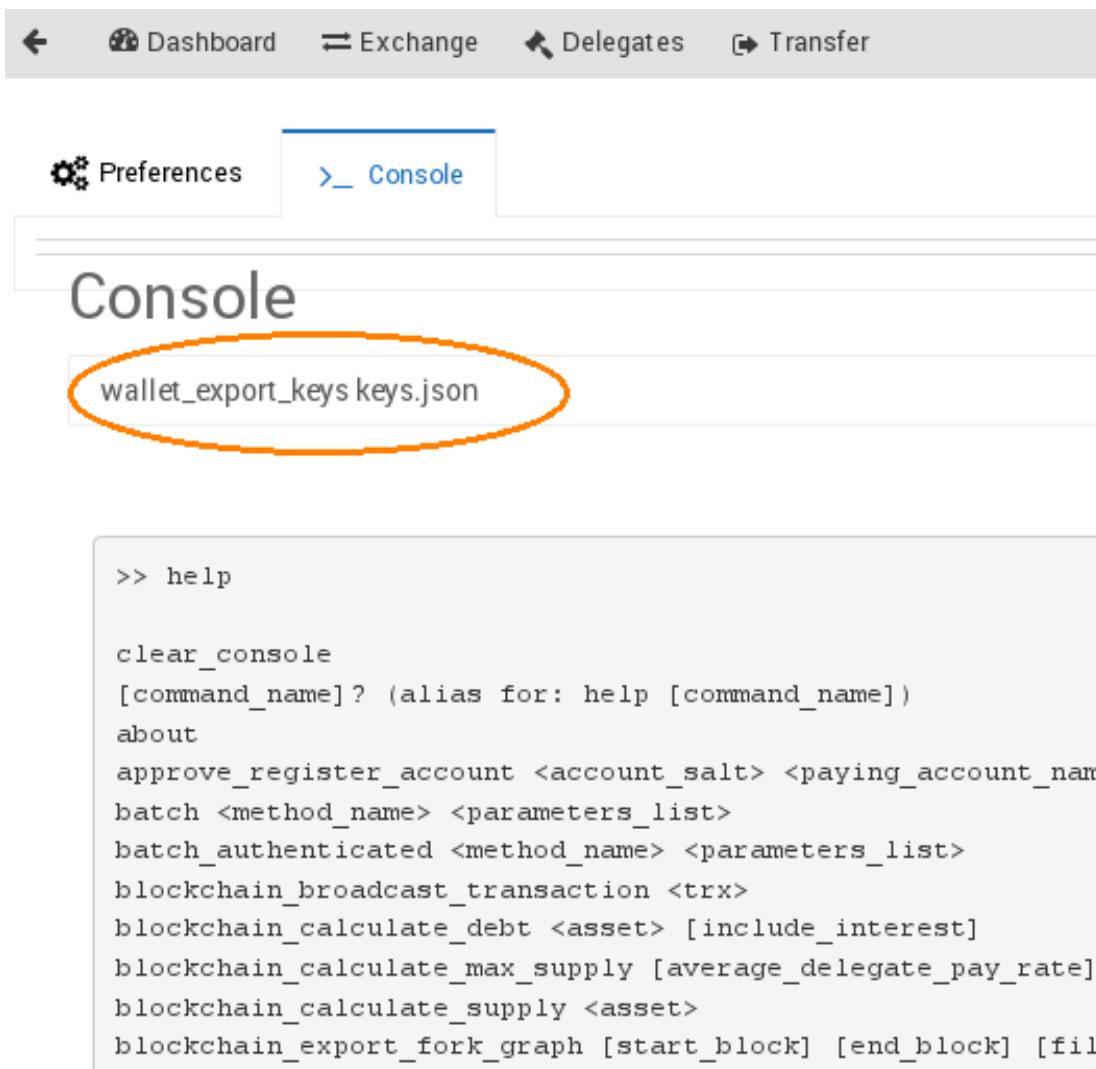
Note: Due to a known bug, if you are on Windows the only option that will work for you is the console command - the file exported using the menu will not be compatible with BTS 2.0. This refers to Windows only.

Export via the console

- navigate to the console: Account List -> Advanced Settings -> Console
- type: `wallet_export_keys [full path to the file]/[file name].json` e.g. on Windows: `wallet_export_keys C:\Users\[your user name]\Desktop\keys.json` e.g. on Mac: `wallet_export_keys /Users/[your user name]/Desktop/keys.json` e.g. on Linux: `wallet_export_keys /home/[your user name]/Desktop/keys.json`
- Please replace [your user name] with your Windows account name.
- and hit Enter

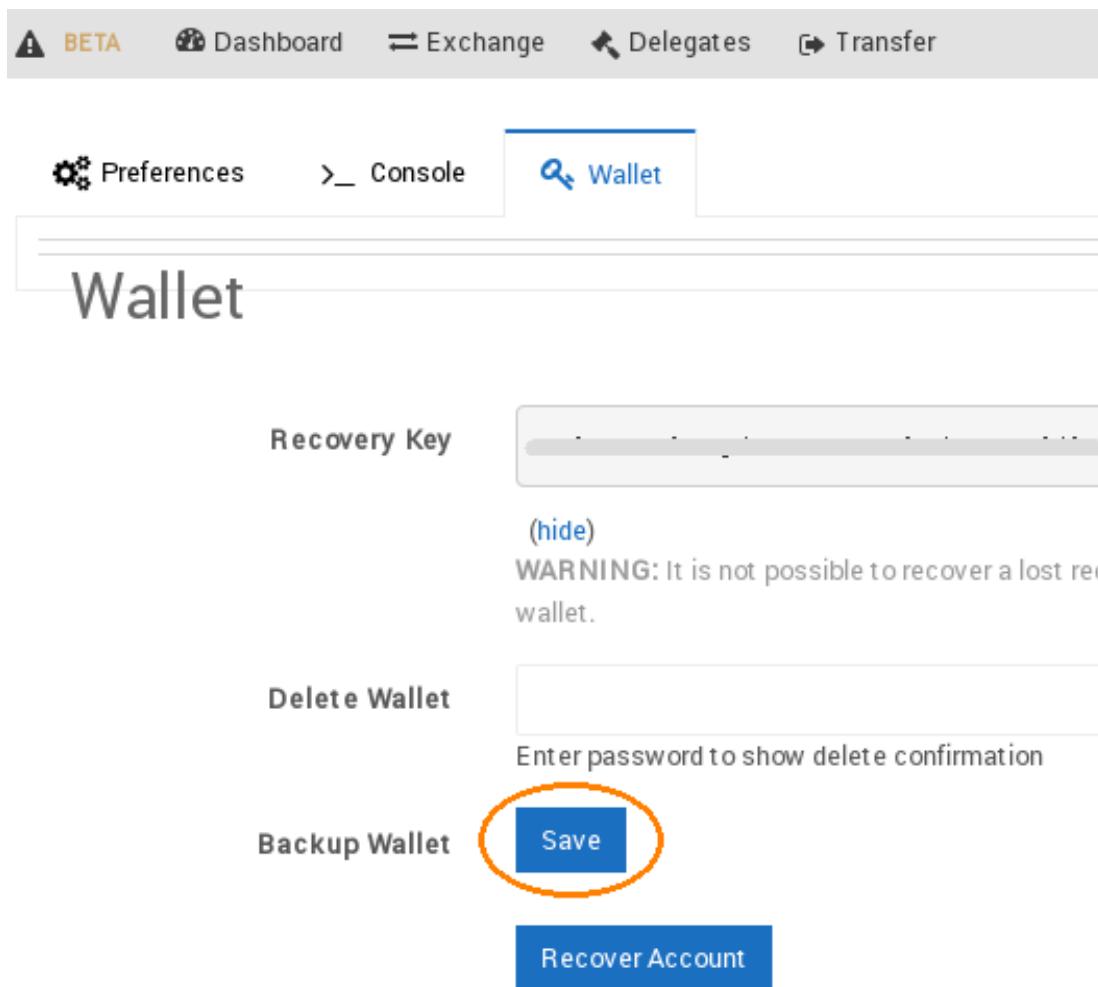
Note: The exported wallet file will be encrypted with your pass phrase! Make sure to remember it when trying to use that file again!

Note: If you are on Windows and your file path tries to access the C drive directly (e.g. C:keys.json) you might need to run the BitShares client as an administrator. So the least complicated option will be to aim for the desktop as in the example above.



wallet.bitshares.org

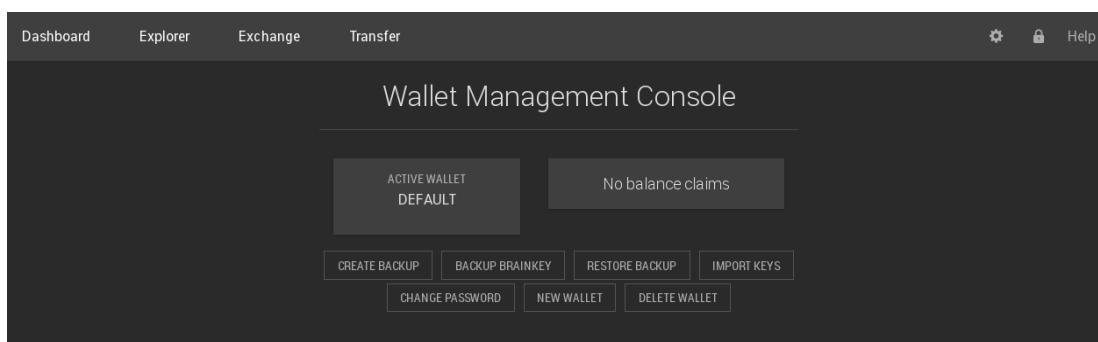
The keys of the [web wallet](#) can be exported simply by downloading a backup wallet. It can be obtained from the web wallet's preferences: (*Account List->Advanced Settings->Wallet*).



Importing Your Wallet

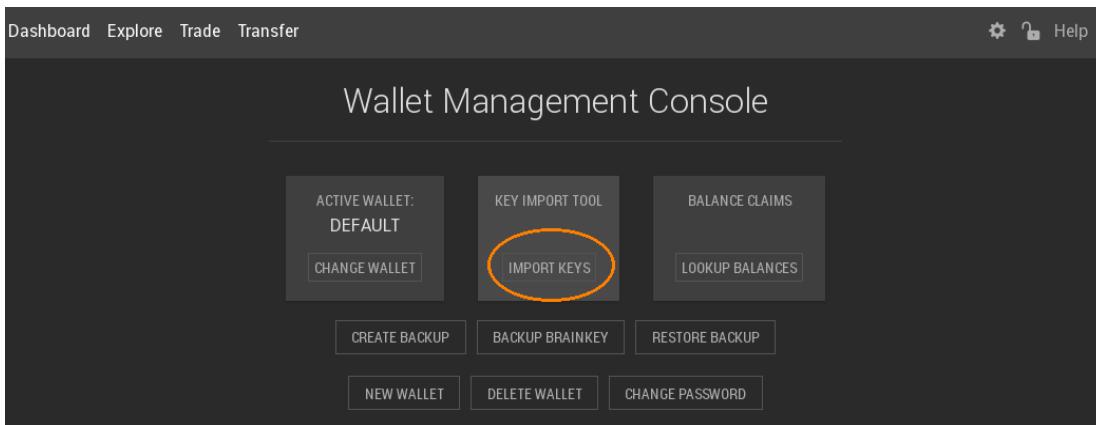
Web Wallet

The web wallet of BitShares 2.0 has a **Wallet management Console**, that will help you import your funds. It can be accessed via *BitShares 2.0: Settings -> Wallets*

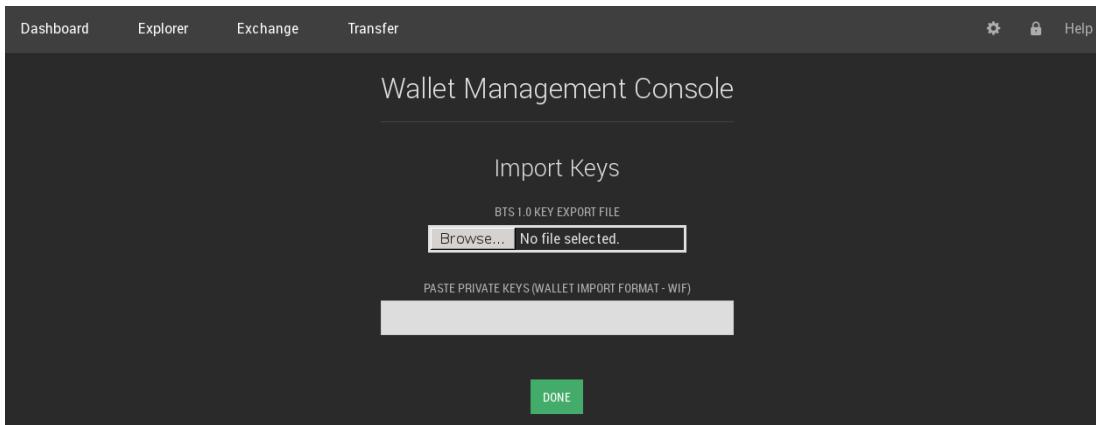


In order to import your existing accounts and claim all your funds you need to choose Import Keys.

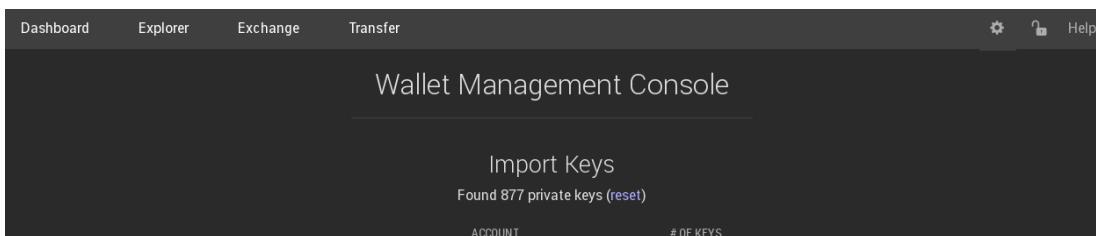
Note: If loading the file files with invalid format please ensure that you have followed the steps described [Exporting Your Wallet](#) and make sure to click Import Keys and **not** Restore Backup.



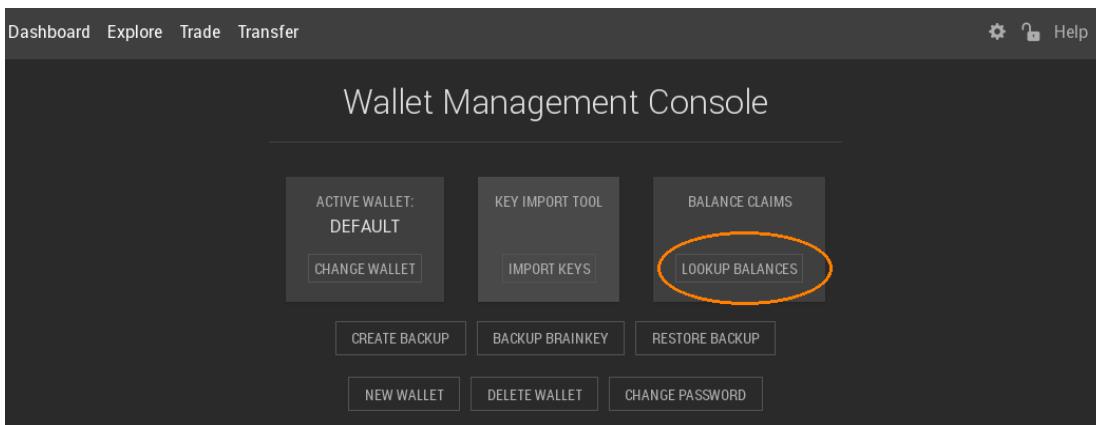
Here you can provide the wallet backup file produced from BitShares 0.9.3c and the pass phrase. Depending on the size of your import file, this step may take some time to auto-complete. Please be patient.



The wallet will list all of your accounts including the number of private keys stored in the account names accordingly. The more often you have used your account, the higher this number should be. Confirm by pressing Import.



The wallet management console will now give an overview over unclaimed balances.



If you click on Balance Claim you will be brought to this screen.

UNCLAIMED	UNCLAIMED (VESTING)	ACCOUNT
RCOIN		sharedrops.xeroc
EWBIE		sharedrops.xeroc
16 USD		sharedrops.xeroc
:CORE		
:FREE		xeroc
:4 EUR		
:CORE		xeroc
:1 BTC		xeroc
TEAUX		xeroc
RCOIN		
:CORE		
:SATM		
:CORE		
:1 USD		
:NOTE		
:CORE		
:CORE		
RCOIN		

You are asked to define where to put your individual balances if you have multiple accounts.

After confirming all required steps, your accounts and the balances should appear accordingly.

Note: After importing your accounts and balances, we recommend to make a new backup of your wallet that will then contain access to your newly imported accounts and corresponding balances.

CLI wallet

The wallet backup file can be imported by

```
>>> import_accounts <path to exported json> <password of wallet you exported from>
```

Note that this doesn't automatically claim the balances.

Claiming Balances

For each account <my_account_name> in your wallet (run `list_my_accounts` to see them)::

```
>>> import_account_keys /path/to/keys.json <my_password> <my_account_name> <my_
˓→account_name>
```

Note: In the release tag, this will create a full backup of the wallet after every key it imports. If you have thousands of keys, this is quite slow and also takes up a lot of disk space. Monitor your free disk space during the import and, if necessary, periodically erase the backups to avoid filling your disk. The latest code only saves your wallet after all keys have been imported.

The command above will only import your keys into the wallet and will **not** claim your funds. In order to claim the funds you need to execute::

```
>>> import_balance <my_account_name> [ "*" ] true
```

Note: If you would like to preview this claiming transaction, you can replace the `true` with a `false`. That way, the transaction will not be broadcast.

To verify the results, you can run::

```
>>> list_account_balances <my_account_name>
```

Manually claim balances

Balances can be imported one by one. The proper syntax to do so is:

```
>>> import_balance <account name> <private key> true
```

But I always import my accounts and then use the GUI to import my balances cause it's way easier.

Migration Remarks

Remarks on Private Keys

Depending on the users (trade) activity and investors behavior please also note the following:

- **TITAN Transactions:** If there is a chance that you might have received a TITAN transaction (default transaction format) since you last opened your wallet, you are required to completely synchronize with the BitShares 1.0 blockchain to catch that transaction and generate the corresponding private key. After that, you can use the graphene-compatible wallet export function.

- **Market transactions:** Each market order is associated a key that is derived when placing your order and is hence part of your wallet. Please note that for the transition to BitShares 2.0, all open orders (except short orders) will be closed and the funds returned to their owners.
- **Cold Storage Funds:** You cold storage funds can be claimed by simply importing your cold private key into BitShares 2.0. This will result in a transaction that claims your funds and puts them into your account.
- **PTS/AGS donators:** You can import your corresponding private keys into BitShares 2.0 directly to claim your funds. Note that, since the web wallet cannot parse your *wallet.dat* file, you may be required to manually dump your private keys. with a 3rd party tool. If you feel not comfortable with this, we recommend you import your *wallet.dat* file into the BitShares 1.0 client and export a graphene-compatible wallet file that can be imported in the BitShares 2.0 web wallet.

Technical Explanation

The BitShares 2.0 wallet architecture is vastly different than BitShares 1.0. In BitShares 1.0, each account consists of dozens or even thousands of keys, each of which is controlling a small portion of your balance and for TITAN users, none of the balances associated with your account use the same key as your account. Under BitShares 2.0, all of these “balances” become unique accounts rather than a single logical account. The BitShares 2.0 wallet has an “import” interface that allows you to specify a set of private keys and the name of an account that you would like to receive all of the funds associated with those keys. Then it generates a transaction that spends the full balances from all accounts associated with those keys to your new unified BitShares 2.0 account. The BitShares 0.9.x wallet provides a utility to dump all private keys associated with a given account to make the migration process easy.

User Guide

For End User

We would like to introduce most features to the user in a friendly and easy to understand way in this *User Guide*. Note, that the blockchain itself is capable of many more things that are not (yet) exposed to the end-user. In the future, we expect to see many more customized wallets that expose different features to their users.

First Steps for End-Users

This guide gives a quick introduction of how to use BitShares as an end-user.

Learn about BitShares

BitShares is different from anything you have experienced yet and as such, a user should know about advantages, risks, and opportunities. As a starter, we highly recommend to read through the [Things you should know](#) to get a first impression of what BitShares is and how it is different to existing systems.

Choose your Client

Several ways existing to enter the BitShares network that focus on different aspects. To actually interact with the BitShares ecosystem, you can either

- download the Official Light Client
- or access the network in the browsers via one of our partners:

- [OpenLedger](#)
- [BitShares.org](#)
- [Decentral.exchange](#)
- [BunkerDex \(under construction\)](#)
- *more to come*

All of these solutions have one thing in common: **You have sole control of your accounts and funds** and they are created on your computer (within the light-client or the browser). Hence, you will only be able to access your account on the computer that you have used to register and create your account, unless you export your wallet and import it somewhere else.

Accounts, Wallets, Keys: Terminology!

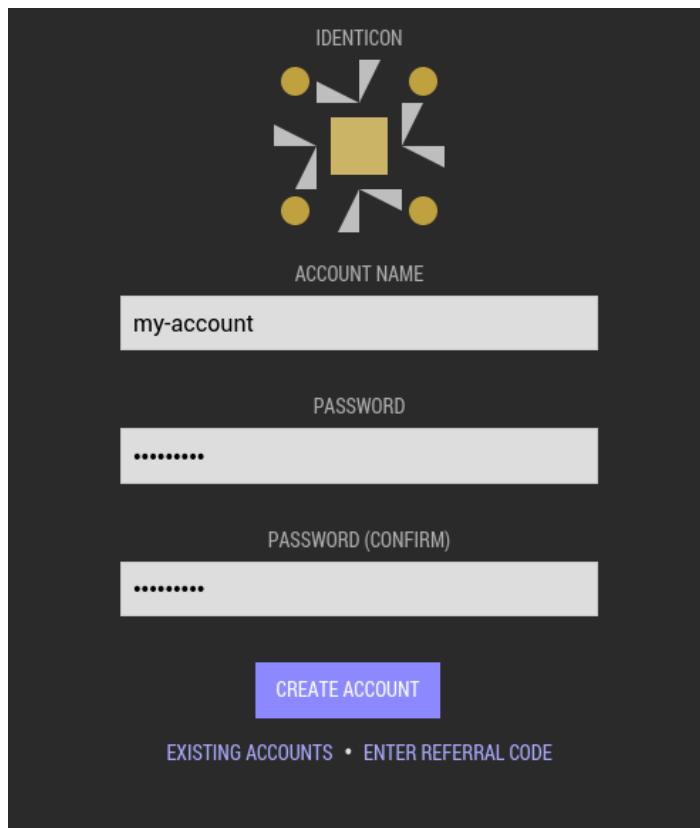
Most clients distinguish between accounts from wallets and all of them use keys to access funds. Let's quickly clarify the terms here:

- **Keys:** Keys refer to the cryptography used to secure access to your account and funds. It is of importance to prevent others from gaining access to these. This is why you have to provide a passphrase that is used to store the keys in an encrypted way.
- **Accounts:** Each user has at least one account that can be used to interact with the blockchain. In the end, this can be seen as a single banking account with an individual balance, transaction history, etc. Since these accounts are registered on the blockchain and are open to the public, we recommend to pick a pseudonym to achieve some privacy. The advantage of using account names is that people can identify you by using a readable and memorable word instead of cryptographic addresses.
- **Wallet:** Since users can (if they have a lifetime membership) register multiple accounts in parallel, all of them are stored in a single wallet. Hence, a wallet can carry many accounts. Furthermore, users can create multiple wallets to organize their accounts properly.

Create an Account

In order to use BitShares, you will need to register an account. All you need to provide is an account name and a password to secure your wallet:

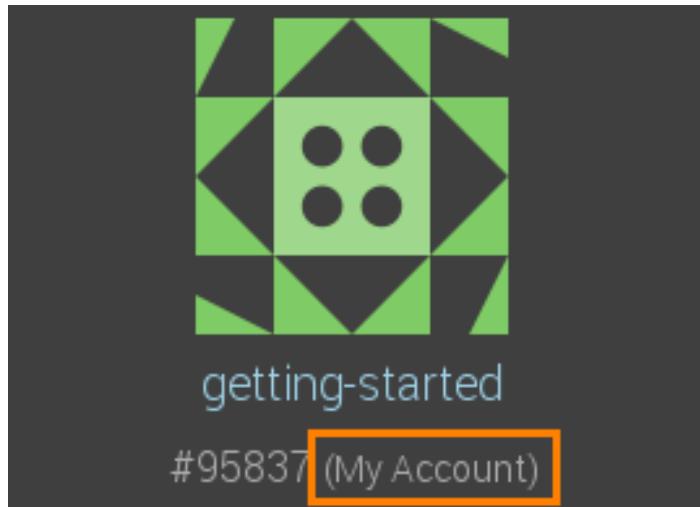
The *identicon* at the top can be used to verify your account name to third parties. It is derived from your account name and gives a second verification factor. And this is how you register your account:



In contrast to any other platform you have ever used: **Creating an account at one of our partners will make your account available at all the other partners as well.** Hence, your account name can be seen similar to a mail address in such that it is **unique** and every participant in the BitShares network can interact with you independent of the actual partner providing the wallet.

After creation of your account, you will see an **Account** link in the top navigation bar to browse to your account.

Note: Whether you control an account or not can be seen from the account's overview. If it states (**Your Account**), you do have the required keys installed in your wallet to access its funds. Otherwise, you can only view the account but cannot transact from it!

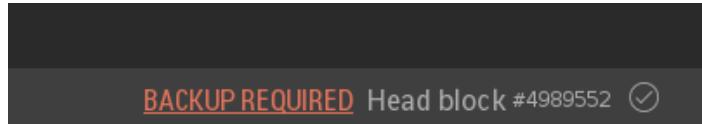


Backup your account

Since you are the only individual that has access to your account and funds, it is **your responsibility** to make a secure backup of your registered account.

Once you have registered your account, you can click the warning in the footer to directly enter the backup page of your wallet:

1. Click the *Backup required* link in the footer



2. Click **Create Backup**
3. Click **Download** and store the file **safely**. Make sure to **remember the passphrase** you provided when you created your wallet (above) as the downloaded file is encrypted with it.
4. (*optionally but recommended*) Note the **xxxxxx * SHA1 checksum** to verify the backup

Fund your account

In order to fund your account you have two options:

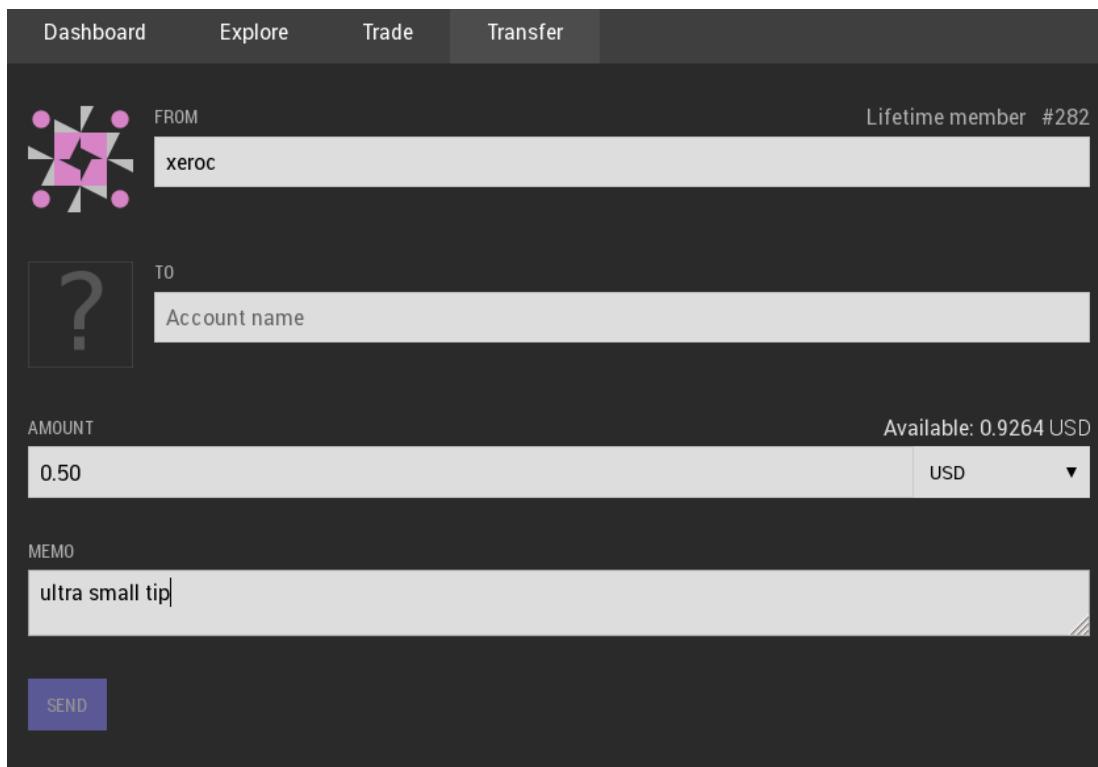
- **Transfers:** Ask a partner or exchange to send funds to you. To do so, you will only need to **provide them with your account name**.
- **Deposits:** By visiting your **deposit/withdraw** page in your account's navigation, you can use one of our partners to move over existing funds into your BitShares account.

In order to understand the meanings of different assets, we recommend you to read through our [assets page](#).

Commonly used Features

Moving Funds

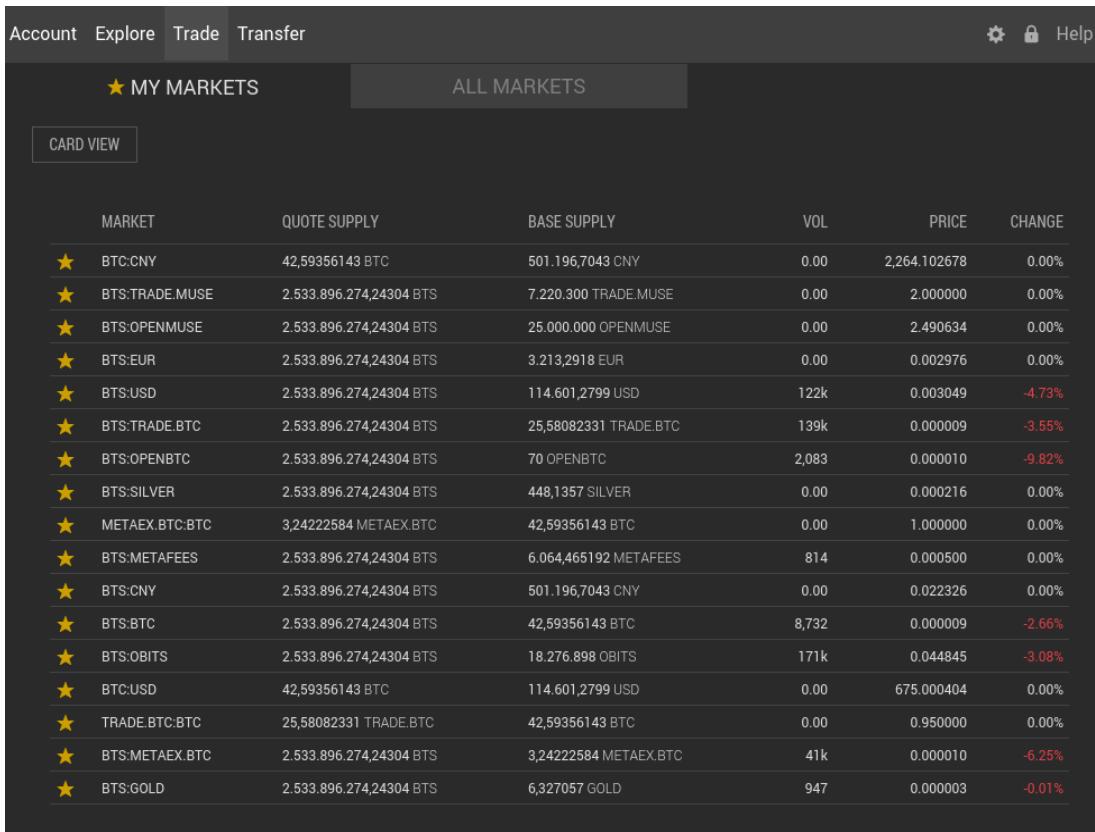
Using the *transfer* tool available from the main menu bar allows to move funds from your account to an arbitrary other account. If you enter the transfer page, your account name will be pre-filled into the source field. You will need to provide the **recipient's account name**, the amount and asset to transfer and can optionally add a memo to help the recipient to identify your transfer (the memo is encrypted and only you and the recipient can read it).



Note: On the BitShares blockchain, people never need to deal with *addresses* or *public keys* but can instead use account names. Your account name becomes the *email address* for your funds.

Trading in the Decentralized Exchange (DEX)

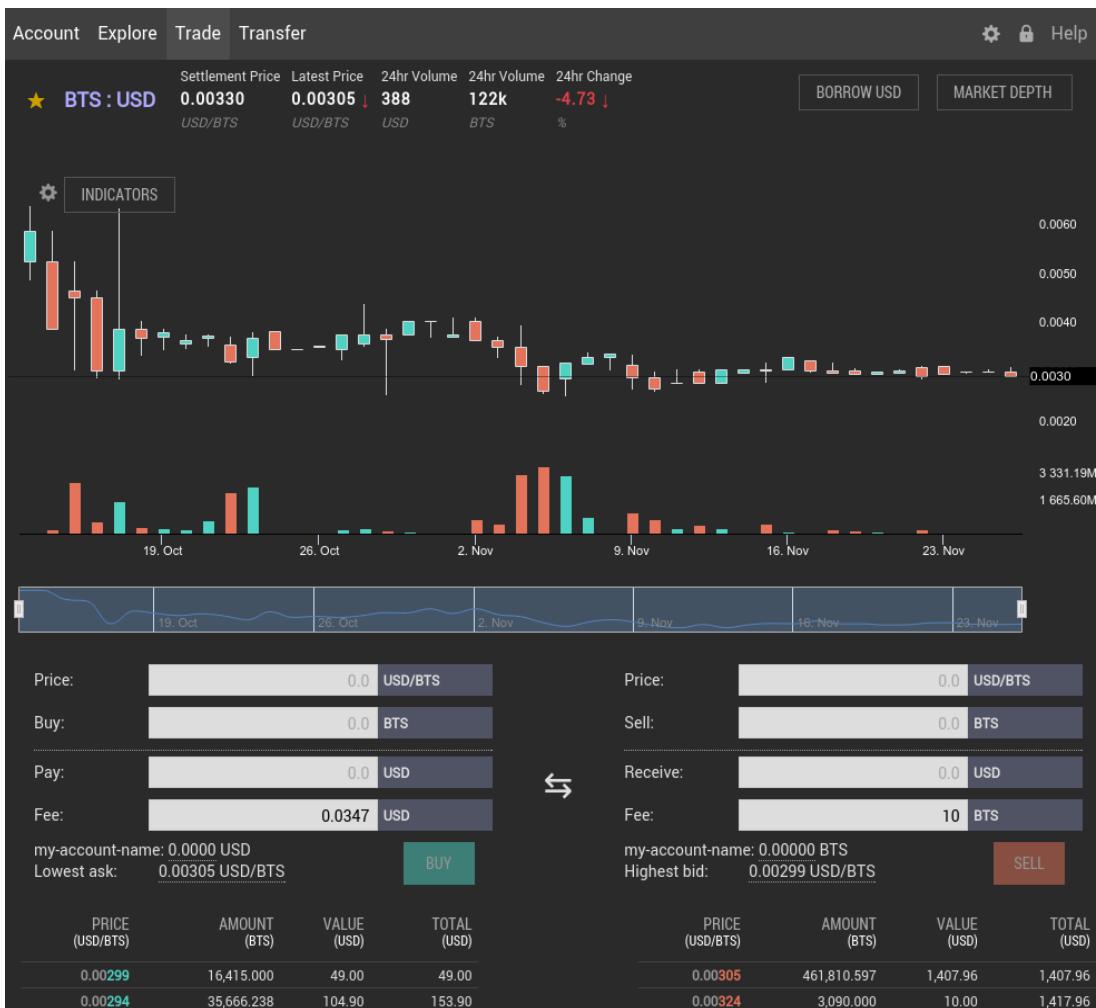
To trade your assets into other assets you can use the built-in decentralized exchange (often referred to as *the DEX*). It is available using the **Trade** menu item and shows a list of commonly used markets.



The screenshot shows the Graphene interface with the 'MY MARKETS' tab selected. The table below lists various markets with the following columns:

MARKET	QUOTE SUPPLY	BASE SUPPLY	VOL	PRICE	CHANGE
★ BTC:CNY	42,59356143 BTC	501,196,7043 CNY	0.00	2,264.102678	0.00%
★ BTS:TRADE.MUSE	2,533,896,274,24304 BTS	7,220,300 TRADE.MUSE	0.00	2,000000	0.00%
★ BTS:OPENMUSE	2,533,896,274,24304 BTS	25,000,000 OPENMUSE	0.00	2,490634	0.00%
★ BTS:EUR	2,533,896,274,24304 BTS	3,213,2918 EUR	0.00	0.002976	0.00%
★ BTS:USD	2,533,896,274,24304 BTS	114,601,2799 USD	122k	0.003049	-4.73%
★ BTS:TRADE.BTC	2,533,896,274,24304 BTS	25,58082331 TRADE.BTC	139k	0.000009	-3.55%
★ BTS:OPENBTC	2,533,896,274,24304 BTS	70 OPENBTC	2,083	0.000010	-9.82%
★ BTS:SILVER	2,533,896,274,24304 BTS	448,1357 SILVER	0.00	0.000216	0.00%
★ METAEX.BTC:BTC	3,24222584 METAEX.BTC	42,59356143 BTC	0.00	1,000000	0.00%
★ BTS:METAFEESES	2,533,896,274,24304 BTS	6,064,465192 METAFEESES	814	0.000500	0.00%
★ BTS:CNY	2,533,896,274,24304 BTS	501,196,7043 CNY	0.00	0.022326	0.00%
★ BTS:BTC	2,533,896,274,24304 BTS	42,59356143 BTC	8,732	0.000009	-2.66%
★ BTS:OBITS	2,533,896,274,24304 BTS	18,276,898 OBITS	171k	0.044845	-3.08%
★ BTC:USD	42,59356143 BTC	114,601,2799 USD	0.00	675,000404	0.00%
★ TRADE.BTC:BTC	25,58082331 TRADE.BTC	42,59356143 BTC	0.00	0.950000	0.00%
★ BTS:METAEX.BTC	2,533,896,274,24304 BTS	3,24222584 METAEX.BTC	41k	0.000010	-6.25%
★ BTS:GOLD	2,533,896,274,24304 BTS	6,327057 GOLD	947	0.000003	-0.01%

By clicking any of the rows, you enter a particular market in which you can participate by selling or buying from the market or by placing a orders into the order books at **your** price.



Once an order is filled, the corresponding asset will appear in your balance immediately. In BitShares, clearing and settlement are performed instantaneously.

The figure shows two side-by-side order forms for Steem. The left form is for 'BUY STEEM' and the right is for 'SELL STEEM'. Both forms have fields for Price (0.0 BTC*), Amount (0.0 STEEM), Total (0.0 BTC*), and Fee (0.14676 BTS). Below these are 'Balance' and 'Lowest ask' fields for buying, and 'Balance' and 'Highest bid' fields for selling. A central '↔' symbol indicates the two forms are related.

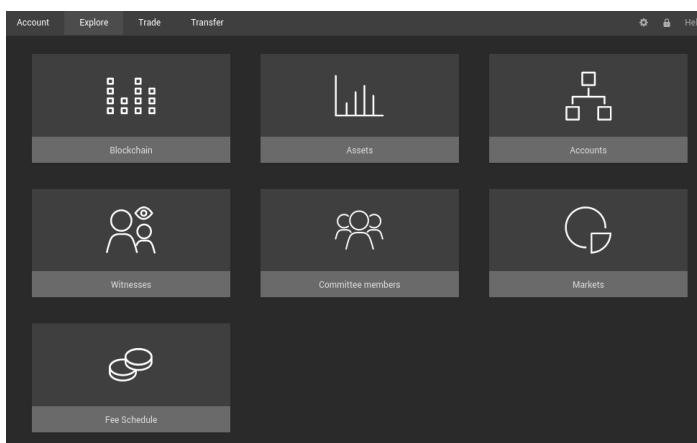
Read more:

- [Decentralized Exchange](#)
- [Trading](#)

Exploring the Blockchain

The blockchain and business can be investigated using the built-in explorer from the main navigation bar. There you can take a closer look at:

- the blockchain,
- the available assets,
- registered accounts,
- witnesses,
- committee members,
- markets, and
- the fee schedule.



Things you should know

The Starters

- **Security and Control over accounts and funds:** No one can access your funds unless you let them, intentionally, or unintentionally. With the power to be independent from 3rd parties, comes the responsibility to *protect what belongs to you*.
- **Can interact with people directly:** With BitShares it becomes possible to interact with people directly without needing to go through a middleman. Hence, BitShares is a platform of free speech that implements a payment platform and exchange for digital goods.
- **Fast:** Transactions in BitShares are verified and irrevocable in only a few seconds time.
- **Decentralized Committee:** Decisions that can effect the BitShares ecosystem are made using a on-chain committee voted upon by shareholders. Hence, no single entity can change the deal retroactively.
- **Flexible:** Protocol upgrades (formerly known as *hard forks*) can be implemented and executed to improve the BitShares business over time and allow to react on external influences quickly.

The Investors

- **Become Shareholder:** If you buy BTS either from a partner exchange or from the DEX, you become a shareholder of the BitShares decentralized business and as such can take a cut of its profits and participate in votes

for future directions.

- **Expenses:** Vote for expenses of the business and hire workers to do important tasks for BitShares.
- **Leaders:** Participate in political decisions by voting for committee members that represent your views!
- **Protocol upgrades:** Improve the technology, integrate new features and adept legal and regulative changes by voting for upgrades.
- **Decision making for a profit:** Take part in decision finding about fair pricing models for transaction fees to a) increase growth and b) make BitShares profitable for its shareholders

Technological Aspects

Accounts

BitShares 2.0 accounts have to be registered on the blockchain. Upon registration they are assigned an incrementing identifier (account id).

This comes with many advantages: Besides improved scalability, we have separated the identity from the transaction authorizing signature. In practice, *owning an account name* is independent from being able to *spend its funds*. Furthermore, both rights (we call them *permissions*) can split among an arbitrary complex relation of people (we call them *authorities*) using *weights* and a required *thresholds*.

Thanks to separating *authorities* from *identities*, BitShares 2.0 can be much faster in processing delay while having much smaller transaction sizes. Hence, all participants are forced to have a named account on the blockchain. Furthermore, most transactions are tied to an account name and can thus be linked to individuals (this includes transfers, trades, shorts, etc. but not *stealthed* transactions).

Note: Even though an account is required to be registered properly, we offer tools to improve privacy and anonymity.

Account Registration

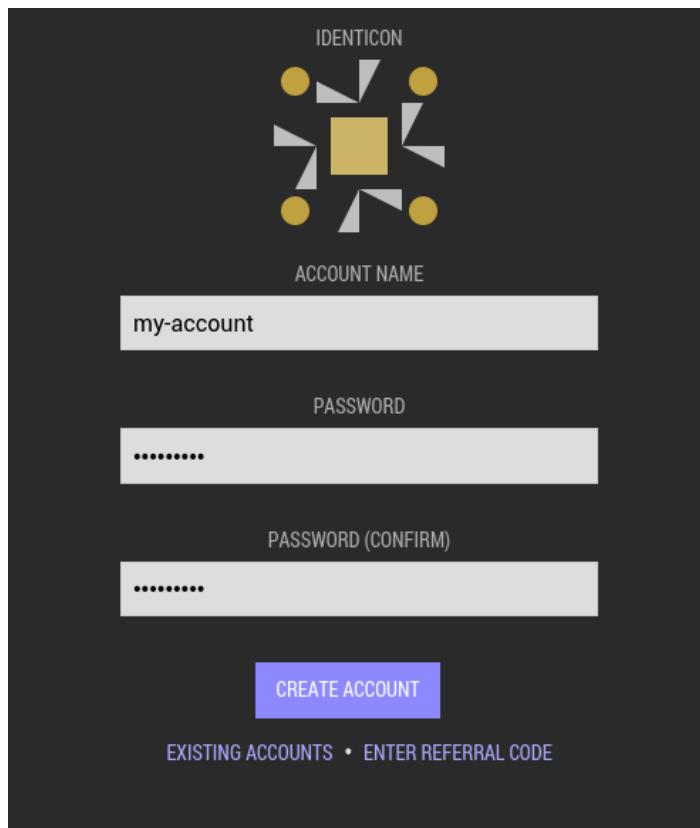
In order to use BitShares, you will need to register an account. All you need to provide is

- an account name
- a password

For regular users, please follow the instructions in the wallet to create a new account:

- OpenLedger
- BitShares.org
- Decentral.exchange
- BunkerDex (under construction)
- *more to come*

The identicon at the top can be used to verify your account name to third parties. It is derived from your account name and gives a second verification factor. And this is how you register your account:



Note that, in contrast to any other platform you have ever used: **Creating an account at one of our partners will make your account available at all the other partners as well.** Hence, your account name can be seen similar to a mail address in such that it is **unique** and every participant in the BitShares network can interact with you independent of the actual partner providing the wallet.

We also provide a [tutorial](#) on how to **manually** register an account using the *CLI wallet*.

Memberships

Accounts in BitShares are separated into three groups. We decided to give users the option to upgrade their accounts into a VIP-like status if they desire and profit from reduced fees and additional features.

Membership Groups

Non-Members

A *regular* account is a *non-member*.

Lifetime Members

Lifetime Members get a percentage cashback on every transaction fee they pay and qualify to earn referral income from users they register with or refer to the network. A Lifetime membership is associated with a certain one-time fee that is defined by the committee.

Anual Members

If a lifetime membership is too much you can still get the same cashback for the next year by becoming an annual subscriber for a smaller one-time fee which lasts for only one year.

Note: Technically, the fees that you pay stay the same, but a part of the fees is refunded in the form of a *vesting balance*. Once the fees have vested you can withdraw them. To see your vesting balances, go to your “Account” tab, then click on “vesting balances” at the bottom left. Vesting balances are recalculated hourly, so you might not yet see them right away.

Warning: Due to some discrepancies, the annual membership has been disabled in most web wallets and will be re-enabled after a proper update eventually.

Fees

Every time an account you referred pays a transaction fee, that fee is divided among several different accounts. The network takes a cut, and the Lifetime Member who referred the account gets a cut.

The registrar is the account that paid the transaction fee to register the account with the network. The registrar gets to decide how to divide the remaining fee between themselves and their own affiliate.

Pending Fees

Fees paid are only divided among the network, referrers, and registrars once every maintenance interval.

Vesting Fees

Most fees are made available immediately, but fees over the vesting threshold (such as those paid to upgrade your membership or register a premium account name) must vest for some days as defined by the committee.

Permissions

In BitShares, each account is separated into

- **Owner Permission:** This permission has administrative powers over the whole account and should be considered for ‘backup’ strategies.
- **Active Permission:** Allows to access funds and some account settings, but cannot change the owner permission and is thus considered the “online” permissions.

Both can be defined in the *Permissions* tab of your account using so called *authorities* (see below) together with a so called *threshold* that has to be exceeded in order for a transaction to be valid.

Authorities

In BitShares an *authority* consists of one or many entities that authorize an action, such as transfers or trades.

An authority consists of one or several pairs of an account name with a *weight*.

In order to obtain a valid transaction, the sum of the weights from signing the parties has to exceed the threshold as defined in the permissions.

Examples

Let's discuss some examples to shed some light on the used terminology and the use-cases. We assume that a new account is created with its active permissions set as described below. Note that the same scheme also works for the owner permissions!

(Flat) Multi-Signature

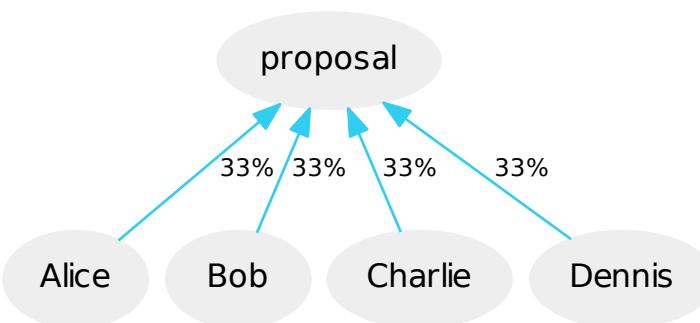
A flat multi-signature scheme is composed of M entities of which N entities must sign in order for the transaction to be valid. Now, in BitShares, we have *weights* and a *threshold* instead of M and N . Still we can achieve the very same thing with even more flexibility as we will see now.

Let's assume, Alice, Bob, Charlie and Dennis have common funds. We want to be able to construct a valid transaction if only two of those agree. Hence a **2-of-4** (N -of- M) scheme can look as follows:

Account	Weight
Alice	1
Bob	1
Charlie	1
Dennis	1
Threshold:	3

This means that each party has the same weight of 1 while 3 parties need to sign the transaction/proposal.

In other words: Alice, Bob, Charlie and Dennis, each have 33% weight while 100% must be reached.



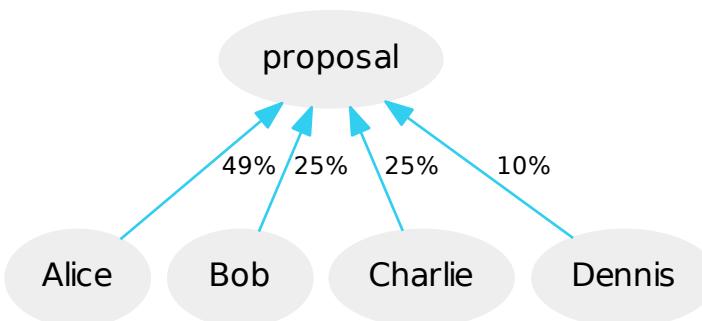
All four participants have a weight of 33% but the threshold is set to 51%. Hence only two out of the four need to agree to validate the transaction.

Alternatively, to construct a 3-of-4 scheme, we can either decrease the weights to 17 or increase the threshold to 99%.

(Flat) Flexible Multi-Signature

With the threshold and weights, we now have more flexibility over our funds, or more precisely, we have more *control*. For instance, we can have separate weights for different people. Let's assume Alice wants to secure her funds against theft by a multi-signature scheme but she does not want to hand over too much control to her friends. Hence, we create an authority similar to:

Account	Weight
Alice	49%
Bob	25%
Charlie	25%
Dennis	10%
Threshold:	51%

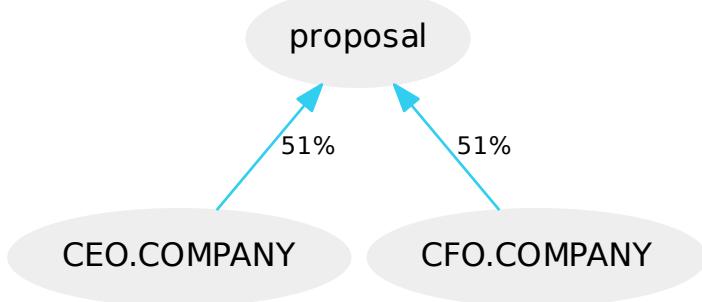


Now the funds can either be accessed by Alice and a single friend or by all three friends together.

Multi-Hierarchical Flexible Multi-Signature

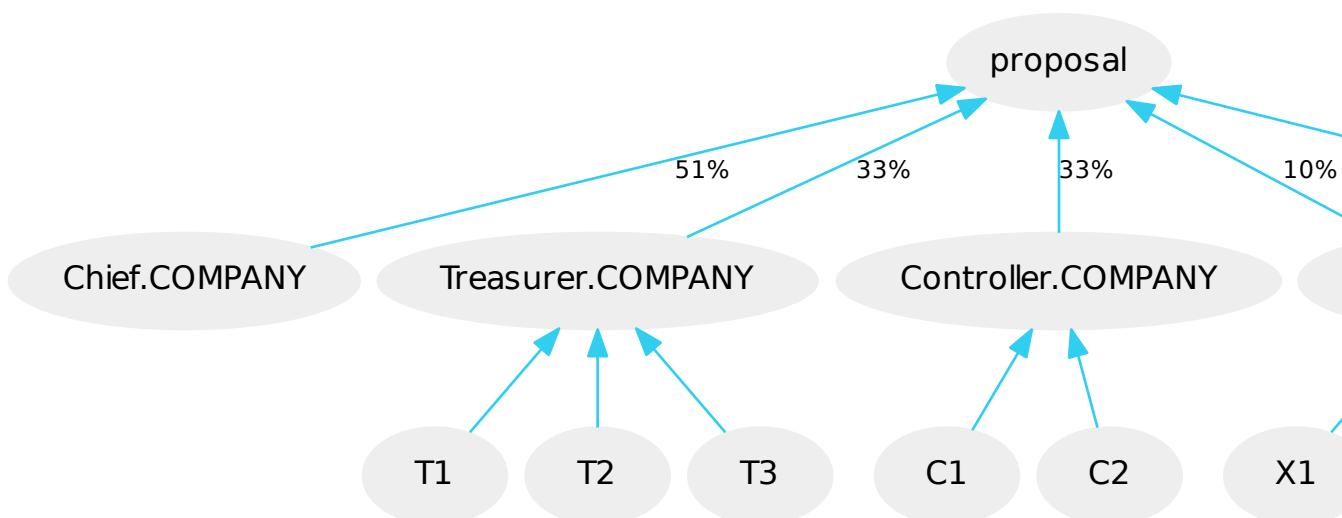
Let's take a look at a simple multi-hierarchical corporate account setup. We are looking at a company that has a Chief of Financial Officer (CFO) and some departments working for him, such as the Treasurer, Controller, Tax Manager, Accounting, etc. The company also has a CEO that wants to have spending privileges. Hence we construct an authority for the funds according to:

Account	Weight
CEO.COMPANY	51%
CFO.COMPANY	51%
Threshold:	51%



whereas CEO.COMPANY and CFO.COMPANY have their own authorities. For instance, the CFO.COMPANY account could look like:

CFO.COMPANY	Weight
Chief.COMPANY	51%
Treasurer.COMPANY	33%
Controller.COMPANY	33%
Tax Manager.COMPANY	10%
Accounting.COMPANY	10%
Threshold:	51%



This scheme allows:

- the CEO to spend funds
- the Chief of Finance Officer to spend funds
- Treasurer together with Controller to spend funds
- Controller or Treasurer together with either the Tax Manager or Accounting to spend funds.

Hence, a try of arbitrary depth can be spanned in order to construct a flexible authority to reflect mostly any business use-case.

Security

Since BitShares 2.0 is about financial freedom and securing wealth, we take users concerns about application and network security serious.

Permission Model

We distinguish between two kinds of permissions:

- **Owner Permission:** This permission has administrative powers over the whole account and should be considered for ‘backup’ strategies.
- **Active Permission:** Allows to access funds and some account settings, but cannot change the owner permission and is thus considered the “online” permissions.

Both are implemented using [Elliptic Curve Cryptography](#) (ECC) with *public* and *private* keys. [Read more](#).

You can find the keys for your permissions in the Permissions tab:

Active Permissions		
ACCOUNT/KEY	WEIGHT	ACTION
BTS7ySDFgAMfaCKSiopuA36yLpec4ZpgDL1y7UTyx4E82Hf8iXuQq	1	REMOVE

Owner Permissions		
ACCOUNT/KEY	WEIGHT	ACTION
BTS5rtmjipgFxxgUhRGwWMLzczwbnzNvkiFrc9zjpre2sjno486Z	1	REMOVE

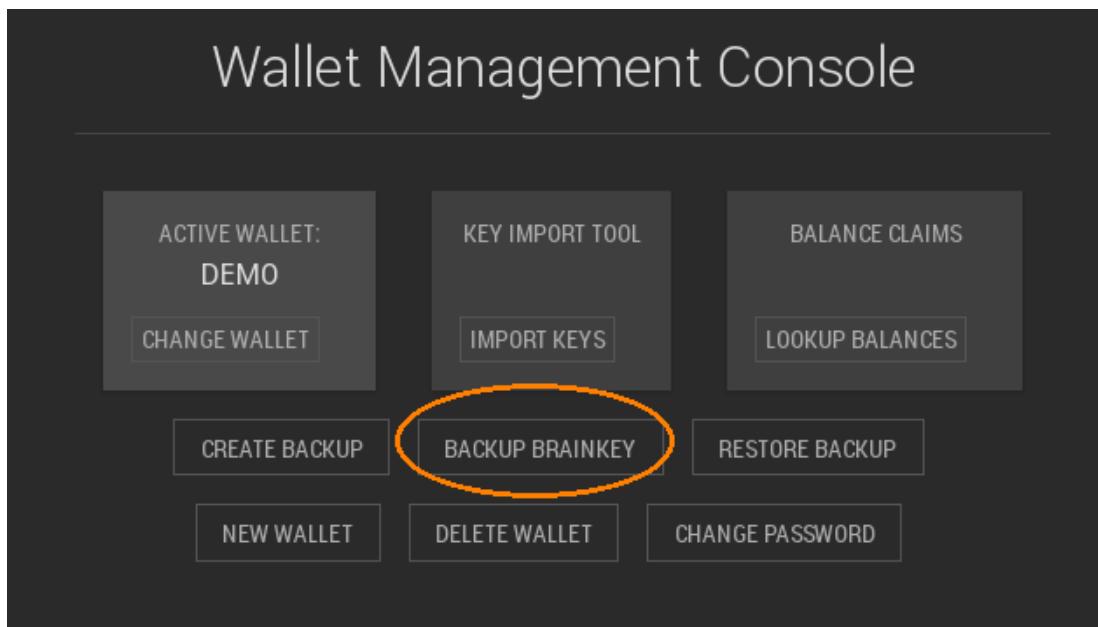
Memo Key		
MEMO PUBLIC KEY	Valid Public Key	
BTS7ySDFgAMfaCKSiopuA36yLpec4ZpgDL1y7UTyx4E82Hf8iXuQq		

The Brain Key

The *brain key* is used as source for all cryptographic keys generated in the wallet. If you have it secured, you will be able to regain access to your accounts and funds (unless the access keys have been changed)

Backing Up the Brain Key

The brain key can be backed up as a string using the *Wallet Management Console** in your settings.



Restoring from Brain Key

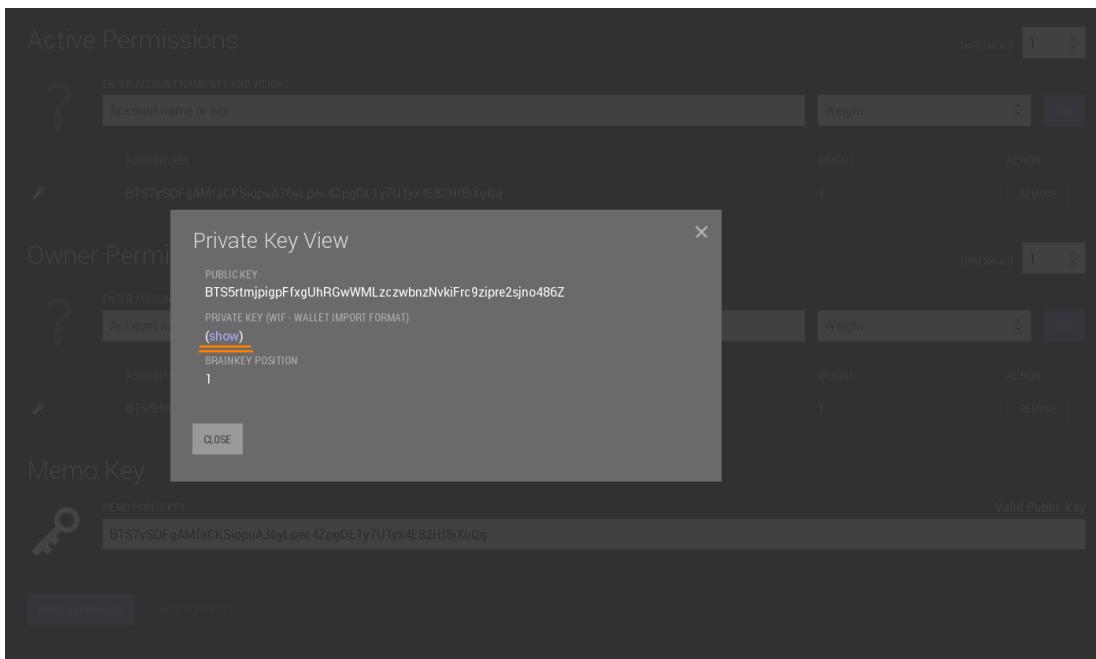
Your wallet can be restored with the brain key from the GUI. When creating a new wallet there is a link at the bottom, Existing Accounts. From there select Create Wallet and then Custom Brain Key (Advanced) from the bottom.

Securing “Cold” Wallets

Obtaining A Private Key

Exporting Existing Private Key

Existing private keys can be obtained from the Permissions tab in the wallet.



Note that this approach should not be considered **cold** as the private keys has been installed in the browser first.

Offline Webpage

Alternatively, you can use a distinct web project that sole purpose is to provide a public, private key pair. It can be downloaded from [github](#) and should be opened in a browser on a non-connected device. The page generated a random private key, derives the corresponding public key and shows it in plain text and as QR code.

Note: Either copy&paste the private key or print the resulting page because the private key cannot be regenerated!

Updating Account Permissions

As described in the [account permissions](#) and [security permissions](#) in theory, we now have to update the accounts permissions.

Note: To be able to update the accounts permissions we need the **owner key**. Hence, if your owner key is compromised the active key can be updated and you will be locked out of your funds.

Make sure to give it a weight that is higher than the threshold. If you want that key to be the sole key for accessing the funds, remove all other keys from the active permissions tabs.

The screenshot shows the Graphene wallet management interface. It includes three main sections:

- Active Permissions:** A table with columns for ACCOUNT/KEY, WEIGHT, and ACTION. One row is highlighted with a yellow circle around the account key "BTS6WM4H2Sb6EeJ4xTTzgyU5qFPrk5MHh5ujRKg8SzK88axViEpZx" and weight "1". Buttons for "Public Key" and "ADD" are visible.
- Owner Permissions:** A table with columns for ACCOUNT/KEY, WEIGHT, and ACTION. One row is highlighted with a yellow circle around the account key "BTS5WAszCsqVN9hDkXZPMyiUib3dyrEA4yd5kSMgu28Wz47B3wUqa" and weight "1". Buttons for "REMOVE" are visible.
- Memo Key:** A section with a "MEMO PUBLIC KEY" input field containing "BTS5TPTziKkLExhVKsQKtSpo4bAv5RnB8oXcG4sMHEwCcTf3r7dqE". Buttons for "PUBLISH CHANGES" (circled in blue) and "RESET CHANGES" are present.

Importing Keys

Private keys can be reimported using the wallet management console. Then funds and account features will be available to your wallet again:

The screenshot shows the "Wallet Management Console" with the "Import Keys" section active. It includes:

- A "BTS 1.0 KEY EXPORT FILE" input field with a "Browse..." button and a message "No file selected."
- A "PASTE PRIVATE KEYS (WALLET IMPORT FORMAT - WIF)" input field with a large redacted content area.
- A green "DONE" button at the bottom right.

Recover account with brain key

If you have the brain key, then you can reclaim access to your account(s) and its funds by

1. Open the Settings
2. Enter the wallet management console
3. Click on “New wallet”
4. Click “Custom Brainkey (advanced)”
5. Provide a new (or the old) passphrase

6. provide the brain key
7. Create the new wallet

Your account should now be accessible from the top navigation bar. If you only have one account, you will see **Account**, else you will see **Dashboard**.

Transactions and Operations

BitShares 2.0 implements a variety of operations besides simple transfers of funds. For instance in order to operate the *decentralized exchange* we need support for buy, sell orders as well. The technical documentation about all operation types can be found in our doxygen docs.

Simple Transfers

A simple transfer operation moves funds from user A to user B. In contrast to most other blockchain-based financial networks, we do **not** use *addresses* or *public keys* for transfers.

The screenshot shows the BitShares 2.0 Transfer interface. At the top, there is a navigation bar with tabs: Dashboard, Explore, Trade, and Transfer. The Transfer tab is active. Below the navigation bar, the 'FROM' field is populated with the account name 'xeroc'. To the right of the 'FROM' field, it says 'Lifetime member #282'. The 'TO' field is labeled 'TO' and contains the placeholder 'Account name'. Below the 'TO' field is a large input area. On the left side of this area, there is a question mark icon. Inside this area, the 'AMOUNT' field contains '0.50', and the 'Available' balance is listed as '0.9264 USD'. To the right of the amount, there is a dropdown menu set to 'USD'. Below the amount, the 'MEMO' field contains the text 'ultra small tip'. At the bottom of the transfer form is a blue 'SEND' button.

Instead, all that is needed for transfers is:

- source account name
- destination account name
- funds (amount and asset)
- memo (optional)

A transfer may contain a memo with arbitrary text.

Note: The `memo` is **encrypted** by default can only be decrypted by the participants of the transfer! The transfer fee depends on the length of the memo!

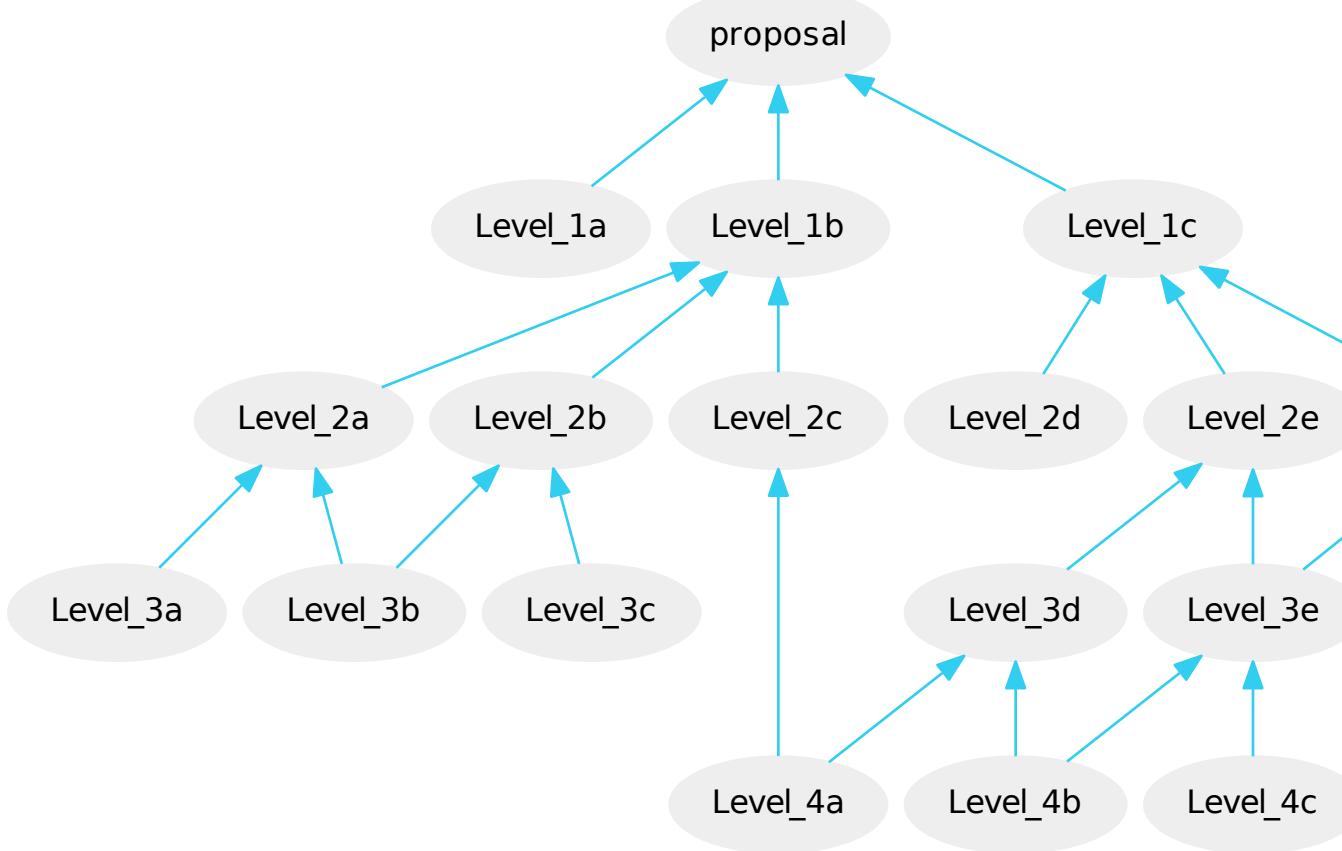
Proposed Transactions

The Graphene technology allows users to *propose* a transaction on the blockchain which requires approval of multiple accounts in order to execute.

At any time, a proposal can be approved in a single transaction if sufficient signatures are available (see `proposal_update_operation` as constructed by the `approve_proposal` call), as long as the authority tree to approve the proposal does not exceed the maximum recursion depth. In practice, however, it is easier to use proposals to acquire all approvals, as this leverages on-chain notification of all relevant parties that their approval is required. Off-chain multi-signature approval requires some off-chain mechanism for acquiring several signatures on a single transaction. This off-chain synchronization can be avoided using proposals.

The user proposes a transaction, then signatory accounts use add or remove their approvals from this operation. When a sufficient number of approvals have been granted, the operations in the proposal are evaluated. Even if the transaction fails, the proposal will be kept until the expiration time, at which point, if sufficient approval is granted, the transaction will be evaluated a final time. This allows transactions which will not execute successfully until a given time to still be executed through the proposal mechanism. The first time the proposed transaction succeeds, the proposal will be regarded as resolved, and all future updates will be invalid.

The proposal system allows for arbitrarily complex or recursively nested authorities. If a recursive authority (i.e. an authority which requires approval of ‘nested’ authorities on other accounts) is required for a proposal, then a second proposal can be used to grant the nested authority’s approval. That is, a second proposal can be created which, when sufficiently approved, adds the approval of a nested authority to the first proposal. This multiple-proposal scheme can be used to acquire approval for an arbitrarily deep authority tree.



Note that each account in the figure can carry a **different weight**. An example of how to setup your permissions accordingly is given in [Permissions](#).

Confidential Transactions

A confidential transfer is one that hides the amount being sent. Confidential transfers are also referred to as blinded transfers. It makes use of Oleg Andreev's [blind signatures](#).

When privacy is important no account is ever used twice and it is impossible for any third party to identify how money is moving through blockchain analysis alone.

Note: Confidential transactions are currently only available using the [cli-wallet](#). A step-by-step guide can be found in the [tutorials](#)

Assets/Tokens

The BitShares 2.0 network consist of several *assets*, *tokens* or *currencies*. All assets are equal from a technological point of view and come with more or less the same features, namely, they can be traded against each other and can be transferred within seconds. The differences between them are of economical nature.

User Issued Assets (UIAs)

Freely traded tokens created by individuals used for a variety of use-cases, such as stock, miles, event tickets or reputation points.

[Read more ...](#)

Market-Pegged Assets (MPA)

These *SmartCoins* track the value of an underlaying asset, such as Gold, or U.S. Dollar. Smartcoins can be created by anyone contracting with the BitShares ecosystem and putting sufficient BTS (at least 175%) into the so called collateralized loans as *collateral*.

[Read more ...](#)

Exchange Backed Assets (EBA)

This kind of asset is commonly known as *I owe you* (IOU). It represents the right to withdraw the *same amount* (minus fees) of a backing asset from a *central* entity. Often they are issued by a bank, an exchange or an other financial institute to represent deposit receipts.

[Read more ...](#)

Privatized Bit-Assets

A flexible mixture between UIA and MPA that allows 3rd parties to create their own customized MPAs.

[Read more ...](#)

Fee Backed Assets

An FBA is a token that pays you a fraction of the transaction fees generated by a particular feature that has been funded independent of BitShares.

[Read more ...](#)

Prediction Market Asset

A prediction market is similar to a MPA, that trades between 0 and 1, only. After an event, a price feed can be used to determine which option to take and participants can settle at this price.

[Read more ...](#)

Frequently Asked Questions

Assets FAQ

General

What happens to the asset creation fee?

50% of the asset creation fee are used to pre-fill the assets fee pool. From the other 50%, 20% go to the network and 80% go to the referral program. This means, that if you are a life-time member, you get back 40% of the asset creation fee after the vesting period (currently 90 days).

Can I change x after creation of the asset

The following parameters can be changed after creation:

- Issuer
- UIA-Options:
- Max Supply
- Market Fee
- Permissions (disable only/nor re-enable)
- Flags (if permissions allow it)
- Core exchange rate
- White/Black Listing
- Description
- MPG-Options:
- Feed Life Time
- Minimum Feeds
- Force Settlement Offset/Delay/Volume

Things that cannot be changes:

- Symbol
- Precision

A tutorial can be found [here](#).

What about Parent and Child assets?

A **parent/child** relation ship for assets can be represented by the name of the symbol, e.g.:

```
PARENT.child
```

can only be created by the issuer of PARENT and no one else.

Changing the issuer

The current issue of an asset may transfer ownership of the asset to someone else by changing the issuer in the asset's settings.

Fee Pool

What is the fee pool all about?

The fee pool allows participants in the network to deal with assets and pay for the transaction fees without the need to hold BTS. Any transaction fee can be paid by paying any asset that has a core exchange rate (i.e. a price) at which the asset can be exchange implicitly into BTS to cover the network fee. If the asset's fee pool is funded, the fees can be payed in the native UIA instead of BTS.

Note: The core exchange rate at which a fee can be exchanged into BTS may differ from the actual market valuation of the asset. A user, thus, may pay a premium or spare funds by paying in BTS.

Warning: Make sure your core exchange rate is higher than the lowest ask, otherwise, people will buy your token from the market and drain your fee pool via implicit arbitrage.

It is the task of the issuer to keep the fee pool funded and the core exchange rate updated unless he wants the owner of his asset to be required to hold BTS for the fee.

What to do if the fee pool is empty?

Open up the issuer's account, click the assets tab and open up the dialog to change the asset. There will be a fee pool tab that allows you to fund the fee pool and claim the accumulated fees!

What is Fee Pool Draining?

If an order is created and paid in a non-BTS asset, the fee is implicitly exchange into BTS to pay the network fee. However, if the order is canceled, 90% of the fee will be returned as BTS. The result is, that if the core exchange rate is lower than the highest bid, people can simply buy your token from the market, and exchange them implicitly with the fee pool by creating and canceling an order. This will deplete the fee pool and leave the issuer with his tokens at a slight loss (depending on the offset of the core exchange rate). For this reason, we recommend to use a core exchange that is slightly higher than the market price of your asset. As a consequence, paying fees in BTS should always be cheaper.

Market Fees

What happens if I enable Market fees?

If *Market Fees* of a UIA are turned on, fees have to be payed for each **market transaction**. This means, that market fees only apply to **filled orders**!

The percentage of market fees that are applied can be defined and changed by the issuer and any fee generated that way will be accumulated for each asset only to be claimed by the issuer.

If the Market Fee is set to 1%, the issuer will earn 1% of market volume as profit. These profits are accumulated for each UIA and can be withdrawn by the issuer.

How to claim accumulated fees?

Open up the issuer's account, click the assets tab and open up the dialog to change the asset. There will be a fee pool tab that allows you to fund the fee pool and claim the accumulated fees!

What if two different market fees are involved in a trade?

Suppose, I set the market fee for MyUIA market at 0.1%. and the market fee for YourUIA market at 0.3%.

In BitShares, You pay the fee upon **receiving an asset**. Hence, one side will pay 0.3% the other will pay 0.1%.

What are Asset Flags and Permissions?

When an asset is created, the issuer can set any combination of flags/permissions. **Flags** are set in stone unless there is **permission** to edit. Once a permission to edit is revoked, flags are permanent, and can never be modified again.

What are the Permissions:

- Enable market fee
- Require holders to be white-listed
- Issuer may transfer asset back to himself
- Issuer must approve all transfers
- Disable confidential transactions

What are the Flags?

- `charge_market_fee`: an issuer-specified percentage of all market trades in this asset is paid to the issuer
- `white_list`: accounts must be white-listed in order to hold this asset
- `override_authority`: issuer may transfer asset back to himself
- `transfer_restricted`: require the issuer to be one party to every transfer
- `disable_force_settle`: disable force settling
- `global_settle`: (only for bitassets) allows bitasset issuer to force a global settling - this may be set in permissions, but should not be set as flag unless, for instance, a prediction market has to be resolved. If this flag has been enabled, no further shares can be borrowed!
- `disable_confidential`: allow the asset to be used with confidential transactions
- `witness_fed_asset`: allow the asset to be fed by witnesses
- `committee_fed_asset`: allow the asset to be fed by the committee

Market Pegged Assets

Can I use the same flags/permissions as for UIAs?

Yes!

What are market-pegged-asset-specific parameters

- `feed_lifetime_sec`: The lifetime of a feed. After this time (in seconds) a feed is no longer considered *valid*.
- `minimum_feeds`: The number of feeds required for a market to become (and stay) active.
- `force_settlement_delay_sec`: The delay between requesting a settlement and actual execution of settlement (in seconds)
- `force_settlement_offset_percent`: A percentage offset from the price feed for settlement ($100\% = 10000$)

- `maximum_force_settlement_volume`: Maximum percentage of the supply that can be settled per day ($100\% = 10000$)
- `short_backing_asset`: The asset that has to be used to *back* this asset (when borrowing)

Tutorials

Creating a new UIA

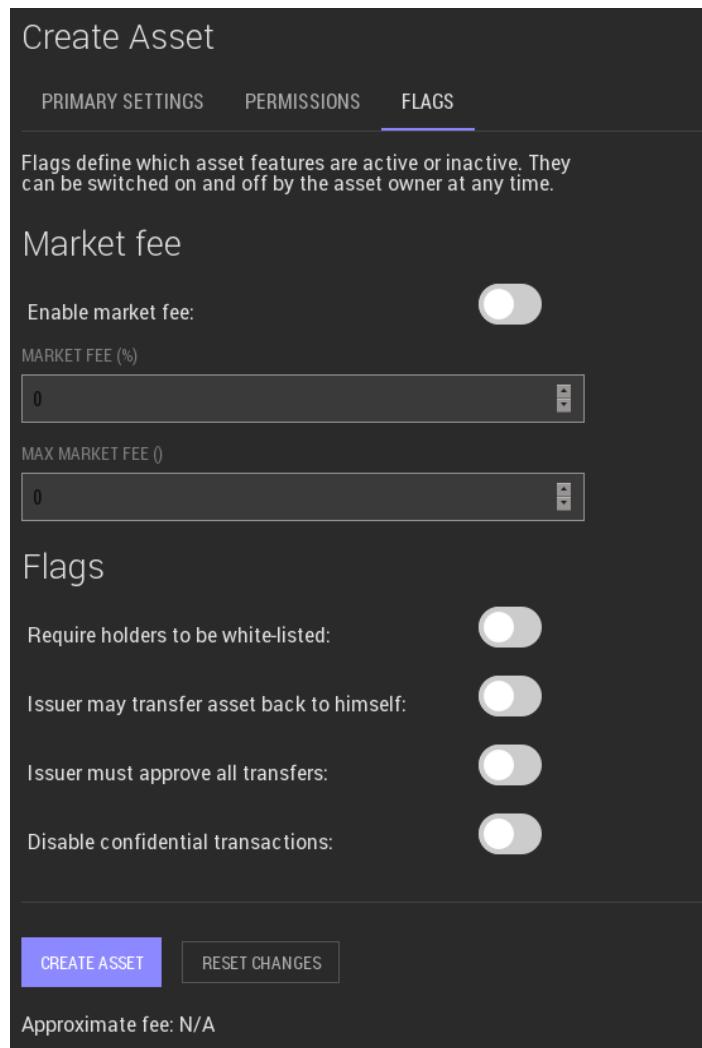
In order to create a new asset, we first need to enter our account's asset page and click *CREATE ASSET*:

The screenshot shows the 'Issued Assets' section of the Graphene UIA. On the left, there is a sidebar with a profile picture, the handle '#95837', and buttons for 'FOLLOW' and 'PAY'. Below that are 'Overview' and 'Assets' buttons. The main area is titled 'Issued Assets' and contains a table with columns: SYMBOL, DESCRIPTION, SUPPLY, MAXIMUM SUPPLY, ISSUE ASSET, and UPDATE ASSET. A button labeled 'CREATE ASSET' is located at the bottom of this table, and it is circled in orange.

We will enter the asset creation page that will allow us to define the assets parameters.

Primary Settings

The most important settings are listed in the primary settings.



The **Symbol** defined here will be reserved in the system for your assets. Once the asset is created, the symbol cannot be changed again!

Note: Symbols with less than **5** characters are very expensive. Please consult the Networks fees in the explorer!

The **maximum supply** is also a permanent setting and denotes the maximum amount of shares that can ever exist at the same time.

The *precision* is used to denote the number of decimal places. A *0* will result in an asset that cannot be separated below integer amounts (e.g. 1, 2, ..)

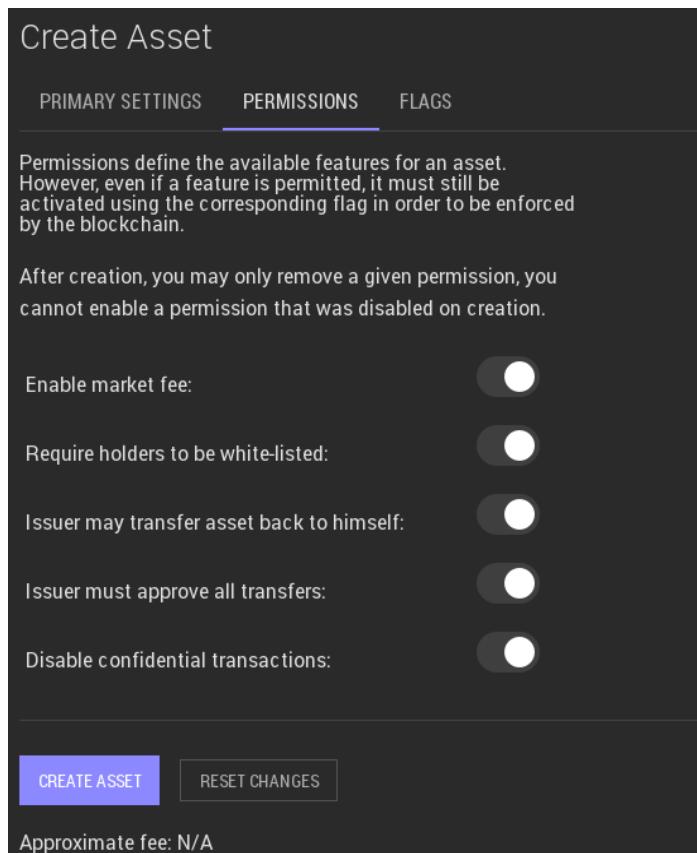
To allow transaction fees to be paid in the native asset, a core exchange rate is required at which a customer can implicitly trade the UIA into BTS from the asset's *fee pool*. This also requires that the fee pool is funded (e.g. by the issuer). Since all prices in BitShares are internally represented as *fractions* (i.e. a/b), we need to define a ratio between quote (the UIA) and base (BTS), i.e. the numerator and denominator for *price* = a/b .

Finally, a **description** can be used to let everyone know the purpose of the asset, or an internet address for further information.

Permissions (optional)

Even though the default settings should be fine for most UIAs, we have the option to **opt-out** of some available features. (By default, or permissions are *enabled*).

Note: Once a permission has been set to *false*, the permission cannot be reactivated!



We have the options to opt-out of:

- Enabling Market Fees
- Requiring holders to be white-listed
- Allow Issuer to withdraw from any account
- Require all transfers to be approved by the issuer
- Allow to disable confidential transactions

Note that setting these permissions does not imply that the features are enabled. To do so, we would also require to enable the corresponding flag(s). (See below)

Flags and Market Fees (optional)

The flags are used to *actually enable* a particular feature, such as market fees or confidential transfers.

Create Asset

PRIMARY SETTINGS PERMISSIONS FLAGS

Primary settings

SYMBOL

MAXIMUM SUPPLY

0

PRECISION

4

Core exchange rate

QUOTE ASSET AMOUNT

1

BASE ASSET AMOUNT

1

BTS

PRICE: 10 /BTS

Description

CREATE ASSET

RESET CHANGES

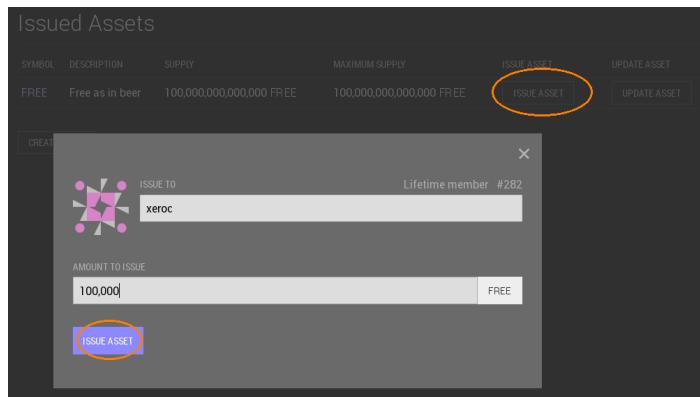
Approximate fee: N/A

If we have set the permission to have a market fee, we can enable the market fees here and set a percentage and max. fee.

We here also can enable the requirements for users to be white-listed, enable confidential transfers and give the issuer the power to withdraw its asset from customer accounts.

Issuing Shares

After creating the asset, no shares will exist until the issuer *issues* them:



The asset creation fee

The asset creation fee depends on the length of your symbol. 3 Character Symbols are the shortest and are rather expensive while symbols with 5 or more characters are significantly cheaper.

50% of the asset creation fee are used to pre-fill the assets fee pool. From the other 50%, 20% go to the network and 80% go to the referral program. This means, that if you are a life-time member, you get back 40% of the asset creation fee after the vesting period (currently 90 days).

Creating a UIA manually

Creating an Asset

Of course a UIA can also be created *manually* by means of the [CLI Wallet](#) command:

```
>>> create_asset <issuer> <symbol> <precision> <options> {} false
```

Note: A *false* at the end allows to check and verify the constructed transaction and does **not** broadcast it. The empty {} could be used to construct a [Market Pegged Assets](#) and is subject of another tutorial.

Parameters

The *precision* can any positive integer starting from 0. As *options* we pass a JSON object that can contain these settings:

```
{
    "max_supply" : 10000,      # Integer in satoshi! (100 for precision 1 and max 10)
    "market_fee_percent" : 0.3,
    "max_market_fee" : 1000, # in satoshi
    "issuer_permissions" : <permissions>,
    "flags" : <flags>,
    "core_exchange_rate" : {
        "base": {
            "amount": 21,           # denominator
            "asset_id": "1.3.0"     # BTS
        },
        "quote": {

```

```

        "amount": 76399,           # numerator
        "asset_id": "1.3.1"       # !THIS! asset
    }
},
"whitelist_authorities" : [],
"blacklist_authorities" : [],
"whitelist_markets" : [],
"blacklist_markets" : [],
"description" : "My fancy description"
}

```

The flags are construction as an JSON object containing these flags/permissions (see [Assets FAQ](#)):

```
{
    "charge_market_fee" : true,
    "white_list" : true,
    "override_authority" : true,
    "transfer_restricted" : true,
    "disable_force_settle" : true,
    "global_settle" : true,
    "disable_confidential" : true,
    "witness_fed_asset" : true,
    "committee_fed_asset" : true
}
```

Permissions and flags are modelled as sum of binary flags (see example below)

White-listing is described in more detail in [Asset User Whitelists](#).

Issuing Shares

After creation of the asset, no shares will be in existence until they are issued by the issuer:

```
issue_asset <account> <amount> <symbol> <memo> True
```

Python Example

```

from grapheneapi import GrapheneClient
import json

perm = {}
perm["charge_market_fee"] = 0x01
perm["white_list"] = 0x02
perm["override_authority"] = 0x04
perm["transfer_restricted"] = 0x08
perm["disable_force_settle"] = 0x10
perm["global_settle"] = 0x20
perm["disable_confidential"] = 0x40
perm["witness_fed_asset"] = 0x80
perm["committee_fed_asset"] = 0x100

class Config():
    wallet_host          = "localhost"
    wallet_port          = 8092

```

```
wallet_user          = ""
wallet_password      = ""

if __name__ == '__main__':
    graphene = GrapheneClient(Config)

    permissions = {"charge_market_fee" : True,
                   "white_list" : True,
                   "override_authority" : True,
                   "transfer_restricted" : True,
                   "disable_force_settle" : True,
                   "global_settle" : True,
                   "disable_confidential" : True,
                   "witness_fed_asset" : True,
                   "committee_fed_asset" : True,
                   }
    flags = {"charge_market_fee" : False,
              "white_list" : False,
              "override_authority" : False,
              "transfer_restricted" : False,
              "disable_force_settle" : False,
              "global_settle" : False,
              "disable_confidential" : False,
              "witness_fed_asset" : False,
              "committee_fed_asset" : False,
              }
    permissions_int = 0
    for p in permissions :
        if permissions[p] :
            permissions_int += perm[p]
    flags_int = 0
    for p in permissions :
        if flags[p] :
            flags_int += perm[p]
    options = {"max_supply" : 10000,
               "market_fee_percent" : 0,
               "max_market_fee" : 0,
               "issuer_permissions" : permissions_int,
               "flags" : flags_int,
               "core_exchange_rate" : {
                   "base": {
                       "amount": 10,
                       "asset_id": "1.3.0"},
                   "quote": {
                       "amount": 10,
                       "asset_id": "1.3.1"}},
               "whitelist_authorities" : [],
               "blacklist_authorities" : [],
               "whitelist_markets" : [],
               "blacklist_markets" : [],
               "description" : "My fancy description"
               }

    tx = graphene.rpc.create_asset("nathan", "SYMBOL", 3, options, {}, True)
    print(json.dumps(tx, indent=4))
```

Creating a MPA manually

We can create a MPA manually by means of the [CLI Wallet](#) command:

```
>>> create_asset <issuer> <symbol> <precision> <options> <mpaoptions> false
```

Note: A *false* at the end allows to check and verify the constructed transaction and does **not** broadcast it. The main difference between create a UIA and a MPA is <mpaoptions>!

All options (except for mpaoptions) are similar to creating a UIA as described in a separate tutorial ([Creating a UIA manually](#)).

MPA-specific settings

In order to create a MPA, we need to define some MPA-specific parameters:

```
{
    "feed_lifetime_sec" : 60 * 60 * 24,
    "minimum_feeds" : 7,
    "force_settlement_delay_sec" : 60 * 60 * 24,
    "force_settlement_offset_percent" : 1 * GRAPHENE_1_PERCENT,
    "maximum_force_settlement_volume" : 20 * GRAPHENE_1_PERCENT,
    "short_backing_asset" : "1.3.0",
}
```

See a detailed explanation of the parameters in [Assets FAQ](#).

Python Example

```
from grapheneapi import GrapheneClient
import json

perm = {}
perm["charge_market_fee"] = 0x01
perm["white_list"] = 0x02
perm["override_authority"] = 0x04
perm["transfer_restricted"] = 0x08
perm["disable_force_settle"] = 0x10
perm["global_settle"] = 0x20
perm["disable_confidential"] = 0x40
perm["witness_fed_asset"] = 0x80
perm["committee_fed_asset"] = 0x100
GRAPHENE_100_PERCENT = 10000
GRAPHENE_1_PERCENT = GRAPHENE_100_PERCENT / 100

class Config():
    wallet_host = "localhost"
    wallet_port = 8092
    wallet_user = ""
    wallet_password = ""

if __name__ == '__main__':
```

```
graphene = GrapheneClient(Config)

permissions = {"charge_market_fee" : True,
               "white_list" : True,
               "override_authority" : True,
               "transfer_restricted" : True,
               "disable_force_settle" : True,
               "global_settle" : True,
               "disable_confidential" : True,
               "witness_fed_asset" : True,
               "committee_fed_asset" : True,
               }
flags      = {"charge_market_fee" : False,
               "white_list" : False,
               "override_authority" : False,
               "transfer_restricted" : False,
               "disable_force_settle" : False,
               "global_settle" : False,
               "disable_confidential" : False,
               "witness_fed_asset" : False,
               "committee_fed_asset" : False,
               }
permissions_int = 0
for p in permissions :
    if permissions[p]:
        permissions_int += perm[p]
flags_int = 0
for p in permissions :
    if flags[p]:
        flags_int += perm[p]
options = {"max_supply" : 10000,
            "market_fee_percent" : 0,
            "max_market_fee" : 0,
            "issuer_permissions" : permissions_int,
            "flags" : flags_int,
            "core_exchange_rate" : {
                "base": {
                    "amount": 10,
                    "asset_id": "1.3.0"},
                "quote": {
                    "amount": 10,
                    "asset_id": "1.3.1"}},
            "whitelistAuthorities" : [],
            "blacklistAuthorities" : [],
            "whitelistMarkets" : [],
            "blacklistMarkets" : [],
            "description" : "My fancy description"
            }
mpaoptions = {"feed_lifetime_sec" : 60 * 60 * 24,
              "minimum_feeds" : 7,
              "force_settlement_delay_sec" : 60 * 60 * 24,
              "force_settlement_offset_percent" : 1 * GRAPHENE_1_PERCENT,
              "maximum_force_settlement_volume" : 20 * GRAPHENE_1_PERCENT,
              "short_backing_asset" : "1.3.0",
              }

tx = graphene.rpc.create_asset("nathan", "BITSYMBOL", 3, options, mpaoptions, ↵True)
```

```
print(json.dumps(tx, indent=4))
```

Creating a Prediction Market

Settings

In order to create a PM, we will need to set a particular parameter when creating the asset. This parameter can not be changed after creation of the asset.

Further, A PM-asset should have the following **flags** (not permissions):

```
{
    "disable_force_settle" : true,
    "global_settle" : false,
    "witness_fed_asset" : false,
    "committee_fed_asset" : false
}
```

and these MPA-options:

```
{
    "feed_lifetime_sec" : 60 * 60 * 24 * 356,
    "minimum_feeds" : 1,
    "force_settlement_delay_sec" : 60
    "force_settlement_offset_percent" : 0
    "maximum_force_settlement_volume" : 100 * GRAPHENE_1_PERCENT,
    "short_backing_asset" : "1.3.0",
}
```

Note: Unfortunately, `create_asset` cannot create prediction markets. Thus, we need to construct our `asset_create_operation` manually (see below)

Note: The precision of the prediction market asset has to be identical with the short backing asset's precision

Settlement Authorities

The issue can choose between three parties that are allowed to settle the prediction market:

- Committee
- Witnesses
- Other accounts

Committee

If only the committee is supposed to be able to settle the market, you need to set the options to::

```
{  
    "witness_fed_asset" : false,  
    "committee_fed_asset" : true  
}
```

Witnesses

If only the witnesses are supposed to be able to settle the market, you need to set the options to::

```
{  
    "witness_fed_asset" : true,  
    "committee_fed_asset" : false  
}
```

Note: The idea here is that the median of all price feeds published by the witnesses indicates a positive or negative resolution of the prediction market.

Other Accounts

Similar to *Privatized BitAssets*, the feed can be also published by a arbitrary set of accounts. It is important to understand that in order to settle a prediction market, only **one price feed** is required. Hence, anyone in the list of allowed settlers can settle the market and no consensus needs to be reached. Alternatively, if you want to settle a market only if several accounts can reach a consensus, a new resolution account can be created that uses *hierarchical multi-signature* similar to the *committee-account*.

The list of settlement price producers can be defined with:

```
>>> update_asset_feed_producers <symbol> ["account-a", "account-b"] true
```

Python Example

```
from grapheneapi import GrapheneClient  
import json  
  
perm = {}  
perm["charge_market_fee"] = 0x01  
perm["white_list"] = 0x02  
perm["override_authority"] = 0x04  
perm["transfer_restricted"] = 0x08  
perm["disable_force_settle"] = 0x10  
perm["global_settle"] = 0x20  
perm["disable_confidential"] = 0x40  
perm["witness_fed_asset"] = 0x80  
perm["committee_fed_asset"] = 0x100  
GRAPHENE_100_PERCENT = 10000  
GRAPHENE_1_PERCENT = GRAPHENE_100_PERCENT / 100  
  
class Config():  
    wallet_host = "localhost"
```

```

wallet_port          = 8092
wallet_user          = ""
wallet_password      = ""

if __name__ == '__main__':
    graphene = GrapheneClient(Config)

    issuer = "nathan"
    symbol = "PMMP"
    backing = "1.3.0"

    account = graphene.rpc.get_account(issuer)
    asset = graphene.rpc.get_asset(backing)

    permissions = {"charge_market_fee" : True,
                   "white_list" : True,
                   "override_authority" : True,
                   "transfer_restricted" : True,
                   "disable_force_settle" : True,
                   "global_settle" : True,
                   "disable_confidential" : True,
                   "witness_fed_asset" : True,
                   "committee_fed_asset" : True,
                   }
    flags = {"charge_market_fee" : False,
             "white_list" : False,
             "override_authority" : False,
             "transfer_restricted" : False,
             "disable_force_settle" : True,
             "global_settle" : False,
             "disable_confidential" : False,
             "witness_fed_asset" : False,
             "committee_fed_asset" : False,
             }
    permissions_int = 0
    for p in permissions:
        if permissions[p]:
            permissions_int += perm[p]
    flags_int = 0
    for p in permissions:
        if flags[p]:
            flags_int += perm[p]
    options = {"max_supply" : 10000000000,
               "market_fee_percent" : 0,
               "max_market_fee" : 0,
               "issuer_permissions" : permissions_int,
               "flags" : flags_int,
               "core_exchange_rate" : {
                   "base": {
                       "amount": 10,
                       "asset_id": asset["id"]},
                   "quote": {
                       "amount": 10,
                       "asset_id": "1.3.1"}},
               "whitelist_authorities" : [],
               "blacklist_authorities" : [],
               "whitelist_markets" : [],
               "blacklist_markets" : [],
               }

```

```

        "description" : "Prediction Market"
    }
mpaoptions = {"feed_lifetime_sec" : 60 * 60 * 24 * 14,
              "minimum_feeds" : 1,
              "force_settlement_delay_sec" : 10,
              "force_settlement_offset_percent" : 0 * GRAPHENE_1_PERCENT,
              "maximum_force_settlement_volume" : 100 * GRAPHENE_1_PERCENT,
              "short_backing_asset" : asset["id"],
            }

op = graphene.rpc.get_prototype_operation("asset_create_operation")
op[1]["issuer"] = account["id"]
op[1]["symbol"] = symbol
op[1]["precision"] = asset["precision"]
op[1]["common_options"] = options
op[1]["bitasset_opts"] = mpaoptions

""" This flag will declare the asset as a prediction market
asset!
"""
op[1]["is_prediction_market"] = True

handle = graphene.rpc.begin_builder_transaction()
graphene.rpc.add_operation_to_builder_transaction(handle, op)
graphene.rpc.set_fees_on_builder_transaction(handle, "1.3.0")
tx = graphene.rpc.sign_builder_transaction(handle, True)
print(json.dumps(tx, indent=4))

```

Closing/Settling a Prediction Market

All the issuer needs to do is publish a valid price feed for the asset. The *global_settle* option will be set automatically and borrow positions can settle at the price feed.

Python Script

```

from grapheneapi import GrapheneClient
import json

class Config():
    wallet_host      = "localhost"
    wallet_port      = 8092
    wallet_user      = ""
    wallet_password   = ""

if __name__ == '__main__':
    graphene = GrapheneClient(Config)
    symbol = "PM"
    issuer = "nathan"
    producer = "nathan"
    pm_result = True # or False           <<<----- Result goes here
    account = graphene.rpc.get_account(issuer)
    asset = graphene.rpc.get_asset(symbol)
    # Publish a price

```

```

settle_price = {"quote": {"asset_id": "1.3.0",
                         "amount": 1 if pm_result else 0},
                "base": {"asset_id": asset["id"],
                         "amount": 1
                     } }
handle = graphene.rpc.begin_builder_transaction()
tx = graphene.rpc.global_settle_asset(symbol, settle_price, True)
print(json.dumps(tx, indent=4))

```

User Issued Assets

BitShares allows individuals and companies to create and issue their own tokens for anything they can imagine. The potential use cases for so called user-issued assets (UIA) are innumerable. On the one hand, UIAs can be used as simple event tickets deposited on the customers mobile phone to pass the entrance of a concert. On the other hand, they can be used for crowd funding, ownership tracking or even to sell equity of a company in form of stock.

All you need to do is click in order to create a new UIA is a few mouse clicks, define your preferred parameters for your coin, such as supply, precision, symbol, description and see your coin's birth after only a few seconds. From that point on, you can issue some of your coins to whomever you want, sell them and see them instantly **traded against any other existing coin** on BitShares.

Unless you want some restriction. As the issuer, you have certain privileges over your coin, for instance, you can allow trading only in certain market pairs and define who actually is allowed to hold your coin by using white- and blacklists. Of course, an issuer can opt-out of his privileges indefinitely for the sake of trust and reputation.

As the owner of that coin, you don't need to take care of all the technical details of blockchain technology, such as distributed consensus algorithms, blockchain development or integration. You don't even need to run any mining equipment or servers, at all.

So what's the drawback?

There is a drawback in this scenario, namely, a centralized issuance of new tokens. To some extend, this could be managed by a hierarchical multi-signature issuer account that prevents any single entity from issuing new coins but instead requires a consensus among an arbitrary set of people to agree on any changes to the coin.

Obviously, the regulations that apply to each kind of token vary widely and are often different in every jurisdiction. Hence, BitShares comes with tools that allow issuers to remain compliant with all applicable regulations when issuing assets assuming regulators allow such assets in the first place.

Use Cases

- Reward Points
- Fan Credits
- Flight Miles
- Event Tickets
- Digital Property
- Crowd-Funding
- Company Shares

Frequently Asked Questions

Tutorials

Market Pegged Assets

A crypto-currency, with the properties and advantages of Bitcoin, that is capable of maintaining price parity with a globally adopted currency (e.g. U.S. dollar), has high utility for convenient and censorship resistant commerce. This can be achieved by BitShares' market pegged assets (MPA), a new type of freely traded digital asset whose value is meant to track the value of a conventional underlying asset by means of an over-collateralized, counterparty risk-free, smart-contract secured blockchain loan.

Instead of creating a UIA where the full control over supply is in the hands of the issuer, we can also create a **Market Pegged Asset** (MPA) and let the market deal with demand and supply. All we need is a *fair price* and another asset that can be used as collateral.

Why would we need *collateral* for? Since the issuer of a MPA has no control over the supply, the blockchain protocol deals with increasing and decreasing supply. In order for a user to get some of the new coins, he will need to put collateral into a **smart contract** (technically, this contract is a *collateralized loan*).

A simple example would be a MPA that is backed by USD (a stable crypto token within BitShares) that requires a collateral ratio of 200%. Then, in order to get new coin, we can borrow 100 USD worth of new coins by paying 200 USD.

By this, the supply of your coin is increased by 100. But how would it be decreased? The USD are locked in the smart contract and can only be reclaimed if the debt (here, 100 coins) are returned. Returning them will result in the coins being removed from the supply because they are no longer backed by any collateral.

So what do we need a *fair price*? Remember that we chose a collateral ratio of 200%? That number tells us how well *backed* your coins are by the collateral. But what would happen if the value of your coin goes to the moon? Then your collateral ratio will reduce to say 150%. At a certain percentage, the blockchain will automatically trigger so called *Margin Calls* which will

1. Take your collateral (here, USD)
2. Sell it in the market to buy back the coin you owe
3. Close the contract
4. Pay your the residual USD

A *fair price* thus tells the market what your coin is worth (e.g. traded for on external exchanges) and triggers margin calls if necessary.

But there is more! Everyone that holds your (MPA) coin in BitShares can convert the coin into the backing asset at a fair price. This procedure is called “settlement” and ensures that your MPA is always worth **at least** the *fair price*.

In the User Interface, MPAs are easily distinguishable from UIAs in the asset explorer.

SmartCoins

BitAssets can be created and owned by anyone on the network. However, those that are owned by the BitShares Committee, are called *SmartCoins*. Among these are:

- (Bit)USD
- (Bit)CNY
- (Bit)EUR

- (Bit)GOLD
- (Bit)Silver

Balances in these assets are labeled with *USD*, *CNY*, etc., because represent the same value as their underly.

Collateralized Tokens

A *SmartCoin* (synonym for MPA) is a crypto-currency that *always* has 100% or more of its value backed by the BitShares core currency (BTS), to which they can be converted at any time, as *collateral* in a collateralized loan.

What makes MPAs unique is that they are free from counterparty risk even though they resemble a collateralized loan. This is achievable by letting the network itself (implemented as a software protocol) be responsible for securing the collateral and performing settlements as will be described in more detail below.

Market Mechanics

Each BitAsset has a feed that is provided by the witnesses that indicate a fair price for that asset. This so called *Settlement Price* or *Feed Price* is used to margin call positions that borrowed BitAssets and can no longer maintain the minimum collateral ratio (i.e. maintenance collateral ratio). The collateral of these positions is used to buy back the debt from the market automatically and the position will be closed. By these rules, the network enforces the exchange participants to always maintain a collateral that is higher than the minimum requirement. Currently, the minimum required collateral ratio is **175%** and can be changed by the witnesses.

Read more about the *margin call mechanics* before trading.

Frequently Asked Questions

Tutorials

Fee Backed Asset

Existing core features of the BitShares protocol are Market Pegged Assets (MPA) and issuer backed User Issued Assets (UIA). In this proposal, we introduce another type of asset: *Fee Backed Assets (FBA)*.

Feed backed assets allow to propose and fund *market based* innovation by sharing a cut of future profits generated by this particular innovation with the people that helped fund it. Think of it as a *Kickstarter* for features. Hence, if people can profit from successful features in the form of fees then it can help the BitShares ecosystem to become more adaptable over time as it promotes innovation and can pay for its development.

If you have any features in mind that require new kind of transaction on the blockchain, you can code that feature and fund it with an FBA.

Feed Backed Assets have been proposed in [BSIP-0007](#).

Privatized BitAssets

Alternatively to regular MPA like the bitUSD, BitShares also offers entrepreneurs an opportunity to create their own SmartCoins with custom parameters and a distinct set of price feed producers.

Privatized SmartCoin managers can experiment with different parameters such as collateral requirements, price feeds, force settlement delays and forced settlement fees. They also earn the trading fees from transactions the issued asset is involved in, and therefore have a financial incentive to market and promote it on the network. The entrepreneur

who can discover and market the best set of parameters can earn a significant profit. The set of parameters that can be tweaked by entrepreneurs is broad enough that SmartCoins can be used to implement a fully functional prediction market with a guaranteed global settlement at a fair price, and no forced settlement before the resolution date.

Some entrepreneurs may want to experiment with SmartCoins that always trade at exactly \$1.00 rather than strictly more than \$1.00. They can do this by manipulating the forced settlement fee continuously such that the average trading price stays at about \$1.00. By default, BitShares prefers fees set by the market, and thus opts to let the price float above \$1.00, rather than fixing the price by directly manipulating the forced settlement fee.

Preparations

First, the reader should familiarize himself with the following articles:

- [Assets FAQ](#)
- [Creating a UIA manually](#)
- [Creating a MPA manually](#)

Parameters

The relevant and interesting parameters are located in the uia flags:

```
{  
    "witness_fed_asset" : false,  
    "committee_fed_asset" : false  
}
```

Setting these two parameters to `false`, allows to manually define the set of feed producers (see below). Alternatively, setting either of both to `true` will give the corresponding entity the responsibility to produce and publish a feed.

Changing the Feed producers

The following command replaces the set of currently allowed feed producers by the new set of feed producers:

```
update_asset_feed_producers <symbol> ["prod-a", "prod-b"] True
```

Producing a Feed

We have a distinct tutorial that describes how feed are can be published: [Publishing a Feed](#).

Prediction Markets

A prediction market is a specialized BitAsset such that total debt and total collateral are always equal amounts (although asset IDs differ). No margin calls or force settlements may be performed on a prediction market asset. A prediction market is globally settled by the issuer after the event being predicted resolves, thus a prediction market must always have the `global_settle` permission enabled. The maximum price for global settlement or short sale of a prediction market asset is 1-to-1.

Note: In the following, we denote a *positive outcome* as a predication market that resolves to *true* (i.e. a price feed of *1*) and a *negative outcome* to resolve to *false* (i.e. a price feed of *0*)

If the bet resolves to *true* (i.e. a price feed of *1*), then the PM-asset can be settled release the collateral to the holder of the asset.

If, instead, the bet resolves to *false* (i.e. a price feed of *0*), then those that sold the PM-asset on the market and went short, made a profit since it PM-asset became worthless.

Creation

Prediction markets are assets that trade freely and can be borrowed from the market at a 1:1 ratio with the backing asset (which could be any other asset, including BTS, USD, GOLD).

The technical details are described in a separate tutorial:

- *Creating a Prediction Market*

Betting

A user can take either bet on a positive outcome, or a negative outcome. We here show how this works, technically.

Betting for a Positive Outcome

If you are confident that the bet will resolve positive, you want to **hold** that particular PM-asset since it allows you to settle it for its collateral on a 1:1 basis.

In order to get hold of those tokens, you can put a buy order for them at any price (between 0 and 1) and wait for it to be filled, or buy at market rates. By this technique, a user can pre define at which odds to buy shares.

For instance, if you think that the bet resolves positively at a probability of 80%, you can put your buy order at a price of 0.8. If the bet resolves positively (price feed of *1*), then you can settle your shares at *1* and make a 20% profit.

If you can buy tokens at a price of 0.2 (i.e. market participants think it is unlikely to resolve positively), then you could make 80% profits at a risk of loosing with 80% probability.

After closing of the bet, a user can claim his profits by **settling** his borrow position and taking out the collateral:

- **Settlement in the CLI wallet:**

```
>>> settle_asset <account> <amount> <symbol> True
```

- **Borrowing in the GUI wallet:** A settlement button is available when hovering the asset in your account's overview.

Betting for a Negative Outcome

In order to bet for a negative outcome (bet resolves to *false* with a price feed of *0*), you need to **sell** the tokens. In order to get them, you should **not** buy them at the market, but instead **borrow** them from the network by paying collateral at a 1:1 ratio.

For example, in the *PM.PRESIDENT2016* if you want to bet on a negative outcome with *100k BTS*, you can borrow *100k PM.PRESIDENT2016* by paying *100k BTS* to the network.

Note: Since PM-Assets can technically be pegged by any other asset, you may need to pay USD (or anything else) instead of BTS.

Once you borrowed the token, you can sell them at any price between 0 and 1. If you think the probability of a negative outcome is 20%, you should consider selling your tokens at 0.2.

If the bet resolves negatively (price feed of 0), your debts is worth $debt = amount * price = 0 \text{ BTS}$, you can reclaim your collateral at zero cost, and get to keep 20% profits from selling the token at 0.2. If instead the bet resolves positively and you sold all tokens, you cannot close your borrow position to redeem your collateral. However, your total loss is reduced by 20% for selling the tokens at the market.

If, by the end of the bet, you still have some of the tokens left, you can of course close your borrow position partly and redeem the corresponding percentage of the collateral.

- **Borrowing in the CLI wallet:**

```
>>> borrow_asset <account> <amount> <PMsymbol> <1:1-amount> true
```

- **Borrowing in the GUI wallet:** Of course, the asset can also be borrowed in the **GUI/web wallet** by using the *Borrow x* button in the market.



Resolving

A price feed needs to be published for the prediction market by the issuer or feed producer. It is essentially a global settlement which will set the parameters of the asset such that the holders of the asset can settle at the outcome of the bet (0, or 1). The details are shown in the tutorial:

- [Closing/Settling a Prediction Market](#)

Decentralized Exchange

The decentralized exchange (further denoted simply as *the DEX*) allows for direct exchange of digital goods traded in the BitShares ecosystem.

A decentralized exchange has a very particular set of advantages over traditional centralized exchanges and we would like to address some of them briefly below. Although the BitShares DEX comes with all of them, it is up to the reader and customer to leverage those features in full or only partially.

- **Separation of Powers:** There is no reason why the same entity needs to be responsible for issuing IOUs and for processing the order book. In BitShares, order matching is performed by the protocol, which is unaware of implications concerning the involved assets.
- **Global Unified Order Book:** Since BitShares is global, anybody with an internet access can use the DEX for trading. This brings the world's liquidity to a single order book for decentralized trading.
- **Trade Almost Anything:** The BitShares DEX is asset agnostic. Hence you can trade at **any** pair. While some pairs may end up with low liquidity, such as SILVER:GOLD, other pairs such as USD:EUR for FOREX trading will see huge volume.

- **No Limits:** The BitShares protocol is unable to limit your trading experience.
- **Decentralized:** The DEX is decentralized across the globe. This not only means that there is no single point of failure, but it also implies that the BitShares exchange is open for trading 24/7 because it's always daytime somewhere.
- **Secure:** Your funds and trades are secured with industry-grade elliptic curve cryptography. No one will be able to access your funds unless you let them. To further strengthen the security, we allow our customers to setup escrow and multi-signature schemes.
- **Fast:** In contrast to other decentralized networks, the BitShares DEX allows for real-time trading and is only limited by the speed of light and the size of the planet.
- **Provable Order Matching Algorithm:** What makes the BitShares DEX unique is the provable order matching algorithm. Given a set of orders, you will always be able to provably verify that these orders have been matched properly.
- **Collateralized Smartcoins:** One of the biggest features of BitShares are its *smartcoins* such as bitUSD, bitEUR, bitCNY, and others. For the sake of convenience, these assets are denoted simply as USD, EUR, CNY, etc. in the wallet. These digital tokens represent the same value as their underlying physical asset. Hence 1 USD in this wallet is worth \$1 and can be redeemed as such. Any of these tokens is backed by BitShares' company shares (BTS) being locked up as collateral and being available for settlement at its current price.

Trading

This page will give a very quick introduction on how to interpret the terms used by the DEX and how trading pairs are presented.

Pairs

In BitShares, almost any asset can be traded with all other assets. Once we have picked two assets, we usually refer to a *market pair*. For instance, we can trade USD against EUR in the USD:EUR pair.

For sake of consistency, we will use the generalized terms *base* and *quote* such that pairs are represented as:

```
quote : base
```

and for instance with *base* being USD and *quote* being EUR, denote the EUR:USD pair.

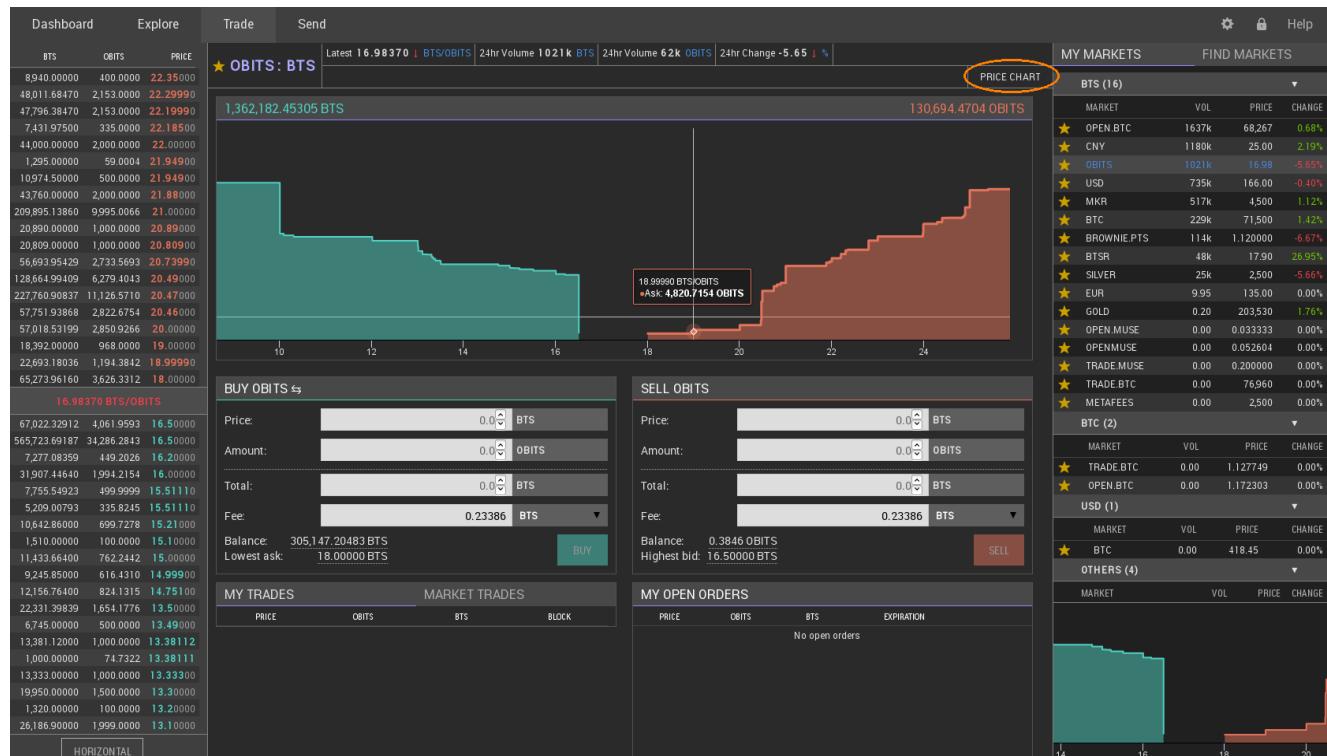
Market Overview

The market overview that can be accessed via the explorer, shows a set of predefined default markets. Note that the list of default markets may vary depending on the wallet provider. Further markets can be added using the *Find Markets* tab. Adding a *Star* to your favorite markets will make it appear in your list of default markets.

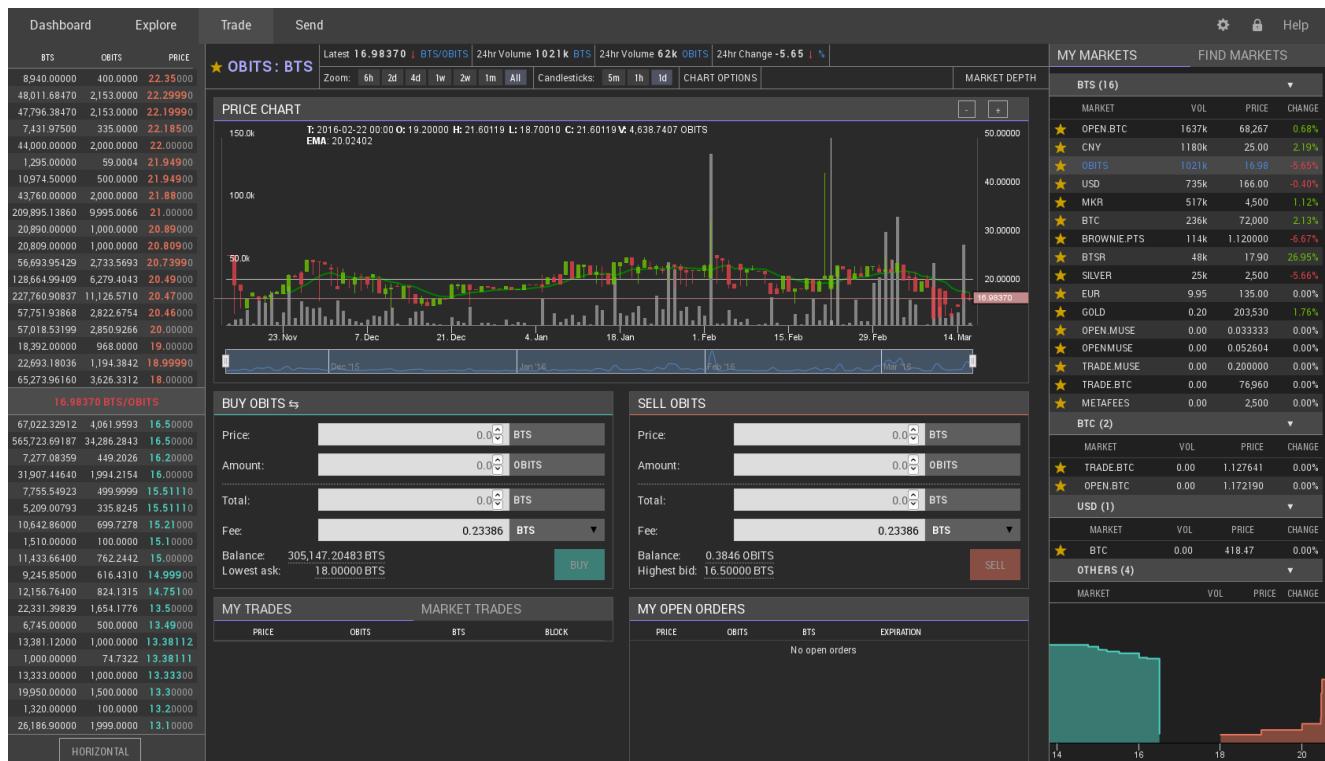
MY MARKETS						FIND MARKETS		
BTS (16)		QUOTE SUPPLY		VOL	PRICE	CHANGE		
★	OPEN.BTC	430 OPEN.BTC		1637k	68,267	0.68%		
★	CNY	607,366,279.3 CNY		1180k	25.00	2.19%		
★	OBITS	18,176,328 OBITS		1021k	16.98	-5.65%		
★	USD	107,986,508.4 USD		768k	168.00	0.80%		
★	MKR	50,000 MKR		517k	4,500	1.12%		
★	BTC	65,718,776.55 BTC		236k	72,000	2.13%		
★	BROWNIE.PTS	9,400,000 BROWNIE.PTS		114k	1,120,000	-6.67%		
★	BTSR	10,600,000 BTSR		48k	17.90	26.95%		
★	SILVER	844,817.2 SILVER		25k	2,500	-5.66%		
★	EUR	1,650 EUR		9.95	135.00	0.00%		
★	GOLD	8,745,979 GOLD		0.20	203,530	1.76%		
★	OPEN.MUSE	95,000,000 OPEN.MUSE		0.00	0.033333	0.00%		
★	OPENMUSE	95,000,000 OPENMUSE		0.00	0.052604	0.00%		
★	TRADE.MUSE	7,220,300 TRADE.MUSE		0.00	0.200000	0.00%		
★	TRADE.BTC	25,580,823.1 TRADE.BTC		0.00	76,960	0.00%		
★	METAFEEES	5,403,331,498 METAFEEES		0.00	2,500	0.00%		
BTC (2)								
		QUOTE SUPPLY		VOL	PRICE	CHANGE		
★	TRADE.BTC	25,580,823.1 TRADE.BTC		0.00	1,123,959	0.00%		
★	OPEN.BTC	430 OPEN.BTC		0.00	1,168,363	0.00%		
USD (1)								
		QUOTE SUPPLY		VOL	PRICE	CHANGE		
★	BTC	65,718,776.55 BTC		0.00	418.06	0.00%		
OTHERS (4)								
		QUOTE SUPPLY		VOL	PRICE	CHANGE		
★	MKR.OPEN.BTC	50,000 MKR		3.35	0.065000	-7.14%		
★	MKR.OPEN.BTH	50,000 MKR		0.00	1,550,000	0.00%		
★	METAFEEES.METAEX.BTC	5,403,331,498 METAFEEES		0.00	0.020000	0.00%		

Market

When entering a market, you will be presented with either the market depth or the price chart depending on your settings.



... or the price chart depending on your settings.



You can switch between your views by pressing the corresponding button as highlighted below.

Order Books

The order book consists of an *ask* and a *bid* side. Since trading pairs do not have a preferred orientation, and can be flipped, the following table shall give an overview of ask/bid and the corresponding buy/sell operations for each side:

Side	Sell	Buy
Ask	<i>quote</i>	<i>base</i>
Bid	<i>base</i>	<i>quote</i>

Obviously, what is on the bid side of the USD:EUR pair will be on the ask side on the EUR:USD pair. Of course prices are internally represented as fractions, and thus results in both pairs being identical.

Trading

To place a trading order, it is required to fill the form on either the *ask* or the *bid* side (respectively, *buy* or *sell* side). You will need to define a *price* and an *amount* to sell/buy. The cost for this order will be calculated automatically. Note that there will be an additional fee required to actually place the order.



Once the order is filled (i.e. someone sold/bought your offer), your account will be credited by the corresponding asset.

Unfilled orders can be canceled at any time.

Order Matching

BitShares 2.0 matches orders on a first-come, first-serve basis and gives the buyer the best price possible up to the limit (also known as “walking the book”). Rather than charging *unpredictable fees* from market overlap (as has been in the previous network), the network charges a defined fee based upon the size of the order matched and the assets involved. Each asset issuer gets an opportunity to configure their fees.

Tutorial

[./tutorials/dex-trading.rst](#)

Short Selling BitAssets

In order to increase your exposure to BTS and offer liquidity to BitAssets, such as USD, EUR, GOLD, etc., you can go *borrow* this bitAsset from the network and *sell it short*. We will here briefly describe the procedure.

Borrowing

The BitShares network is capable of issuing any amount of any BitAsset and lend it out to participants given enough collateral.

- *settlement price:* The price for 1 BTS as it is traded on external exchanges.

- *maintenance collateral ratio* (MCR): A ratio defined by the witnesses as minimum required collateral ratio
- *maximum short squeeze ratio* (MSQR): A ratio defined by the witnesses as to how far shorts are protected against short squeezes
- *short squeeze protection* (SQP): Defines the most that a margin position will ever be forced to pay to cover
- *call price* (CP): The price at which short/borrow positions are margin called

Margin Call

The BitShares network is capable of margin calling those positions that do not have enough collateral to back their borrowed bitAssets. A margin call will occur any time the highest bid is less than the *call price* and greater than *SQP*. The margin position will be forced to sell its collateral anytime the highest offer to buy the collateral is less than the call price (x/BTS):

```
SQP      = settlement price / MSQR
call price = DEBT / COLLATERAL * MCR
```

The margin call will take the collateral, buy shares of borrowed bitAsset at market rates up to the SQP and close the position. The remaining BTS of the collateral are returned to the customer.

Read more about the [margin call mechanics](#) before trading.

Settlement

Holders of any bitAsset can request a settlement at a *fair price* at any time. The settlement closes the borrow/short positions with lowest collateral ratio and sells the collateral for the settlement.

Note, that there is a maximum daily settlement volume (currently 2%) defined by the [committee](#) to prevent exploitation via external price movements.

Selling

After borrowing bitAssets, they can be sold free at any of the corresponding markets at any price a buyer is willing to pay. With this step, the short-selling is now complete and you are short that particular bitAsset.

Updating Collateral Ratio

At any time, the holder of a borrow/short position can modify the collateral ratio in order to flexibly adjust to market behavior. If the collateral ratio is increase, an additional amount of BTS is locked as collateral, while reducing the collateral ratio will require an amount of the corresponding BitAsset to be payed back to the network.

Covering

To close a borrow/short position, one must hold the borrowed amount of that particular bitAsset to hand it over to the BitShares network. After that, the BitAssets are reduced from the corresponding supply and the collateral is released and given back to its owner.

Discussion

Shorts can pick their place in line for settlement. Think of it this way, if you fall in the bottom 2% of shorters by collateral you have been given notice of potential margin call since only 2% can be settled, daily. This is like any other market where they give you 24 hours to add collateral. If someone is short and doesn't want to meet the new higher collateral limits then they can either cover on their own terms or add collateral.

By giving 24 hours shorts have an opportunity to cover prior to any price manipulation by big players.

If there is a 10% premium on BitUSD relative to the feed, then the attacker would have to increase reported price feed (value of BTS) by 10% just to get the force-settlement price to equal the previously fair value for BitUSD. They would have to push beyond 10% before the short starts taking a loss relative to a voluntary cover. All savvy market participants would be aware of a large force-settle order and would therefore reset the manipulator making it much harder to manipulate the price. In effect, price manipulation represents "free money" to those who know it is going on.

Look at it another way, someone enters a large force-settlement order it becomes an opportunity for the shorter to do reverse manipulation. It is a tug of war where both sides (short and long) have equal opportunity to manipulate the market in their favor. They go to battle and the result is just the fair market price at that point in time. It is not a guaranteed win for the potential manipulator.

Margin call mechanics

The mechanics of a margin call in Bitshares are currently poorly understood, so I'd like to try to clarify a little by using examples from the USD:BTS market. I think part of the current confusion lies in people talking about the same market but using different market directions, ie. USD:BTS or BTS:USD, so terms like above/below don't mean the same thing to different people. I will only use USD in these examples, but USD can be replaced by any bit asset in this context. I prefer to use the USD:BTS market direction, so these examples will have prices in BTS/USD.

What is a margin call?

A margin call is the market forcing you to sell your collateral in order to buy enough USD to close your position. In the USD:BTS market a margin call is equivalent to a bid: it is an order to buy USD for BTS.

A margin call will happen because the price has increased to the point where your collateral is insufficient with respect to the current collateral requirements of the Bitshares market rules. The required collateral is a tuneable parameter in Bitshares, set by the maintenance collateral ratio (MCR) which is maintained by the feed producers (ie. the witnesses).

How is the call price calculated?

As mentioned above the call price of a margin position depends on the MCR and the amount of debt and collateral in your position. It is independent of the price feed (settlement price). As an example, say you have opened the following position:

- Debt: 10 USD
- Collateral: 10000 BTS
- MCR is 1.75

The call price of your position is $10000 \text{ BTS} / (10 * 1.75 \text{ USD}) = 571.429 \text{ BTS/USD}$.

How is the collateral ratio (CR) calculated?

The collateral ratio depends on the feed price (settlement price). Taking a feed price of 300 BTS/USD and building on the above example with 10 USD debt and 10000 BTS collateral:

- CR: $(10000 \text{ BTS} / 300 \text{ BTS/USD}) / 10 \text{ USD} = 3.33$

Execution Conditions

When will a margin call happen?

This is where it gets complicated. Margin Call are only possible if the feed price is below your call price. A margin call will happen whenever the squeeze protection price goes above the call price of your position. To better understand how this works, let's go back to our margin position and look at collateral ratios:

Say we have the following:

- Debt: 10 USD
- Settlement price: 300 BTS/USD
- CR: 1
- Collateral is therefore 3000 BTS

This is also known as the Black Swan level, and we want to perform a margin call before the collateral ratio goes this low. This is why we have the Maintenance Collateral Ratio (MCR), to enforce a buffer zone before a position goes into Black Swan territory. So if we apply the MCR of 1.75 to this position:

- Debt: 10 USD
- Settlement Price: 300 BTS/USD
- CR: 1.75
- Collateral is therefore $3000 \text{ BTS} * 1.75 = 5250 \text{ BTS}$

This is much safer, there is a bit of margin for the position to be closed before going into Black Swan levels. Since in our example, the USD **requires** 1.75 ratio, the call price of this position is now exactly equal to the feed price of 300 BTS/USD.

- Call price: $5250 / (10 * 1.75) = 300 \text{ BTS/USD}$

The remaining question then is, at what point should we force the position to attempt to close itself? This is where the SQPR comes in. Let's look at two scenarios, SQPR of 1.1 and SQPR of 1.5:

** SQPR of 1.1 **

- Settlement price: 300 BTS/USD
- SQPR: 1.1
- Squeeze Protection Price (SQPP): 330 BTS/USD

In this case, any margin position that has a call price below 330 BTS/USD will be forced to settle, and therefore be added to the orderbook as an order to buy USD for BTS.

** SQPR of 1.5 **

- Settlement price: 300 BTS/USD
- SQPR: 1.5
- Squeeze Protection Price (SQPP): 450 BTS/USD

In this case, any margin position that has a call price below 450 BTS/USD will be forced to settle, and therefore be added to the orderbook as an order to buy USD for BTS.

Discussion

Another way of looking at this is by looking at the Collateral Ratio of the position. If we want to stay at or above the squeeze protection price, what is the required collateral ratio? Let's do the math:

- Settlement Price: 300 BTS/USD
- MCR: 1.75
- SQPR: 1.1
- Debt: 10 USD
- Call price: $CP = SQPP = 300 * 1.1 = 330$ BTS/USD
- Collateral = $(10 \text{ USD} * 1.75) * 330 \text{ BTS/USD} = 5775 \text{ BTS}$

The collateral ratio of this position is $(5775 \text{ BTS} / 300 \text{ BTS/USD}) / 10 \text{ USD} = 1.925$.

This is equivalent to the MCR

- SQPR: $1.75 * 1.1 = 1.925$.

In other words, in order to stay **safe** and not be margin called, the margin position must maintain a collateral ratio higher than $MCR * SQPR$.

- **Safe position:** $CR > MCR * SQPR$

At what price will the margin call execute?

This is the part I believe is most misunderstood, so I will use some screenshots of a fictional USD:BTS market to explain. We will use the following parameters:

- SQPR: 1.2
- MCR: 1.75
- SQPR * MCR: 2.1
- Settlement price: 300 BTS/USD
- Squeeze protection price: $300 * 1.2 = 360$ BTS/USD
- Debt: 10 USD
- Collateral: 5687.5
- CR: 1.896
- Call price: 325 BTS/USD

From what we've seen above, it's clear that this position should be margin called: it has a CR of 1.896 which is well below the safe ratio of 2.1.

It will therefore get added to the order book as a bid to buy USD like this:



The margin called order will buy any USD priced in the range 325–360 BTS/USD. The squeeze protection price acts as a price ceiling, meaning the forced margin order will not execute at a very high price in an illiquid market: it is protected from high prices by the SQPR.

Margin calls only execute in the range [Call Price - SQPP]

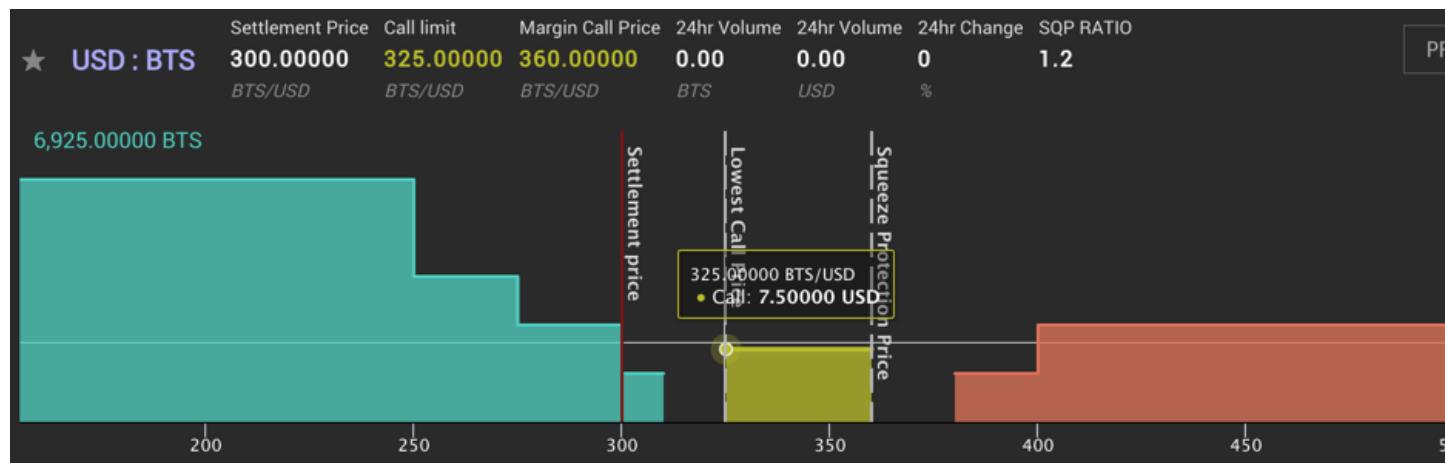
A margin call will occur any time the lowest ask is higher than the call price and lower than the SQPP. This has several consequences, as we will see below. It can create some very strange situations, and also force the margin called orders to “buy high”.

Consequence #1: Asks below the call price prevent margin calls from executing

Because margin calls only execute in the range Call Price - SQPP, if there is a sell order for 5 USD at 315 BTS/USD in this market, the call order will not use it, which makes the market look like this:



If a second sell order of 2.5 USD were added at 345 BTS/USD, the margin called order would still not buy any USD because of the “blocking” sell order at 315 BTS/USD:



If the order at 315 BTS/USD were to be removed, either from being cancelled or from being filled, the order at 345 BTS/USD would instantly get filled by the margin called order, and the margin called position would have a reduced debt of $10 - 2.5 = 7.5$ USD:



Consequence #2: Margin calls cannot “buy cheap” As we’ve seen above, unless the settlement price goes above the call price of the position, forced margin calls always buy at a premium relative to the settlement price. Even if there are sell orders available at or near the feed price, the margin called orders will not be matched with those sell orders if their call price is higher than the price of those sell orders.

Blockchain Governance

The blockchain can and needs to be 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). In contrast to most existing crypto currencies, we are not hoping for a fee market to grow but instead have the committee members deal with fine-tuning of the business plan. 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 committee member by showing that the business is worth being available in that particular chain

- Deploy the innovative business idea as a smart contract on the blockchain and have the shareholders approve the upgrade in combination with *Fee Backed Asset* that pays future fees of the smart contract to holders of that asset (*Fee Backed Asset*)

Vesting Balances

In BitShares 2, some balances are vesting over time. This feature has been introduced initially in BitShares 1 when merging several blockchain businesses into one blockchain.

Now, we make even more use of this functionality in such that an accounts income in form of

- worker pay,
- witness pay,
- the referral program, or
- cashback

is vesting over several days with different strategies.

For instance, a worker can define for how long he would like his pay to vest to encourage shareholders to vote for him due to no imminent additional sell pressure from the worker.

Strategies

CCD / Coin Days Destroyed

The economic effect of this vesting policy is to require a certain amount of “interest” to accrue before the full balance may be withdrawn. Interest accrues as coindays (balance * length held). If some of the balance is withdrawn, the remaining balance must be held longer.

Linear Vesting with Cliff

This vesting balance type is used to mimic traditional stock vesting contracts where each day a certain amount vests until it is fully matured.

Tutorials

List Vesting Balances

The vesting balances of an account can be seen from::

```
>>> get_vesting_balances <account-name>
```

and takes the form:

```
[ {
    "id": "1.13.241",
    "owner": "1.2.282",
    "balance": {
        "amount": 4158699804,
        "asset_id": "1.3.0"
    },
}
```

```

"policy": [
  1, {
    "vesting_seconds": 7776000,
    "start_claim": "1970-01-01T00:00:00",
    "coin_seconds_earned": "16408550952570000",
    "coin_seconds_earned_last_update": "2016-02-08T09:00:00"
  }
],
"allowed_withdraw": {
  "amount": 2114844530,
  "asset_id": "1.3.0"
},
"allowed_withdraw_time": "2016-02-08T11:26:12"
}
]

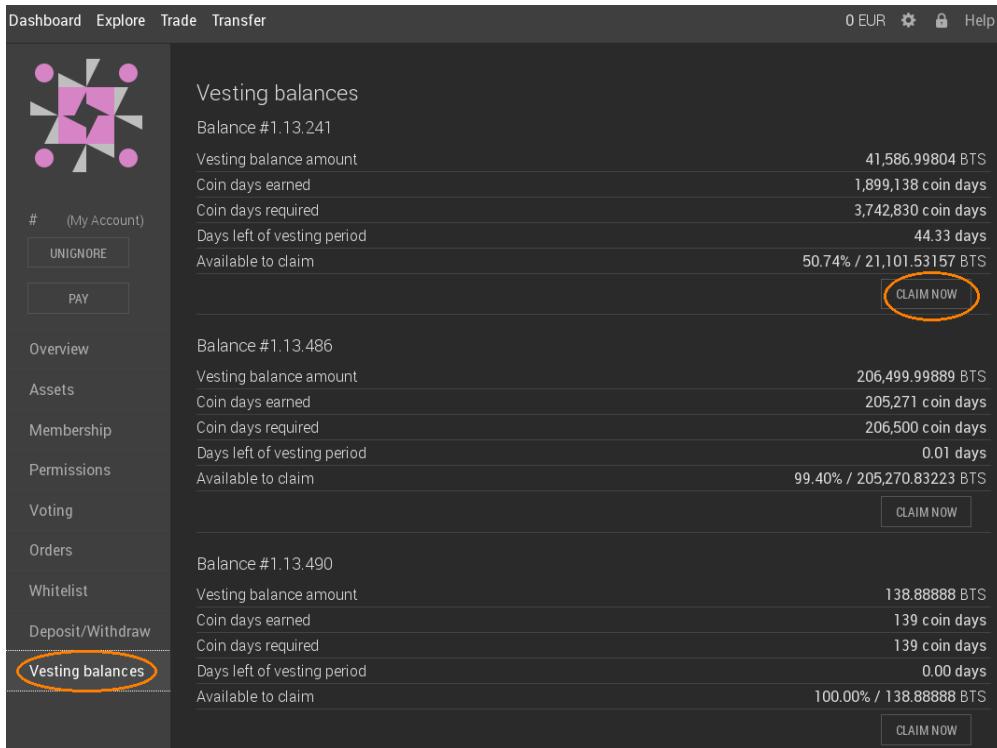
```

The balance gives the total vesting balance (amount plus asset), whereas allowed_withdraw shows the balance that can be withdrawn already. The object also tells us that the vesting policy is in terms of coin-days accrued (in contrast to linear vesting).

Claiming A Vesting Balance

Web Wallet

Claiming vesting balances using the web wallet (GUI) is quite simple. All you need to do is enter your account's page, click on *Vesting Balances* and pick the balance you would like to claim. The corresponding transaction is constructed automatically and will be signed after your confirmation.



Balance #	Vesting balance amount	Coin days earned	Coin days required	Days left of vesting period	Available to claim	Action
1.13.241	41,586.99804 BTS	1,899,138 coin days	3,742,830 coin days	44.33 days	50.74% / 21,101.53157 BTS	CLAIM NOW
1.13.486	206,499.99889 BTS	205,271 coin days	206,500 coin days	0.01 days	99.40% / 205,270.83223 BTS	CLAIM NOW
1.13.490	138.88888 BTS	139 coin days	139 coin days	0.00 days	100.00% / 138.88888 BTS	CLAIM NOW

Console Wallet

From the CLI wallet, vesting balances from witnesses can be claimed by using::

```
withdraw_vesting <account> <amount> <asset>
```

Unfortunately, no call exists for non-witness-pay vesting balances, yet but a transaction can be *constructed manually* with the operation `vesting_balance_withdraw_operation` and takes the form:

```
[  
  33, {  
    "fee": {  
      "amount": 0,  
      "asset_id": "1.3.0"  
    },  
    "vesting_balance": "1.13.0",  
    "owner": "1.2.0",  
    "amount": {  
      "amount": 0,  
      "asset_id": "1.3.0"  
    }  
  }  
]
```

Voting

If you hold some BTS tokens, you are considered a shareholder of the BitShares business and thus have a say in where it should be heading in future. As a shareholder you can cast a vote for three different entities within the network:

- **Block producers** bundle transactions into blocks and sign them with their signing key. These so called *witnesses* keep the blockchain alive by producing one block every few seconds and get paid by newly issued BTS shares similar to Bitcoin. Their second job is to produce reliable and accurate price feeds for the smartcoins.
- **Committee Members** *govern the blockchain and the business parameters*, and define the transaction fees.
- **Workers** are freelancers or businesses that provide a non-profitable service for the BitShares ecosystem. Essentially, they apply for a job in the ecosystem by providing actual work and get paid accordingly (if the shareholders approve).

Since voting might be a too time-consuming task for many shareholders, BitShares offers them a way to delegate their voting power to so called **proxies**. The sole purpose of proxies is to follow the ecosystem and be vote according to their *followers*. This is similar to many political votes where citizens vote for representatives which vote on their behalf.

Voting itself is **very simply** with the User interface and requires only a few clicks which are described in the *voting tutorials*

Note: The proxy `xeroc` is owned by the author of the documentation articles you are currently reading and has a discussion thread available.

Referral Program

The purpose of the referral program is to incentivize people to bring in more people. It compares to a Multi-Level-Marketing (MLM) scheme with the difference of having only **1 level** where referred individuals can opt-out by up-

grading their account to a *Life-Time Member (LTM)*. Every life time member, can get a cut of the fees based on child accounts linked to ours via referral.

BitShares has several different kinds of accounts: * Basic Account, and * Lifetime Member.

Basic Accounts are free, but do not qualify for the referral program, nor any cash back on transaction fees.

Lifetime Members pay an upgrade fee and earn **80% cash back on every fee they pay**. They also qualify for **80% of the fees paid by Basic Accounts they refer** to the network. These 80% can be split among the registrar, that actually registers the accounts, and an affiliate referrer, that brought in the new user.

Note: In Q1/2016, the *anual membership* has been removed from the code base and no longer exists. References to this kind of memberships can be safely ignored.

The referral fees are controlled by the blockchain and are distributed like this:

- 20% go to the network
- 80% go to the referral program
 - of this 80%, x% go to the registrar
 - of this 80%, 100%-x% go to the affiliate referrer

Note: For many cases it may make sense to upgrade the account even though you don't want to participate in marketing at all simply for the reasons to get a cashback of 80% of the fees you pay for your own transactions!

What to do?

If you want to participate in the referral program, you need to have a life-time member account, first! Then you can bring in new users by

- running your own faucet and actually register new accounts (will give you 80% of all the fees of those minus a fraction that you decide to give to affiliates (the referrers))
- referring people to a hosted wallet that offers you a cut of the fees as an affiliate.

In this case, most hosted wallets add your account as affiliate if you provide the following link structure to people

```
https://<url>/?r=<your-account>
```

with `<your-account>` being the name of your BitShares Lite-Time Member account.

Note: If you want link to pages with in the wallet, e.g. a particular decentralized market, you need to have the `?r=` parameter **before** the `#`, e.g.:

```
https://<url>/?r=<your-account>#/market/USD_BTC
```

Examples

When an Basic Account pays \$100 to become a Lifetime Member, \$50 is paid to their Referrer, \$30 is paid to the nearest Lifetime Member, and \$20 is paid to the Network. After this point the Lifetime Member becomes its own

referrer and nearest Lifetime Member and its prior Referrers no longer get any revenue from this user.

Terms & Conditions

Please see the [Referral Program - Terms & Conditions](#) for more details.

Claiming Referral Bonus and Cashback

If you have a life time member account and

- already paid some fees, or
- have referred people that paid some fees,

you can withdraw them in the “Vesting” tab of your account.

For Blockchain “Employees”

BitShares 2 separates responsibilities and incentives activities that are beneficial to the network, thus acknowledging different skill sets and interested community members to have incentives to contribute in the most appropriate way.

- Witnesses are paid for maintaining the back-bone of the network.
- Committee members are unpaid volunteers that organize the community and propose changes to the network.
- Marketers are paid in referral fees.
- Workers are paid for whatever they propose and do.
- Shareholders are people holding BTS. They can cast a vote and influence the DAC’s businesses

Each of the above (except Marketers) requires users to vote for the people, proposals, and/or changes. Those with sufficient approval will be compensated.

Workers are the “catch all” group where if you have an idea for something that could improve the network, you can get “paid” by the network to do it. Organizing meet-ups, developing a new tool or feature for the community, and maintaining websites and infrastructure (e.g. the mumble server team or linux distribution) are all examples of things workers may do.

Shareholder

In contrast to most crypto-currencies, BitShares does not claim to be a currency but rather an *equity* in a decentral autonomous company (DAC). As a result, the market valuation of BitShares is free floating and may be as volatile as any other equity (e.g. of traditional companies).

Every entity hold the core token (BTS) is considered a shareholder of the BitShares decentralized company.

Nonetheless, BTS tokens can be used as *collateral* in financial smart contracts such as market pegged assets and thus back every existing smartcoin such as the bitUSD.

Furthermore, each BTS can be used to cast a vote for

- *Block Producers*
- *Committee members*

and can thus participate in the corporate governance.

Committee

Since Bitcoin struggled to reach a consensus about the size of their blocks, the people in the cryptocurrency space realized that the governance of a DAC should not be ignored. Hence, BitShares offers a tools to reach on-chain consensus about business management decisions.

The BitShares blockchain has a set of parameters available that are subject of shareholder approval. Shareholders can define their preferred set of parameters and thereby create a so called *committee member* or alternatively vote for an existing committee member. The BitShares committee consists of several *active* committee members.

The BitShares ecosystem has a set of parameters available that are subject of shareholder approval. Initially, BitShares has the following blockchain parameters:

- **fee structure:** *fess that have to be paid by customers for individual transactions*
- **block interval:** *i.e. block interval, max size of block/transaction*
- **expiration parameters:** *i.e. maximum expiration interval*
- **witness parameters:** *i.e. maximum amount of witnesses (block producers)*
- **committee parameters:** *i.e. maximum amount of committee members*
- **witness pay:** *payment for each witnesses per signed block*
- **worker budget:** *available budget available for budget items (e.g. development)*

Please note that the given set of parameters serves as an example and that the network's parameters are subject to change over time.

Additionally to defining the parameters any active witness can propose a protocol or business upgrade (i.e. hard fork) which can be voted on (or against) by shareholders. When the total votes for the hard fork are greater than the median witness weight w then the hard fork takes effect.

Tutorials

Creating a New Committee Member

We can create a new committee member with::

```
>>> create_committee_member account "url" true
```

Howto Propose Committee Actions

Setting Smartcoin Parameters

This paragraph shows how the committee account can act using the proposed transaction system. Specifically, as an example I'm using the creation of BitShares proposal 1.10.21, a proposal to update a committee-controlled BitAsset to reduce `maximum_force_settlement_volume` for asset CNY from 2000 (20%) to 200 (2%).

First check the asset to see what its current configuration is:

```
>>> get_asset CNY
{
    ...
    "bitasset_data_id": "2.4.13"
}
```

Then check its bitasset object to get the currently active options:

```
>>> get_object 2.4.13
{
    ...
    "options": {
        "feed_lifetime_sec": 86400,
        "minimum_feeds": 7,
        "force_settlement_delay_sec": 86400,
        "force_settlement_offset_percent": 0,
        "maximum_force_settlement_volume": 2000,
        "short_backing_asset": "1.3.0",
        "extensions": []
    },
    ...
}
```

Then do update_bitasset to update the options. Note we copy-paste other fields from above; there is no way to selectively update only one field.

```
>>> update_bitasset "CNY" {"feed_lifetime_sec" : 86400, "minimum_feeds" : 7, "force_
→settlement_delay_sec" : 86400, "force_settlement_offset_percent" : 0, "maximum_
→force_settlement_volume" : 200, "short_backing_asset" : "1.3.0", "extensions" : []}_
→false
```

If this was a privatized BitAsset (i.e. a user-issued asset with feed), you could simply set the broadcast parameter of the above command to true and be done.

However this is a committee-issued asset, so we have to use a proposed transaction to update it. To create the proposed transaction, we use the transaction builder API. Create a transaction builder transaction with begin_builder_transaction command:

```
>>> begin_builder_transaction
```

This returns a numeric handle used to refer to the transaction being built. In the following commands you need to replace \$HANDLE with the number returned by begin_builder_transaction above.

```
>>> add_operation_to_builder_transaction $HANDLE [12, {"fee": {"amount": 100000000, "asset_id": "1.3.0"}, "issuer": "1.2.0", "asset_to_update": "1.3.113", "new_options": { "feed_lifetime_sec": 86400, "minimum_feeds": 7, "force_settlement_delay_sec": 86400, "force_settlement_offset_percent": 0, "maximum_force_settlement_volume": 200, "short_backing_asset": "1.3.0", "extensions": []}, "extensions": []}]
>>> propose_builder_transaction2 $HANDLE init0 "2015-12-04T14:55:00" 3600 false
```

The propose_builder_transaction command is broken and deprecated. You need to recompile with [this patch](#) in order to use the new propose_builder_transaction2 command which allows you to set the proposing account.

Then set fees, sign and broadcast the transaction:

```
>>> set_fees_on_builder_transaction $HANDLE BTS
>>> sign_builder_transaction $HANDLE true
```

Notes:

- propose_builder_transaction2 modifies builder transaction in place. It is not idempotent, running it once will get you a proposal to execute the transaction, running it twice will cause you to get a proposal to propose the transaction!

- Remember to transfer enough to cover the fee to committee account and set review period to at least `committee_proposal_review_period`
- Much of this could be automated by a better wallet command.

How to Approve/Disapprove a Committee Proposal

Approve Proposal

Now we need to convince the other committee members to approve. We can do so on the blockchain by asking them for approval with

```
>>> approve_proposal <fee-paying-account> <proposal-id> {"active_approvals_to_add" : [  
    ↵ "<MY-COMMITTEE-MEMBER>"]} true
```

where `<proposal-id>` takes the form `1.10.xxx` and identifies the actual proposal to approve.

Removing Approval

A previous approval can also be removed if the proposal is not yet expired, executed or within the preview period. This is done by::

```
>>> approve_proposal <fee-paying-account> <proposal-id> {"active_approvals_to_remove" :  
    ↵ [ "<MY-COMMITTEE-MEMBER>"]} true
```

Note that we now use `active_approvals_to_remove` instead of `active_approvals_to_add`.

How Committee Proposes a Change in Fee

Create an Proposal

Let's assume we want to propose a new fee for the account creation operation. We want 5 BTS as basic fee and want premium names to cost 2000 BTS. Additionally, a price per kbyte for the account creation transaction can be defined. We get

```
{  
    "account_create_operation" : {  
        "basic_fee" : 500000,  
        "premium_fee" : 200000000,  
        "price_per_kbyte": 100000}  
}
```

We propose the fee change for account `<committee_member>` with::

```
>>> propose_fee_change <committee_member> "2015-10-14T15:29:00" {"account_create_  
    ↵ operation" : {"basic_fee": 500000, "premium_fee": 200000000, "price_per_kbyte":  
    ↵ 100000}} false
```

Approve Proposal

Now we need to convince the other committee members to approve. We can do so on the blockchain by asking them for approval with

```
>>> approve_proposal <committee_member> "1.10.1" {"active_approvals_to_add" : ["<MY-
→COMMITTEE-MEMBER>"]} true
```

where 1.10.1 is the id of the proposal in question.

Witnesses

In BitShares, the witnesses' job is to collect transactions, bundle them into a block, sign the block and broadcast it to the network. They essentially are the block producers for the underlying consensus mechanism.

For each successfully constructed block, a witness is payed in shares that are taken from the limited reserves pool at a rate that is defined by the shareholders by means of approval voting.

How to run a witness is described in a *separated tutorial*.

Workers / Budget Items

Thanks to the funds stored in the reserve pool, BitShares can offer to not only pay for its own development and protocol improvement but also support and encourage growth of an ecosystem.

Payouts

A blockchain parameter (defined by shareholders through the committee) defines the daily available budget. No more than that can be paid at any time to all workers combined.

The daily budget is distributed as follows:

- The available budget is taken out of reserves pool.
- The budget items are sorted according to their approval rate (Pro – Con) in a descending order.
- Starting at the worker with the highest approval rate, the requested daily pay is payed until the daily budget is depleted.
- The worker with the least approval rate that was paid may receive less than the requested pay

Hence, in order to be successfully funded by the BitShares ecosystem, the shareholder approval for your budget item needs to be highly ranked.

Since the payments for workers from the non-liquid reserve pool result in an increased supply of BTS, these payments are vesting over a period of time defined by shareholders.

Working for BitShares

In order to be get paid by BitShares, a proposal containing

- the date of work begin,
- the date of work end,
- a daily pay (denoted in BTS),
- the worker's name, and
- an internet address.

has to be publish on the blockchain and approved by shareholders.

A worker can also choose one of the following properties:

- **vesting**: *pay to the worker's account*
- **refund**: *return the pay back to the reserve pool to be used for future projects*
- **burn**: *destroys the pay thus reducing share supply, equivalent to share buy-back of a company stock.*

A blockchain parameter (defined by shareholders through the committee) defines the daily available budget. No more than that can be paid at any time to all so called *workers* combined.

The daily budget is distributed follows:

- The available budget is taken out of reserves pool.
- The budget items are sorted according to their approval rate in a descending order.
- Starting at the worker with the highest approval rate, the requested daily pay is payed until the daily budget is depleted.
- The worker with the least approval rate that was paid may receive less than the requested pay

Hence, in order to be successfully funded by the BitShares ecosystem, the shareholder approval for your budget item needs to be highly ranked.

Since the payments for workers from the non-liquid reserve pool result in an increased supply of BTS, these payments are vesting over a period of time defined by shareholders.

A description on how to create your own worker can be found in the [tutorials](#).

Pseudo Workers

Three types of pseudo workers exist that are not primarily used to for salary.

Polling Workers

A worker proposal can be used to poll the shareholders for an opinion. Those workers usually have no or very small pay. Additionally, they come with a *proposal*, *recommendation* or other topic on which shareholders can publish a binary opinion (pro, or contra).

Refund Worker

This worker is used to set an approval limit for worker proposals and their payment by simply refunding his payment/salary to the reserve pool. If its amount of daily pay is as large as the daily available funds, and the worker has highest approval among all worker proposals, all funds will be returned to the reserves and no one will be payed. If, however, an other worker proposal has more votes than the refund worker, the proposal gets paid its salary, and the rest is return.

Burn Worker

This type of worker is similar to the *Refund Worker* above but **burns** his pay.

Marketer

Frequently Asked Questions

CLI Wallet FAQ

Why does the CLI client crash immediately when I try to run it for the first time?

The CLI client is unable to run on its own, i.e. without being connected to the witness node (via a web socket connection). So to successfully run the CLI client you need to do this:

- make sure you have this entry uncommented in the `witness_node_data_dir/config.ini` file `rpc-endpoint = 127.0.0.1:8090`
- before you start the CLI client, you need to start the witness node (and wait a while till it's up and running)

How can I close the CLI client in a clean way?

In Windows closing the whole window produces a nasty exception. In Windows you can try `ctrl-d` which stops the process but still produces a nasty exception.

How can I import to my CLI client a wallet originally created in the web GUI?

CLI and WEB wallet are two separated applications. They use separated ways to represent backups. You can currently only manually import keys from the GUI into the CLI.

How can I create, register and upgrade an account to Lifetime Membership?

Without already having an account, or knowing someone that has an account, it is not possible. You can't create accounts out of nowhere.

But you can work around it by importing an active key of a **Lifetime Member** account that has funds:

1. In the gui, go to the permissions tab of an account that is funded and has a LTM status.
2. Click on the BTS public key on the ACTIVE tab and copy the private key.
3. In the cli-wallet run: `import_key <account_name> <private_key>`
4. Then run: `suggest_brain_key` and copy the brain key. (You might want to make a backup of your brain key somewhere.)
5. Create a new account with this command: `create_account_with_brain_key <brainkey> <new_account_name> <imported_name> <imported_name> true`

This will create a new account called `<new_account_name>` and set the registrar and referrer to `<imported_name>`. The brainkey can be used to regenerate the account (even in the GUI wallet). You can manually delete the other active key from the `wallet.json` file.

Witness FAQ

What is the best way to interact with the witness node?

The only way you can interact with the witness node is through the CLI client by using its API. You can also use the GUI (i.e. the light client). In the GUI, change *Settings -> API connection*, add `ws://127.0.0.1:8090/ws` (according to settings of your witness node) and select it.

How do I check whether the witness node is already synced?

Run the `info` command in the CLI client and check the `head_block_age` value.

If it seems to be unable to sync beyond a certain date

You should always make sure you use the newest build available [here](#) as earlier releases will get stuck due to hard-forks.

Whose private key is [“BTS6MRyAjQ..”,”5KQwrPbwD..”]? Why is it predefined ion the config.ini?

It's a shared key for some special purpose. But I don't remember what it is. If I remember BM or someone else has ever explained it in the forum, but I can't find the post right now. Just let it be there. I think if you comment it out, it will appear again automatically, it's generated by the code of `witness_node`.

What is the meaning of all those different text colors in the witness node console?

- green - debug
- white - info/default
- yellow/brown - warning
- red - error
- blue - some kind of info, I don't know

Related source files are in `libraries/fc/include/fc/log/` and `libraries/fc/src/log/`.

How can I close the witness node in a clean way?

In windows use `ctrl-c`.

Is it safe to delete logs stored in `witness_node_data_dir/logsp2p`?

Yes, but

- they're rotated automatically after 24 hours anyway
- if you don't use them you should probably modify `config.ini` so they aren't written to disk in the first place.

What is the difference between public and private testnet?

Not much. The biggest difference is that public testnet are intended for wider audience and has fixed (not easy to change parameters), while private testnets can be setup with arbitrary settings.

Investor Guide

Note: This guide is still under construction. Please excuse if what you are searching for is not yet available

The investor guide serves as an entry point for existing and potential investors in the BitShares ecosystem. We here merely discuss the BTS token as well as investment opportunities available within BitShares itself and deliberately do not advertise 3rd party businesses. Please be reminded that this is an information platform and thus we do not give investment advice.

Claim your Investment

You are considered as a AngelShare holder if you have donated BTC or BTS to one of these addresses:

- **BTC:** 1ANGELwQwWxMmbdaSWhWLqBETPTkWb8uDc
- **PTS:** PaNGELmZgzRQCKeEKM6ifgTqNkC4ceiAWw

There is also an [AngelShare Explorer](#) specifically for these donations

AngelShares have been gifted 50% of the initial BTS shares. The other 50% went to AngelShares, the other went to holders of PTS.

Claiming your Stake

In order to claim your BTS, you need to look in your bitcoin wallet and search for transactions the the above mentioned address. The keys that correspond to the inputs of that transaction are what you need to obtain your BTS (*FAQ <<http://www1.agsexplorer.com/ags101>>*).

If you have located the private keys (in wallet import format - WIF), you can safely import them into your BitShares account using the *Import Keys* tools in the Wallet Management Console of your BitShares wallet.

If, however, you only have a *wallet.dat* file, you will need to

1. install BitShares 0.9.3
2. create an account in the BitShares 0.9.3 wallet
3. import the *wallet.dat* file into your account
4. *export a BitShares 2.0 compatible backup file*
5. *import the new backup into BitShares 2.0*
6. *claim you funds from the BitShares 2.0 genesis block*

Businesses, Developers and Business Developers

BitShares Improvement Proposal

BSIP stands for BitShares Improvement Proposal but can also seen as an improvement *protocol*. A BSIP is a design document providing information to the BitShares community, or describing a new feature for BitShares or its processes or environment. The BSIP should provide a concise technical specification of the feature or the idea and a rationale for it. It may not only describe technical improvements but also document *best-practises* and recommendations.

We intend BSIPs to be the primary mechanisms for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into BitShares. The BSIP author is responsible for building consensus within the community and documenting dissenting opinions.

Because the BSIPs are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal.

A detailed description of the structure of BSIPs can be found in [BSIP-0001](#)

Funding Your Ideas

You have a great idea for a business on top of within the BitShares platform and are searching for a way to fund your idea? Here are the options we can offer you:

- **Basic Worker Proposal:** All BitShares shareholders can cast a vote for your proposal/idea and will decide if they want this feature or not at the price you are asking. If they accept the price you asked, you will be `#paidbyprotocol` in BTS. If not, you can of course refine your proposal and try again incorporating feedback received from shareholders and the community.
- **Fee Backed Asset:** If you produce a feature for BitShares that offers a new service, you can have a [*Fee Backed Asset*](#) created for your feature and profit from future revenue streams generated by that particular feature.
- **Crowdfunding:** Instead of profiting from your FBA, you can sell them to investors to fund the implementation. Or you simply create, issue and sell a [*User Issued Assets*](#).
- **Mixed Funding:** You sell some of the FBA and keep the rest for yourself. Maybe the best option, you get money instantly to fund the development and you will get more later when people use the feature.

Remarks::

If you want to develop a blockchain/backend feature, you *should* probably cooperate with Cryptonomex as they can consult you how to implement everything on the blockchain and most shareholders will ask them to check the quality of your code prior to its hard fork, anyway.

Other tasks, such as basic GUI development, etc. can mostly be dealt with independently, unless access to the main repository is required. This can be asked from Cryptonomex or other active, but independent, web developers.

Tutorials

General Tutorials

How to Run and Use a Full Node

In order to improve decentralization of service, every user can run his own full node (often referred to *non-block-producing* witness node) and we here show how to do so.

Download and Install the Witness Node

We first need to download, (compile) and install the witness node. All that is needed is described here:

To reduce compilation time, you can tell the compile infrastructure to only compile the witness node by running

```
$ make witness_node
```

instead of

```
$ make
```

Running the Full Node

In order to run a full node that we can connect to, we need to open the RPC interface, this can be done by:

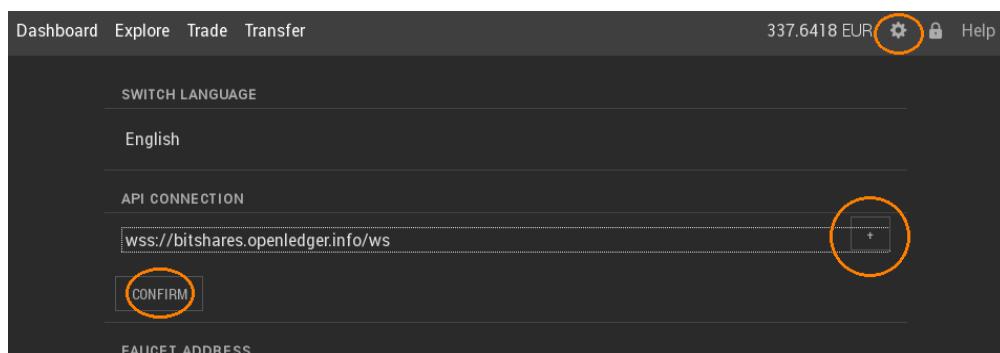
```
./programs/witness_node/witness_node --rpc-endpoint="0.0.0.0:8090"
```

This will open port *8090* and make it available over the internet (unless you run behind a router/firewall). If you want it to be open for your machine only, replace *0.0.0.0* by *localhost*.

Note, that the full node needs to synchronize the blockchain with the network first, which may take a few minutes.

Connecting to your own Full Node

In your wallet (may it be the light wallet or a hosted wallet) you can define the full node to which it should connect to in the preferences:



For your own full nodes, type:

```
ws://localhost:8090
```

and confirm.

Congratulation, you are now connected to the network via your own full node.

Account Registration

The process of registering a new account on the blockchain consists of two parts:

- Picking a random brain key and deriving a *priate/public* key pair
- Create the corresponding account and registering it on the blockchain

Brain, Private and Public Key Derivation

We can derive a new set of keys using the `suggest_brain_key` command in the `/apps/cliwallet`. The result will look like this:

```
>>> suggest_brain_key
{
    "brain_priv_key": "FILINGS THEREOF ENSILE JAW OVERBID RETINAL PILULAR RYPE CHITTY_",
    ↪RAFFERY HANDGUN ERANIST UNPILE TWISTER BABYDOM CIBOL",
    "wif_priv_key": "5JVrt2921aikA7QP5ZCtR2sJh4wbEnsHsK6qo67Shnk9ArKMzNT",
    "pub_key": "BTS7D8jpQ2UwaQxqKyGpuhFQ9LBugCNxDhE7UN2jqgjVQgzG7zo9n"
}
```

The hierarchy for these values goes like this::

```
HASH(brain_priv_key) -> wif_priv_key
HASH(HASH(brain_priv_key)) -> pub_key
```

Hence, if you keep the brain key, you will be able to recover your required keys to access your account and/or funds.

Note: Even though `suggest_brain_key` shows only one private key that will be used for the **owner authority** most wallet implementations will derive an additional second private key to be used for the **active authority**!

Creating and Registering an account

If you want to create and register a new account on your own because you have the funds in another account and don't want someone else involved, you can make use of the command `create_account_with_brain_key`:

```
>>> create_account_with_brain_key <brain_key> <account_name> <registrar_account>
↪<referrer_account> <broadcast>
```

For our example, we would get::

```
>>> create_account_with_brain_key "FILINGS THEREOF ENSILE JAW OVERBID RETINAL PILULAR_",
↪RYPE CHITTY RAFFERY HANDGUN ERANIST UNPILE TWISTER BABYDOM CIBOL" mywallet myfunds_
↪anonymous 100 true
```

Registering an Account

If you want to register the account of someone else, all you need is the public key. In theory, the BitShares blockchain distinguishes three keys for each account, namely the **owner**, **active**, and the **memo** key. However, for the sake of simplicity, we here make use of only one public key (see example above).

In order to register an account, we need another account that has enough funds to pay the fee for the registration transaction. This account will be called `registrar_account`. Another account `referrer_account` can be registered that will get `referrer_percent` of the referral bonus program. Any registered account can take the role of the referrer. Hence we here say that user `anonymous` has referred us. The syntax goes like this::

```
>>> register_account name, owner_pubkey, active_pubkey, registrar_account, referrer_
↪account, referrer_percent, broadcast
```

For our example we say we register a new user called `mywallet`, use the pubkey derived above and let our account `myfunds` pay the fee::

```
>>> register_account mywallet BTS7D8jpQ2UwaQxqKyGpuhFQ9LBugCNxDhE7UN2jqgjVQgzG7zo9n_
->BTS7D8jpQ2UwaQxqKyGpuhFQ9LBugCNxDhE7UN2jqgjVQgzG7zo9n myfunds anonymous 100 true
```

Note: Note that in order to register an account, the registrar (here: `myfunds`) needs to be a **lifetime member!**

Manually Construct Any Transaction

General Procedure

The general principle for generating, signing and broadcasting an arbitrary transactions works as follows:

1. Create an instance of the transaction builder
2. Add arbitrary operation types
3. Add the required amount of fees
4. Sign and broadcast your transaction

The corresponding API calls in the *CLI Wallet* are:

```
>>> begin_builder_transaction
>>> add_operation_to_builder_transaction $HANDLE [opId, {operation}]
>>> set_fees_on_builder_transaction $HANDLE BTS
>>> sign_builder_transaction $HANDLE true
```

The `begin_builder_transaction` call returns a number we call `$HANDLE`. It allows to construct several transactions in parallel and identify them individually!

The `opId` and the JSON structure of the `operation` can be obtained with:

```
get_prototype_operation <operation-type>
```

The operation types available are:

```
typedef fc::static_variant<transfer_operation, limit_order_create_operation, limit_order_cancel_operation, call_order_update_operation>
```

In practise, each operation has to pay a fee, and hence, each operation has to carry a `fee` member. When crafting a transaction, you now have the choice between either defining each fee for your operations individually, or you use `set_fees_on_builder_transaction` that sets the fee for each operation automatically to the chosen asset.

Example: Transfer

A simple `transfer` takes the following form:

```
get_prototype_operation transfer_operation
[
  0, {
    "fee": {
      "amount": 0,
      "asset_id": "1.3.0"
    },
    ...
  }
]
```

```
"from": "1.2.0",
"to": "1.2.0",
"amount": {
    "amount": 0,
    "asset_id": "1.3.0"
},
"extensions": []
}
]
```

The operation id for the `transfer_operation` is thus 0 (third line) and the core elements (removing fee) of this operation take the form:

```
{
"from": "1.2.0",
"to": "1.2.0",
"amount": {
    "amount": 0,
    "asset_id": "1.3.0"
}
}
```

We add an operation to a transaction as follows (line breaks inserted for readability):

```
>>> begin_builder_transaction
0
>>> add_operation_to_builder_transaction
    0
[0, {
    "from": "1.2.0",
    "to": "1.2.0",
    "amount": {
        "amount": 0,
        "asset_id": "1.3.0"
    }
} ]
```

The corresponding `id` can be obtained with `get_account`, and `get_asset`.

We add a fee payed in BTS, sign and broadcast the transaction (if valid):

```
>>> set_fees_on_builder_transaction 0 BTS
>>> sign_builder_transaction 0 true
```

Proposing a Transaction

Proposed transactions can be used everywhere multiple parties have to agree for a transaction to become valid.

Crafting a Transaction

If is recommended that the reader first reads through the following tutorial:

- *Manually Construct Any Transaction*

Proposing a Transaction

A proposed transaction is encapsulated within another operation type. We can achieve this by slightly modifying our procedure:

```
>>> begin_builder_transaction
>>> add_operation_to_builder_transaction $HANDLE [opId, {operation}]

# New:
>>> propose_builder_transaction2 $HANDLE <proposing-account-name> <expiration>
  ↪<review-period-secs> false

>>> set_fees_on_builder_transaction $HANDLE BTS
>>> sign_builder_transaction $HANDLE true
```

Definition

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

```
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::min(
uint32_t
refresh =
view_period_seconds
=
0,
bool
broadcast =
true)
```

Approving a Proposal

A proposal can be approved simply by:

```
approve_proposal <proposing-account> <proposal_id> { "active_approvals_to_add" : [
    ↳<account-name-required-for-approval>" ] } false
```

When replacing the final `false` with `true`, the transaction will be broadcasted!

Available approval options are:

- `active_approvals_to_add`
- `active_approvals_to_remove`
- `owner_approvals_to_add`
- `owner_approvals_to_remove`
- `key_approvals_to_add`
- `key_approvals_to_remove`

Definition

```
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

Example: Setting Smartcoin parameter

A simple `asset_update` takes the following form:

```
get_prototype_operation asset_update_bitasset_operation
[
  12, {
    "fee": {
      "amount": 0,
      "asset_id": "1.3.0"
    },
    "issuer": "1.2.0",
    "asset_to_update": "1.3.0",
    "new_options": {
      "feed_lifetime_sec": 86400,
      "minimum_feeds": 1,
      "force_settlement_delay_sec": 86400,
      "force_settlement_offset_percent": 0,
      "maximum_force_settlement_volume": 2000,
      "short_backing_asset": "1.3.0",
      "extensions": []
    },
    "extensions": []
  }
]
```

The operation id for the `asset_update_bitasset_operation` is thus 12 (third line) and the core elements (removing fee) of this operation take the form:

```
{
  "issuer": "1.2.0",
  "asset_to_update": "1.3.0",
  "new_options": {
    "feed_lifetime_sec": 86400,
    "minimum_feeds": 1,
    "force_settlement_delay_sec": 86400,
    "force_settlement_offset_percent": 0,
    "maximum_force_settlement_volume": 2000,
    "short_backing_asset": "1.3.0",
    "extensions": []
  },
  "extensions": []
}
```

We add an operation to a transaction as follows (line breaks inserted for readability):

```
>>> begin_builder_transaction
0
>>> add_operation_to_builder_transaction
0
[12, {
    "asset_to_update": "1.3.113",
    "issuer": "1.2.0",
    "extensions": [],
    "new_options": {
        "feed_lifetime_sec": 86400,
        "force_settlement_delay_sec": 86400,
        "short_backing_asset": "1.3.0",
        "maximum_force_settlement_volume": 200,
        "force_settlement_offset_percent": 0,
        "minimum_feeds": 7,
        "extensions": []
    },
}]
}
```

The corresponding asset id can be obtained with `get_asset`.

Now let's make it a proposal for the committee members to sign:

```
>>> propose_builder_transaction2 0 init0 "2015-12-10T14:55:00" 3600 false
```

We add a fee payed in BTS, sign and broadcast the transaction (if valid):

```
>>> set_fees_on_builder_transaction 0 BTS
>>> sign_builder_transaction 0 true
```

Confidential Transfers

This tutorial shows how to use the CLI wallet to perform confidential transfers in BitShares. A confidential transfer is one that hides both the amount being sent and the parties involved in the trade. Confidential transfers are also referred to as blinded transfers. When privacy is important no account should ever be used twice, and coupled with diligent measures to backup and protect the wallet and document the cryptographic keys used it is impossible for any third party to identify how money is moving by using blockchain analysis alone.

It is important to realize that the current implementation of Stealth functionality requires careful attention to detail and entails a higher level of risk for loosing your account balance if the steps followed herein are followed casually and without regard to such potential loss. However, by following this guide you can rest assured your balance will be held in the blockchain but remain totally and completely hidden from everyone but you. With that staunch admonition out of the way let's get started.

We will illustrate the CLI commands required to complete every step in this tutorial. You must be familiar with the `witness_node` and `cli_wallet` software to follow this guide. For further information on how to build and use that software consult the [readme file](#) on github or the BitShares wiki. I will be using simple passwords and account names to simplify the presentation. You should change these to more secure values to provide a higher level of security. Also note that the CLI wallet echos the command entered which is not reflected in the quoted session logs in the examples that follow.

Step 1: Create a Blind Account

Blind Accounts are not registered on the blockchain like the named accounts of BitShares. Instead a blind account is nothing more than a labeled public key. The label assigned to the key is only known to your *wallet*. Thus it is crucial

that you create a new wallet for the blind account and back it up after completing the balance transfer.

The first step in creating a blind account is to create a new wallet and set a good quality password for it that would be difficult to crack. Then, using this wallet we'll create a labeled account and protect it with a "brainkey". The "brainkey" is effectively the private key used by your account. All BitShares cryptography is based on public / private key pairs, one public which can be shared the other private known only to you.

For confidential accounts the "brainkey" is only stored in the wallet, so if the wallet file is lost or destroyed and you have not recorded the "brainkey" on paper or some other place there will be no way to recover your confidential account balance. Even if you do record your "brainkey" elsewhere outside of the wallet, I do not believe any recovery methods yet exist to import your "public key / brainkey" pair into a wallet so you could access your confidential balance. At least it would be possible to do so at some future date, unlike the impossibility it would be if you lost the wallet and failed to record the "brainkey".

```
>>> create_blind_account alice "alice-brain-key which is a series of words that are_
    ↵usually very long"
1483572ms th_a      wallet.cpp:743           save_wallet_file     ] saving_
    ↵wallet to file /home/admin/BitShares2/blindAliceWallet
BTS5Qmr9H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNak7mse
true
```

The result of the `create_blind_account` command is to print out the Public Key associated with the blind account named `alice` and the command success or failure (`true`). Note that the CLI interface will update the wallet file you specified on the command line following the `-w` when you started the `cli_wallet`. The CLI wallet has no command to quit or exit the interface so we terminate the session with a control-C (shown as `^C`), which returns us to the operating system shell.

Step 2: Transferring a Balance From a Standard Public Account to a Blind Account

Now that we've created a new, unregistered blind account named `alice` we can transfer assets into it from any source, public or not. We'll describe the steps to transfer a balance from one blind account to another momentarily, which is essential to fully obscure a balance from public view, but for now we'll illustrate a transfer from a publicly registered account to our newly created blind account.

To begin, we need a wallet that has an account with assets. It can be any account with assets, so we'll use Peter's Public account and transfer funds from that into the `alice` account. We will also need to have the public key handy that was printed for the `alice` account when it was created. Since the `alice` account is not registered we need a way to refer to it before we can do the transfer. So in this CLI session we'll also show you how to create a label to do that and then we'll use that label to transfer assets into the `alice` blind account.

```
>>> list_account_balances "peters-public-registered-account"
5000 BTS

>>> set_key_label "BTS5Qmr9H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNak7mse" "Alice-is-
    ↵Blind"
true

>>> transfer_to_blind "peters-public-registered-account" BTS [[Alice-is-Blind, 100]]_
    ↵true
3369305ms th_a      wallet.cpp:3794           transfer_to_blind     ] to_amounts:_
    ↵[[{"Alice-is-Blind", "100"}]]
peters-public-registered-account sent 100 BTS to 1 blinded balance fee: 40 BTS
100 BTS to Alice-is-Blind
receipt:_
    ↵2B2vTjJ19hgqzGp8qdc8MEWmsgEUGEcnJcoQTYNQqMU8bRoFmbQYemXs56FoUc4Z5PdVM65nsyS7gwJMq9Z_
    ↵SkpWQFhEqLGuZi1N3jQm8yBwaLD2DQzwY5AEW1rSK9HWJbfqNLtx8U4kc3o9xKtJoED2SgHW6jDQ7igBTcVhuUiKSswFu3DFa6L_
    ↵Wm5khjgy1LrR5uhmp
```

```
>>> list_account_balances "peters-public-registered-account"
list_account_balances verbaltech2
4860 BTS
```

The above 2 steps transmit BTS assets from a public, registered account named “peters-public-registered-account” into a single unregistered blind account named alice using a label to refer to it named “Alice-is-Blind”. It is important to note that these labels are NOT persistent from one CLI session to the next, so every time you transfer assets from a source account such as “peters-public-registered-account” used here to a blind account you will need to set a label to refer to the blind account.

Adding a Contact

There is currently no facility to transfer assets to a blind account from the light wallet or the OpenLedger web wallet. They only support the WIF (Wallet Import Format) and thus will not accept your blind account’s “brainkey” as a valid private key. In the future you may be able to avoid setting labels each time you transfer from a public to a blind account by defining a contact. However, keep in mind that every association you establish in the path between a public account and a confidential account may make it that much easier to trace your steps, so think twice about the tradeoffs you make for the sake of convenience. They just might circumvent the measures you are taking to hide your balance. This is also true if you transfer assets directly between a public account to a confidential account and leave them in the confidential account. To totally obscure where your balance is held you need to transfer to at least 2 different confidential accounts. We will cover this in a bit more detail later. In the next step we’ll look at how to receive the transmitted assets into alice’s blind account.

Step 3: Receiving an Asset Balance Transmitted From Another Account

Transferring assets from one account to a confidential account involves at least 2 steps, the first to transmit the assets and the second to receive them into the confidential account. We covered the process required to transmit assets in Step 2, now lets see what it takes to complete the transfer and verify we have the correct balance:

```
>>> receive_blind_transfer
-> "2B2vTjJ19hgqzGp8qdc8MEWmsgEUGECNJcoQTYNQqMU8bRofmbQYemXs56FoUc4Z5PdVM65nsyS7gwJMq9ZSkpWQFhEqLGuZi
-> "peter" "from Peter"
100 BTS peter => alice "from Peter"
```

Using the balance receipt value returned from the transfer_to_blind command in Step 2 we can receive (i.e. import) the balance into alice’s blind account. Note that the source of the balance must be labeled which is the parameter that follows the long balance receipt key. It is meant to represent to source account from which the assets are being transferred, however it need not be. The last of the 3 parameters is a memo text field which is an arbitrary text value. Note that all 3 parameters are required. In the next section we will describe how to list the confidential accounts and their balances so that we can verify our transfer is correct and complete.

Listing Blind Accounts and Their Balances

For any wallet in which you have created confidential accounts you can list the accounts present using the “get_my_blind_accounts” CLI command, and use the accounts returned from that to obtain their balances:

```
>>> get_my_blind_accounts
[[
"alice",
"BTS5Qmr9H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNak7mse"
]]
```

```
>>> get_blind_balances "alice"
100 BTS
```

To review, you have learned how to:

1. create a new CLI wallet and add a blind account to it
2. create a label to refer to a blind account
3. send assets from a public account to a blind account
4. receive or import assets sent from another account into a blind account
5. list the blind accounts contained in a cli wallet
6. list the asset balances of blind accounts

These are the basic steps for a simple unidirectional transfer of a single asset from a public account to a single blind account. Next we will examine how to cover our trail to obscure our balance by using a second blind account and finally we will see how to transfer from a blind account back into a public account to wrap up our look into protecting your assets with confidential accounts using the CLI wallet.

The first part was a basic demonstration of how to use the BitShares CLI wallet to transfer an asset from a registered, public account to a confidential (i.e. blind) account. It explained the steps involved and the current limitations of using confidential features. Here in part 2 we will show how to transfer assets from one confidential account to another, and conclude our look at confidential by describing how to transfer assets from a confidential account back into a registered public account.

The first part mentioned that to truly hide an account balance and eliminate any public tractability of how the assets arrived there, at least 2 confidential accounts should be used in the path from public source to final confidential destination. This is due to the fact that the destination address of transfers from a public account are visible. There may be no way for the public to query the holdings of confidential accounts but it would not be wise to leave assets in such an obvious hiding place either.

However, if those assets are moved to yet another confidential account there is no way their whereabouts can be traced through blockchain analysis alone. Because transfers between confidential addresses cannot be traced, even the inference that assets remain in the first confidential address (the destination out of the public registered account) is highly questionable. Additional layers of confidential to confidential transfers would provide even greater security that assets cannot be found for those with a higher sense of paranoia. It should go without saying that disbursing assets to multiple confidential accounts is an important security strategy for large balances. Lastly, be aware that the assets held in confidential accounts are not counted for purposes of voting. Thus you should consider how the use of confidential accounts will affect your participation and influence in the policies and proposals of the BitShares ecosystem.

Step 4: Transferring Assets Between Confidential Accounts

Let's start by creating a second wallet and confidential account we will use as our hypothetical final destination. We'll call this account bobby. We've already shown how to do this in part 1, but you may wish to review those basic steps before you continue.

```
>>> create_blind_account bobby "bobby-brain-key which is a series of words that are_
↳ usually very long"
1434971ms th_a          wallet.cpp:743           save_wallet_file      ] saving_
↳ wallet to file /home/admin/BitShares2/blindBobWallet
BTS6V829H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNakaV26
true
```

We need to restart the CLI wallet with the alice account, where we have a 100 BTS balance. We will create a label to refer to Bob's confidential account (bobby) and transfer some BTS assets from alice to bobby. Note that the process is the same as before, and we need to set a label for the bobby (destination) account to do the transfer.

```
>>> set_key_label "BTS6V829H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNakaV26" "bobby"
true

>>> blind_transfer alice bobby 80 BTS true
318318ms th_a          wallet.cpp:743           save_wallet_file      ] saving_
→wallet to file /home/admin/BitShares2/blindAliceWallet
blind_transfer_operation temp-account fee: 15 BTS
5 BTS to alice
receipt:
→iiMe3q3X4DqW1AqCXfkYEcuRsRATxMwSvJpaUuCbMTcxRUUGeBPwYU1SRRs4tEQGPNmP
→$Js4jTJkDGHzUm33o6h14wa1XNsmedLJCKnwmyGeqFB4vPRk9ZxnaizbMNu8bHr62xQaTc73ALxAZEPRdkNLyqMk
→$oDEFja3vCPgcyDYCQmkVnNiAQaKeMG83KrW11QZMHQZfzZ8ofTSTEy8qruLAa27vrjAM6q2ckbD8ZTNMWhkSWniq
→$4fay3Tbcd2zsy9EgxuxN

80 BTS to bobby
receipt:
→iiMe3q3X4DqW1AqCXfkYEcuRsRATxMwSvJpaUuCbMTcxRUUsn1qUt jfqLYuaNycrpKHfmUG1PR9mx2d2nVKB15RYSryyjSn54AD
```

There is a bit more output printed than what is shown above, but the important results are provided. From this you can see we first set a label to refer to the newly created “bobby” account, and the blind_transfer command fee was 15 BTS, which sent 80 BTS of the balance (100 BTS was transferred to the alice account in Part 1) to the bobby confidential account and provided 2 balance receipts: the first for 5 BTS coming back to the alice account as returned change (leftover funds), and the second which is the receipt for the 80 BTS being sent to the bobby account, which we will need in order to receive the transfer in the bobby account contained in the blindBobWallet file.

As you can see using confidential in the CLI wallet is a rather tedious “manual” process. Do note however that you do not need to do a “receive_blind_transfer” to import the 5 BTS change back into the alice account, at least that is taken care of. Also important to note is as far as the outside world can see alice sent some amount less than 100 BTS to two new outputs, one of which is the change returned, which makes it yet that much more difficult to track what is going on, especially since the amounts of each output are invisible.

Step 5: Transferring Assets From a Confidential Account Back to a Public Account

In this final step of our round-trip process we will transfer some of the BTS from the bobby confidential account back to original public account named peter we started out with. There is nothing new required to accomplish for this step, but a couple of points are worth mentioning. First, keep in mind that the source address for transfers coming into a public account may be visible, so consider using one or more intermediary confidential accounts to add layers of insulation between the public account and the resting place for your confidential assets. Second, although you are sending to a registered, public account which one might think needs no label to access, that isn't the case.

A label must be assigned to the public destination address to return assets from a blind (confidential) account. The public key value for the account is readily available using the account's permission page explorer. Use the account/key shown under the Active Permissions heading.

```
>>> receive_blind_transfer
→"iiMe3q3X4DqW1AqCXfkYEcuRsRATxMwSvJpaUuCbMTcxRUUsn1qUt jfqLYuaNycrpKHfmUG1PR9mx2d2nVKB15RYSryyjSn54AD
→"alice" "from Alice"
100 BTS alice => bobby "from Alice"

set_key_label BTS9oxUqKFD8gfGoXb6AwDBEoBt8W47g4Mt z8SW8inUeHemR9nOi9 "peter"
true
```

```

>>> blind_transfer bobby peter 50 BTS true
2263915ms th_a           wallet.cpp:743          save_wallet_file      ] saving_
    ↵wallet to file /home/admin/BitShares2/blindBobWallet
blind_transfer_operation temp-account fee: 15 BTS
15 BTS to bobby
receipt:
    ↵boqRZqyKaZW6bExrstPwFdXvzUBJSjGeaqy482NxBJ6S9Un4zimalmzysTrUipBiBpm4CrLTvCJZfqDaAaqEpmxWAWAKhi2Gm
50 BTS to peter
receipt:
    ↵boqRZqyKaZW6bExrstPwFdXvzUBJSjGeaqy482NxBJ6S9VPCqArXCypsZWZnpCeG7jfS3oUnbtmn5bmmVH5HCXJg9QxCmn4po
    ↵o3SqaCF43MRDK3ouYrFBcAK2TPXfnnvAU3r1UvhNHpxuNaS1cebd88Nn6BTxSifKdJ8ysFft98e88Cbek

>>> get_blind_balances bab1
get_blind_balances bab1
15 BTS

```

The explanation for this CLI session is essentially the same as it was for step 4. Although the account information is different the commands used and their role in the transfer process are the same.

One last example demonstrates how to split a balance between multiple confidential accounts. This is very useful because it not only saves on transfer fees it also obscures what amounts end up where. The point of showing this is primarily to illustrate the syntax of the command.

```

>>> list_account_balances "peters-public-registered-account"
4860 BTS

>>> set_key_label "BTS5Qmr9H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNak7mse" "Alice-is-
    ↵Blind"
true

>>> set_key_label "BTS6V829H9SM39EHmVgXtsVjUGn2xBUtqbF6MdQ6RpnxUWNakaV26" "bobby"
true

>>> transfer_to_blind peter BTS [[alice,800],[alice,2000],[bobby,2000]] true
peter sent 4800 BTS to 3 blinded balances fee: 40 BTS
800 BTS to alice
receipt:
    ↵2Dif6AK9AqYGDLDLYcpcwBmzA36dZRmuXXJR8tTQsXg32nBGs6AetDT2E4u4GSVbMKEiT154sqYu1Bc23cPvz$AyPGEJTLkVpi
2000 BTS to alice
receipt:
    ↵28HrgG1nzkGEDNnLleZmNvN9JmTVQp7X88nf7rfayjM7sACY8yA7FjV1cW5QXHi1sqv1ywCqfnGiNBqDQWMwpGB1KdRwDcJPa
2000 BTS to bobby
receipt:
    ↵82NxBJ6S9Un4zimalmzyboqRZqyKaZW6bExrstPwFdXvzUBJSjGeaqy4sTrUipBiBpm4CrLTvCJZfqDaAaqEpmxWAWAKhi2Gm
20 BTS to peter
receipt:
    ↵cwBmzA36dZRmuXXJR8tTQs2Dif6AK9AqYGetDT2E4u4GSVbMKEiT154saot4e1FUDnNPz41uFDLDLYcpXg32nBGs6Afu2G6rpgr

```

In this case the only thing the public sees is that account ‘peter’ sent 4800 BTS to four different places. Note that although 800 and 2000 BTS were sent to the alice confidential account they do not show up that way on the blockchain.

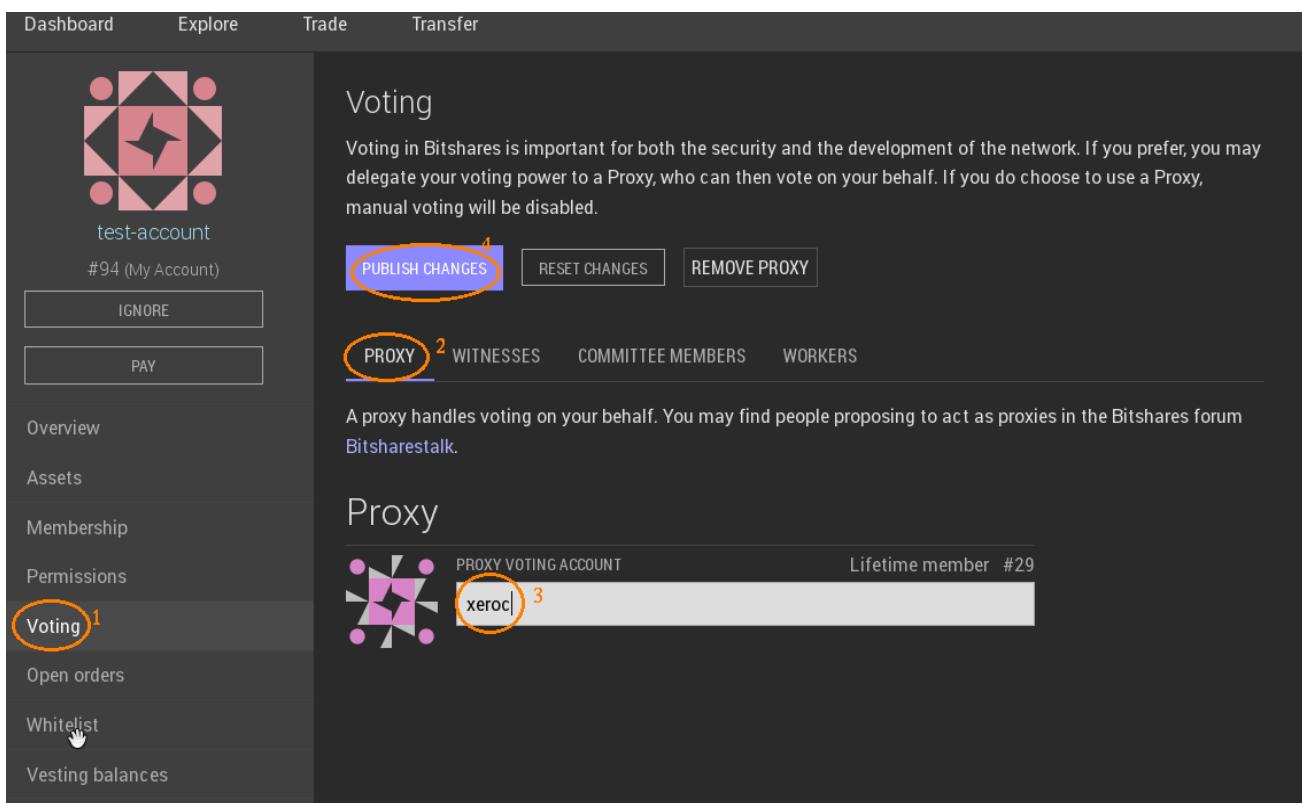
Conclusion: The outside world has no idea *how much* is in each output, only that they all add up to 4800 BTS.

Voting

Casting your vote or setting your proxy is very simply using the user interface (e.g. hosted wallet or light wallet).

Setting a proxy

The picture below shows how to set your trusted proxy:



Note: The proxy `xeroc` is owned by the author of the documentation articles you are currently reading and has a discussion thread available.

Voting for Witness, Committee Member or Worker

If you have not set a proxy, you can cast your own vote for witnesses, committee members or workers and publish your vote as shown in the picture below.

Voting

Voting in Bitshares is important for both the security and the development of the network. If you prefer, you may delegate your voting power to a Proxy, who can then vote on your behalf. If you do choose to use a Proxy, manual voting will be disabled.

PUBLISH CHANGES 5 RESET CHANGES

PROXY 2 WITNESSES 3 COMMITTEE MEMBERS 4 WORKERS 4

Witnesses are the block producers of Bitshares. They validate transactions and ensure the safety of the network. You may vote for as many witnesses as you'd like, and they will all receive the same amount of votes. Witness proposals can be found here: [Bitsharestalk](#).

WITNESS	NAME	WEBPAGE	VOTES
	Account name	ADD	

Note: If you already have a proxy defined and want to start voting on your own, you need to first click Remove proxy!

Howto trade in the DEX

The decentralized exchange (DEX) of BitShares has a similar look&feel as traditional centralized exchanges. However, trading in the DEX can have many different appearances, depending on what user-interface is used. We here describe the user interface of the official wallet. We recommend to also read through:

- *Decentralized Exchange*
- *Trading*

Playing Orders

Orders can be placed in the same way as everywhere else, by providing

- the amount to buy/sell
- the price at which to buy/sell

The screenshot shows two side-by-side order forms on a dark-themed exchange interface.

BUY BTC Form:

- Price: 0.0
- Amount: 0.0
- Total: 0.0
- Fee: 0.23386
- Balance: 5.0401 USD
- Lowest ask: 442.00000109 USD
-

SELL BTC Form:

- Price: 0.0
- Amount: 0.0
- Total: 0.0
- Fee: 0.23386
- Balance: 0.00008442 BTC
- Highest bid: 380.00000000 USD
-

Fees

In contrast to other exchanges, BitShares asks for a tiny **flat fee** for placing an order. This fee can be payed in USD, BTC, or GOLD and is independent of the actual assets that are traded.

If you cancel an order that has not been fully or partially filled, 90% of the fee will be payed back to your account. However, this chargeback will be in BTS and not in the asset you have originally paid the fee in.

API Usage

How to Run and Use the Cli-Wallet

The Cli-Wallet is used to interact with the blockchain. Everything that adds new data to the blockchain requires a signature from a private key. These signed transactions can be produced by the cli-wallet.

Download and Install the Witness Node

We first need to download, (compile) and install the cli-wallet. All that is needed is described here:

To reduce compilation time, you can tell the compile infrastructure to only compile the witness node by running.

```
$ make cli-wallet
```

instead of

```
$ make
```

Executing the cli-wallet

All it takes for the cli-wallet to run is a trusted **public API server** to interface with the blockchain. These public API servers are run by businesses and *individuals*. In this example, we use the public API node of OpenLedger and connect via secured websocket connection:

```
wss://bitshares.openledger.info/ws
```

```
./programs/cli_wallet/cli_wallet -s wss://bitshares.openledger.info/ws
```

This will open the cli-wallet and unless you already have a local wallet, will ask you to provide a passphrase for your local wallet. Once a wallet has been created (default wallet file is `wallet.json`), it will prompt with

```
locked >>>
```

The wallet can be unlocked by providing the passphrase:

Note: The passphrase is given in clear text and is echoed by the wallet!

```
locked >>> unlock "supersecretpassphrase"
null
unlocked >>>
```

After this point, you can issue *any command available to the cli-wallet* or construct your *own transaction manually*.

You can get more detailed information either by pressing *Tab*, twice, or by issuing `help`. Detailed explanations for most calls are available via

```
unlocked >> gethelp <command>
```

Note: Many calls have a obligatory broadcast-flag as last argument. If this flag is `False`, the wallet will construct and sign, but **not** broadcast the transaction. This can be very useful for a cold storage setup or to verify transactions.

Opening RPC port

The cli-wallet can open a RPC port so that you can interface your application with it. You have the choices of

- websocket RPC via the `-r` parameter, and
- HTTP RPC via the `-H` parameter:

```
./programs/cli_wallet/cli_wallet -s wss://bitshares.openledger.info/ws -H 127.0.0.0
→1:8092 -r 127.0.0.1:8093
```

Transferring Funds using the cli-wallet

Once we have the *cli-wallet set up*, we can transfer funds easily with the following syntax:

```
unlocked >> transfer <from> <to> <amount> <asset> <memo> <broadcast>
```

Note: In order to transfer, the wallet must be unlocked. If the broadcast flag is `False`, the wallet will construct and sign, but **not** broadcast the transaction. This can be very useful for a cold storage setup or to verify transactions.

If *alice* wants to send *10 USD* to *bob*, she could call::

```
unlocked >> transfer alice bob 10 USD "a gift" true
```

The wallet will return the actual signed transaction.

Assets

Update/Change an existing UIA

A UIA can be modified by the issuer after creation. For this, a separated call `update_asset` has been created.

What can and cannot be changed

Except for

- Symbol
- Precision,

every parameter, option or setting can be updated.

Note: Once a permission is removed, it can not be re-enabled again!

Python Example

```
from grapheneapi import GrapheneClient
import json

perm = {}
perm["charge_market_fee"] = 0x01
perm["white_list"] = 0x02
perm["override_authority"] = 0x04
perm["transfer_restricted"] = 0x08
perm["disable_force_settle"] = 0x10
perm["global_settle"] = 0x20
perm["disable_confidential"] = 0x40
perm["witness_fed_asset"] = 0x80
perm["committee_fed_asset"] = 0x100

class Config():
    wallet_host      = "localhost"
    wallet_port      = 8092
    wallet_user      = ""
    wallet_password   = ""

if __name__ == '__main__':
    graphene = GrapheneClient(Config)

    permissions = {"charge_market_fee" : True,
                   "white_list" : True,
                   "override_authority" : True,
                   "transfer_restricted" : True,
                   "disable_force_settle" : True,
                   "global_settle" : True,
                   "disable_confidential" : True,
                   "witness_fed_asset" : True,
                   "committee_fed_asset" : True,
                   }
```

```

flags      = {"charge_market_fee" : True,
             "white_list" : True,
             "override_authority" : True,
             "transfer_restricted" : True,
             "disable_force_settle" : True,
             "global_settle" : False,
             "disable_confidential" : True,
             "witness_fed_asset" : False,
             "committee_fed_asset" : True,
             }
permissions_int = 0
for p in permissions :
    if permissions[p]:
        permissions_int += perm[p]
flags_int = 0
for p in permissions :
    if flags[p]:
        flags_int += perm[p]
options = {"max_supply" : 100000000000,
           "market_fee_percent" : 0,
           "max_market_fee" : 0,
           "issuer_permissions" : permissions_int,
           "flags" : flags_int,
           "core_exchange_rate" : {
               "base": {
                   "amount": 10,
                   "asset_id": "1.3.0"},
               "quote": {
                   "amount": 10,
                   "asset_id": "1.3.1"}},
           "whitelistAuthorities" : [],
           "blacklistAuthorities" : [],
           "whitelistMarkets" : [],
           "blacklistMarkets" : [],
           "description" : "My fancy description 2"
           }

tx = graphene.rpc.update_asset("SYMBOL", None, options, True)
print(json.dumps(tx, indent=4))

```

Publishing a Feed

A price feed operation takes the following form:

```
{
  "publisher": "1.2.0",
  "asset_id": "1.3.0",
  "feed": {
    "settlement_price": {
      "base": {
        "amount": 0,
        "asset_id": "1.3.0" },
      "quote": {
        "amount": 0,
        "asset_id": "1.3.0" }
    },
  }
```

```
"maintenance_collateral_ratio": 1750,
"maximum_short_squeeze_ratio": 1200,
"core_exchange_rate": {
    "base": {
        "amount": 0,
        "asset_id": "1.3.0" },
    "quote": {
        "amount": 0,
        "asset_id": "1.3.0" }
}
}
```

It contains the *publisher* name, the *asset_id* for which the feed has been generated the *feed* as a structure of *base* and *quote* ratio, the maintenance collateral ratio ($1750 = 175\%$), the short squeeze ratio ($1200 = 120\%$) and the core exchange rate for implicit exchange of the fee.

Python Script

```
from grapheneapi import GrapheneClient
import json

class Config():
    wallet_host          = "localhost"
    wallet_port          = 8092
    wallet_user          = ""
    wallet_password      = ""

if __name__ == '__main__':
    graphene = GrapheneClient(Config)

    price = 1.50 # one BTS costs 1.50 SYMBOLS
    asset_symbol = "SYMBOL.BIT"
    producer = "nathan"

    account = graphene.rpc.get_account(producer)
    asset = graphene.rpc.get_asset(asset_symbol)
    base = graphene.rpc.get_asset("1.3.0")
    price = price * 10 ** asset["precision"] / 10 ** base["precision"]
    denominator = base["precision"]
    numerator = round(price*base["precision"])
    price_feed = {"settlement_price": {
        "quote": {"asset_id": "1.3.0",
                  "amount": denominator
                 },
        "base": {"asset_id": asset["id"],
                  "amount": numerator
                 }
       },
       "maintenance_collateral_ratio": 1750,
       "maximum_short_squeeze_ratio": 1300,
       "core_exchange_rate": {
           "quote": {"asset_id": "1.3.0",
                     "amount": int(denominator * 1.05)
                    }
         }
    }
```

```

        },
        "base": { "asset_id": asset["id"],
                  "amount": numerator
                } })
handle = graphene.rpc.begin_builder_transaction()
op = [19, # id 19 corresponds to price feed update operation
      {"asset_id" : asset["id"],
       "feed"      : price_feed,
       "publisher" : account["id"]
      }]
graphene.rpc.add_operation_to_builder_transaction(handle, op)
graphene.rpc.set_fees_on_builder_transaction(handle, "1.3.0")
tx = graphene.rpc.sign_builder_transaction(handle, True)
print(json.dumps(tx, indent=4))

```

Prediction Markets

Committee Tutorials

Worker Tutorials

How to Create a Worker

Workers are currently created with the cli_wallet with the following command syntax::

```
>>> create_worker owner_account work_begin_date work_end_date daily_pay name url_
↪worker_settings broadcast
```

example, owner_account is creating a one day worker starting Oct 28 and will get paid 1 BTS/day (vesting in 1 day, 1 BTS = 100,000 ‘satoshi’) to make an android app. The first command won’t broadcast, this will just check::

```
>>> create_worker "worker-name" "2015-10-28T00:00:00" "2015-10-29T00:00:00" 100000
↪"Description" "http://URL" {"type" : "vesting", "pay_vesting_period_days" : 1} false
```

The URL should point to something describing your proposal. We recommend to answer the following questions:

- What will you do with the funds?
- By when will you be done?
- For how much?

The variable type can be

- refund to return your pay back to the pool to be used for future projects,
- vesting to pay that you pay yourself, or
- burn to destroys your pay thus reducing share supply, equivalent to share buy-back of a company stock

The variable pay_vesting_period_days is the integer number of days you set for vesting. Some people don’t want workers to withdraw and sell large sums of BTS immediately, as it puts sell pressure on BTS. Also, if you require vesting, you have “skin in the game” and thus an incentive to improve BTS value. Pay is pay per day (not hour or maintenance period) and is in units of 1/100,000 BTS (the precision of BTS)

To actually generate a worker proposal, replace the last false by true.

How to see proposals on the chain

Since there is no support in the UI yet, go to <http://cryptofresh.com/> and look at the worker proposal chart. You also can inspect all the objects 1.4.*::

```
>>> get_object 1.14.4
get_object 1.14.4
[{
    "id": "1.14.4",
    "worker_account": "1.2.22517",
    "work_begin_date": "2015-10-21T11:00:00",
    "work_end_date": "2015-11-21T11:00:00",
    "daily_pay": 1000000000,
    "worker": [
        1, {
            "balance": "1.13.235"
        }
    ],
    "vote_for": "2:73",
    "vote_against": "2:74",
    "total_votes_for": "14632377015617",
    "total_votes_against": 0,
    "name": "bitasset-fund-pool",
    "url": "https://bitsharestalk.org/index.php/topic,19317.0.html"
}]
]
```

How to Vote for a Worker

Currently the GUI doesn't have an interface, but you can vote using the CLI::

```
>>> update_worker_votes your-account {"vote_for": ["proposal-id"]} true
```

for example::

```
>>> update_worker_votes "awesomedbitsharer" {"vote_for": ["1.4.0"]} true
```

you can also vote against or abstain (remove your vote for or against):

```
>>> update_worker_votes your-account {"vote_against": ["proposal-id"]} true
>>> update_worker_votes your-account {"vote_abstain": ["proposal-id"]} true
```

How Workers Get Paid

Every hour the worker budget is processed and workers are paid in full order of the number of votes for minus the number of votes against. The last worker to get paid will be paid with whatever is left, so may receive partial payment. The daily budget can be estimated by inspecting the most recent budget object 2.13.* for example::

```
>>> get_object 2.13.361
get_object 2.13.361
[{
    "id": "2.13.361",
    "time": "2015-10-28T15:00:00",
    "record": {
```

```

    "time_since_last_budget": 3600,
    "from_initial_reserve": "106736452914941",
    "from_accumulated_fees": 15824269,
    "from_unused_witness_budget": 2250000,
    "requested_witness_budget": 180000000,
    "total_budget": 1520913100,
    "witness_budget": 180000000,
    "worker_budget": 1340913100,
    "leftover_worker_funds": 0,
    "supply_delta": 1502838831
}
]

```

So the daily budget is:

```
worker_budget*24 = 1340913100 * 24 = 32181914400 (or 321,8191.44 BTS)
```

There is currently a maximum daily worker pay of 500k BTS, and this can be found using the `get_global_properties` command in the `cli_wallet`

Technical Details

Every second,

```
[ 17 / (2^32) * reserve fund ]
```

is allocated for witnesses and workers. The reserve fund is maximum number of BTS available less those currently in circulation ([source](#))

Every hour the total available reserve fund is calculated by finding how many BTS are available to be distributed and how many BTS will be returned to the reserve fund (i.e., “burnt”) during the next maintenance interval.

First find how many BTS have not been distributed::

```
>>> from_initial_reserve = max_supply - current supply of BTS
```

The `max_supply` can be obtained by:

```
>>> get_object 1.3.0
```

and the `current_supply` is given in:

```
>>> get_object 2.3.0
```

Modify it by adding the accumulated fees and witness budget remaining (i.e., 1.5 BTS per block is budgeted, so budget remaining is 1.5 BTS * (number of blocks left in maintenance period+blocks missed by witnesses)) in this maintenance cycle (they will be added to the “reserve fund” permanently at maintenance):

```
updated reserve fund = from_initial_reserve + from_accumulated_fees + from_unused_
↪witness_budget
```

variables all from: `get_object 2.13.*` (choose the most recent one, for example)

Next calculate how much is available to be spent on workers and witnesses is::

```
total_budget = (updated reserve fund) * (time_since_last_budget) * 17 / (2^32)
```

rounded up to the nearest integer

Ok, now to find how much workers will get in this budget period (1 hour), you find the smaller of the available pay AFTER subtracting witness budget from the total_budget OR the worker_budget_per_day/24 from get_global_properties:

```
worker_budget=min( total_budget - witness_budget , worker_budget_per_day / 24 )
```

That is how much per hour allocated for all workers. NOW you rank each worker and pay them one hours worth of pay in order or # votes.

Claim Worker Pay

Every second, $[17/(2^{32}) * \text{reserve fund}]$ is allocated for witnesses and workers where reserve fund is how many BTS are currently not distributed (see the [source code](#)).

Every hour the total available reserve fund is calculated by finding how many BTS are available to be distributed and how many BTS will be returned to the reserve fund (i.e., “burnt”) during the next maintenance interval.

First find how many BTS have not been distributed:

```
# get max_supply from
get_object 1.3.0
# get current_supply from
get_object 2.3.0

==> from_initial_reserve = max_supply - current supply of BTS
```

Then modify it by adding the accumulated fees and witness budget remaining (i.e., 1.5 BTS per block is budgeted, so budget remaining is

```
1.5 BTS * (number of blocks left in maintenance period+blocks missed by witnesses))
```

in this maintenance cycle (they will be added to the “reserve fund” permanently at maintenance)

```
get_object 2.13.361

==> updated reserve fund = from_initial_reserve + from_accumulated_fees + from_
    ↵ unused_witness_budget
```

For example:

```
>>> get_object 2.13.361
get_object 2.13.361
[{
    "id": "2.13.361",
    "time": "2015-10-28T15:00:00",
    "record": {
        "time_since_last_budget": 3600,
        "from_initial_reserve": "106736452914941",
        "from_accumulated_fees": 15824269,
        "from_unused_witness_budget": 2250000,
        "requested_witness_budget": 180000000,
        "total_budget": 1520913100,
        "witness_budget": 180000000,
```

```

        "worker_budget": 1340913100,
        "leftover_worker_funds": 0,
        "supply_delta": 1502838831
    }
}
]
```

Then calculate how much is available to be spent on workers and witnesses is:

```
total_budget = (updated reserve fund) * (time_since_last_budget) * 17 / (2^32) #rounded up
→to the nearest integer
```

Ok, now to find how much workers will get in this budget period (1 hour), you find the smaller of the available pay AFTER subtracting witness budget from the total_budget OR the worker_budget_per_day/24 from “get_global_properties”

```
worker_budget=min(total_budget-witness_budget,worker_budget_per_day/24)
```

That is how much per hour allocated for all workers. NOW you rank each worker and pay them one hours worth of pay in order or # votes.

References:

https://github.com/cryptonomex/graphene/blob/4c09d6b8ed350ff5c7546e2c3fd15d0e6699daf2/libraries/chain/db_main.cpp

Witness Tutorials

Howto Run a Block-producing Witness

This document serves as an introduction on how to become an actively block producing witness in a Graphene-based network (e.g. the BitShares2.0 network).

We will have to register a new account from the and add some initial funds for the witness registration fee. After that, we will create, configure and run a witness node.

Requirements

- A registered account in the corresponding network (see i.e. *Accounts*)
- Some funds in the account to pay for the registration fee
- Executable binary (see *Installation*)

Hardware Advice

- Dedicated servers with minimum 16 GB (32 GB advised) Ram, SSD disks advised.

Active Witness Duties

- Be a reliable blockproducer
- Maintain a public seednode
- Publish accurate, frequently updated (check 1-2 times per hour), price feeds for the Smart Coins & Market Pegged Assets

Overview

We will now perform the following steps:

- run a local (non block producing) full node
- create a CLI wallet for the network
- import your account (and funds) into CLI wallet
- upgrade our account to a lifetime member
- register a new witness
- upvote the witness with our funds
- sign blocks

Run the witness as a node in the network

We first run the witness node without block production and connect it to the P2P network with the following command::

```
$ programs/witness_node/witness_node --rpc-endpoint 127.0.0.1:8090
```

We open a RPC port for local host so that we can later connect the CLI wallet with it. After the network was synced and periodically receives new blocks from other participants, we can go on to the next step.

Creating a wallet

We now open up the cli_wallet and connect to our plain and stupid witness node::

```
$ programs/cli_wallet/cli_wallet -s ws://127.0.0.1:8090
```

First thing to do is setting up a password for the newly created wallet prior to importing any private keys::

```
>>> set_password <password>
null
>>> unlock <password>
null
>>>
```

Wallet creation is now done.

Basic Account Management

We can import the account name (owner key) and the balance containing (active) key into the CLI wallet::

```
>>> import_key <accountname> <owner wif key>
true
>>> import_key <accountname> <active wif key>
true
>>> list_my_accounts
[{
  "id": "1.2.15",
  ...
  "name": <accountname>,
  ...
}]
>>> list_account_balances <accountname>
XXXXXXXXX BTS
```

Both keys can be exported from the web wallet.

Since only lifetime members can become witnesses, you must first upgrade to a lifetime member. This step costs the lifetime-upgrade fee::

```
>>> upgrade_account <accountname> true  
[a transaction in json format]
```

Registering a Witness Object

To become a witness and be able to produce blocks, you first need to create a witness object that can be voted in.

We create a new witness object by issuing::

Our witness is registered, but it can't produce blocks because nobody has voted it in. You can see the current list of active witnesses with `get_global_properties`:

```
>>> get_global_properties
{
    "active_witnesses": [
        "1.6.0",
        "1.6.1",
        "1.6.2",
        "1.6.3",
        "1.6.4",
        "1.6.5",
        "1.6.7",
        "1.6.8",
        "1.6.9"
    ],
    ...
}
```

Now, we should vote our witness in. Vote all of the shares your account <accountname> in favor of your new witness.:

```
>>> vote_for_witness <accountname> <accountname> true true
[a transaction in json format]
```

Note: If you want to experiment with things that require voting, be aware that votes are only tallied once per day at the maintenance interval. `get_dynamic_global_properties` tells us when that will be in `next_maintenance_time`. Once the next maintenance interval passes, run `get_global_properties` again and you should see that your new witness has been voted in.

Now we wait until the next maintenance interval.

Configuration of the Witness Node

Get the witness object using:

```
get_witness <witness-account>
```

and take note of two things. The `id` is displayed in `get_global_properties` when the witness is voted in, and we will need it on the `witness_node` command line to produce blocks. We'll also need the public `signing_key` so we can look up the corresponding private key.

Once we have that, run `dump_private_keys` which lists the public-key private-key pairs to find the private key.

Warning: `dump_private_keys` will display your keys unencrypted on the terminal, don't do this with someone looking over your shoulder.

```
>>> get_witness <accountname>
{
    [...]
    "id": "1.6.10",
    "signing_key": "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
    [...]
}
```

The `id` and the `signing_key` are the two important parameters, here. Let's get the private key for that signing key with::

```
>>> dump_private_keys
[ [
  ...
  ],
  [
    "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
    "5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"
  ]
]
```

Now we need to start the witness, so shut down the wallet (ctrl-d), and shut down the witness (ctrl-c). Re-launch the witness, now mentioning the new witness 1.6.10 and its keypair::

```
./witness_node --rpc-endpoint=127.0.0.1:8090 \
  --witness-id '"1.6.10"' \
  --private-key '["GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8
  ↪", "5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"]'
```

Alternatively, you can also add this line into your config.ini::

```
witness-id = "1.6.10"
private-key = ["GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
  ↪"5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"]
```

Note: Make sure to use YOUR public/private keys instead of the ones given above!

Verifying Block Production

If you monitor the output of the `witness_node` and you have been voted in the top list of block producing witnesses, you should see it generate blocks signed by your witness::

```
Witness 1.6.10 production slot has arrived; generating a block now...
Generated block #367 with timestamp 2015-07-05T20:46:30 at time 2015-07-05T20:46:30
```

Backup Server

To stay a reliable block producer it is recommended you have a ‘hot swappable’ backup server with same specs as the live server running an instance of `witness_node`. ***IT IS IMPORTANT THAT THIS BACKUP SHOULD NOT HAVE THE SAME SIGNING KEY PAIR in the config.ini as your main node!***

How it works:

1. Your ‘live’ witness node is signing blocks with the private key which is stated in the config.ini.
2. Your ‘backup’ witness node is running a copy of the software with another private key in the config.ini (generate a new public/private keypair with `cli_wallet` command: `suggest_brain_key`).
3. On a third server you monitor your ‘live’ node on regular intervals with an automated script (e.g.: <https://github.com/roelandp/Bitshares-Witness-Monitor>).
4. As soon as your ‘live’ node is starting to fail producing blocks the ‘missing blocks’ parameter increases and you can issue a command to `update_witness` to your backup’s ‘Public Key’.

5. Investigate the issue with your ‘live’ node and stay happy.

Price Feeds

Besides producing new blocks another very important task of the witness is to feed **ACCURATE** prices into the blockchain. Educational material on how this can be implemented is available in `scripts/pricefeed` at [github](#) together with the corresponding documentation.

Only active witnesses are allowed to publish pricefeeds for the ‘official’ currency / smartcoin markets. For any other asset you need to be whitelisted by the creator of the market to be able to publish pricefeeds.

A couple of price feed scripts in various stages of development and for you to code-inspect and try. (You can always setup a testnet node and test the publishing of pricefeeds).

- Wackou’s BTS tools includes a pricefeed publishing script: https://github.com/wackou/bts_tools
- Alt’s BTS Price: <https://github.com/pch957/btspice>
- Xeroc’s Bitshares Pricefeed: <https://github.com/xeroc/bitshares-pricefeed>

Howto Become an Active Witness

This document serves as an introduction on how to become an actively block producing witness in the BitShares2.0 network. We will create, configure and run a witness node in the following steps:

- create a wallet for the testnet
- import an account and funds
- upgrade our account to a lifetime member
- register a new witness
- upvote the witness with our funds
- sign blocks

Run the witness/full node on the network

We first run the *Full Node* without block production and connect it to the P2P network with the following command:

```
$ programs/witness_node/witness_node --rpc-endpoint 127.0.0.1:8090
```

The argument `--rpc-endpoint 127.0.0.1:8090` opens up a RPC port 8090 for connections from localhost.

Creating a wallet

We now open up the *CLI Wallet* and connect to our plain and stupid *Full Node*:

```
programs/cli_wallet/cli_wallet -s ws://127.0.0.1:8090
```

First thing to do is setting up a password for the newly created wallet prior to importing any private keys:

```
>>> set_password <password>
null
>>> unlock <password>
null
```

Wallet creation is now done.

Basic Account Management

We can import the account name (owner and active keys) to be able to access our funds in BitShares 2.0:

```
>>> import_key <accountname> <owner wif key>
true
>>> import_key <accountname> <active wif key>
true
>>> list_account_balances <accountname>
XXXXXXXXX BTS
```

Since **only lifetime members can become witnesses**, you must first upgrade to a lifetime member. This step costs the lifetime-upgrade fee which will eventually cost about \$100.

```
>>> upgrade_account <accountname> true  
[a transaction in json format]
```

Becoming a Witness

To become a witness and be able to produce blocks, you first need to create a witness object that can be voted in.

Note: If you want to experiment with things that require voting, be aware that votes are only tallied once per day at the maintenance interval. `get_dynamic_global_properties` tells us when that will be in `next_maintenance_time`. Once the next maintenance interval passes, run `get_global_properties` again and you should see that your new witness has been voted in.

Before we get started, we can see the current list of witnesses voted in, which will simply be the ten default witnesses:

Our witness is registered, but it can't produce blocks because nobody has voted it in. You can see the current list of active witnesses with `get_global_properties`.

Now, we should vote our witness in. Vote all of the shares in our account `<accountname>` in favor of your new witness.

```
>>> vote_for_witness <accountname> <accountname> true true  
[a transaction in json format]
```

We need wait until the next maintenance interval until we can see votes casted for our witness.

Get the witness object using `get_witness` and take note of two things. The `id` is displayed in `get_global_properties` when the witness is voted in, and we will need it on the `witness_node` command line to produce blocks. We'll also need the public `signing_key` so we can look up the corresponding private key.

```
>>> get_witness <accountname>  
{  
    [...]  
    "id": "1.6.10",  
    "signing_key": "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",  
    [...]  
}
```

Once we have that, run `dump_private_keys` which lists the public-key private-key pairs to find the private key.

Warning: `dump_private_keys` will display your keys unencrypted on the terminal, don't do this with someone looking over your shoulder.

The `id` and the `signing_key` are the two important parameters, here. Let's get the private key for that signing key with:

```
>>> dump_private_keys  
[[  
    ...  
], [  
    "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",  
    "5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"  
]
```

Now we need to start the witness, so shut down the wallet (ctrl-d), and shut down the witness (ctrl-c). Re-launch the witness, now mentioning the new witness 1.6.10 and its keypair:

```
./witness_node \  
    --rpc-endpoint=127.0.0.1:8090 \  
    --witness-id '"1.6.10"' \  
    --private-key '[ "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",  
    ↪ "5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ" ]'
```

Alternatively, you can also add this line into your config.ini:

```
witness-id = "1.6.10"  
private-key = [ "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",  
    ↪ "5JGi7DM7J8fSTizz4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ" ]
```

Note: Make sure to use YOUR public/private keys instead of the ones given above!

If you monitor the output of the `witness_node`, you should see it generate blocks signed by your witness:

```
Witness 1.6.10 production slot has arrived; generating a block now...
Generated block #367 with timestamp 2015-07-05T20:46:30 at time 2015-07-05T20:46:30
```

Change the Signing Key of your Witness

As a witness you may want to change your key occasionally or on a regular basis. You can do so by calling `update_witness`:

```
>>> update_witness <witness-name> <url> <new-publickey> false
```

By replacing `false` with `true`, the signed transaction will be broadcast and executed otherwise it will only print the signed transaction for review.

You can get a new public key by calling:

```
>>> suggest_brain_key
```

Developers

Distributed Access to the BitShares Decentralised Exchange

I hope to encourage and promote more access points and backup WebSocket (wss) gateways for BitShares. This is the logical progression from [Run your own decentralised exchange](#) post. ###Distributed Access to the BitShares Network

BitShares node setup

[Run your own decentralised exchange](#)

Once you have a full node setup, you can allow BitShares shareholders secure access to your server to trade and check their accounts by following these steps. >A DNS Alias (CNAME) is required to point to your server ip address. >See [dyn.com](#) for DNS Alias setup. >You may have to wait a few days for the DNS to work through the internet. >Please change [altcap.io](#) to your DNS alias in the examples below.

Table of Contents

[TOC]

Create a New User

I recommend creating a new user on your server to run the Bitshares gui and give the user sudo access. >You can use any name - I have used bitshares in this example

```
sudo adduser bitshares
sudo gpasswd -a bitshares sudo
sudo gpasswd -a bitshares users
```

Install Nginx

Install Nginx web server

```
# ssh into your new user bitshares
ssh bitshares@altcap.io
sudo apt-get install nginx
# check version
nginx -v
# add user to web server group
sudo gpasswd -a bitshares www-data
# start nginx
sudo service nginx start
```

This will start Nginx default web server. Check it by typing the ip address of your server in a web browser or your alias `altcap.io`.

Configure Nginx

To configure the web server, edit the default site and save as new DNS alias name using http port 80 only until you setup letsencrypt.org SSL Certificate.

Create your web folder

```
sudo mkdir -p /var/www/altcap.io/public_html
sudo chown -R bitshares:bitshares var/www/altcap.io/public_html
sudo chmod 755 /var/www
```

Configure Nginx

```
# edit default setup and save as altcap.io
sudo nano /etc/nginx/sites-available/default
```

Point to your new virtual host

```
##### altcap.io #####
server {
    listen 80;
    server_name altcap.io;
    #rewrite ^ https://$server_name$request_uri? permanent;
    #rewrite ^ https://altcap.io$uri permanent;
    #
    root /var/www/altcap.io/public_html;
    # Add index.php to the list if you are using PHP
    index index.html index.htm;
    #
    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }
}

CTRL+O to save as altcap.io (^O Write Out)
```

Update Virtual Host File

```
sudo cp altcap.io /etc/nginx/sites-available/altcap.io
```

Activate sim link and disable default web server

```
sudo ln -s /etc/nginx/sites-available/altcap.io /etc/nginx/sites-enabled/altcap.io
sudo rm /etc/nginx/sites-enabled/default
```

Link local folder to www root and add a simple index.html

```
ln -s /var/www/altcap.io/public_html ~/public_html
nano ~/public_html/index.html
```

Add some text to index.html

```
<html>
  <head>
    <title>altcap.io</title>
  </head>
  <body>
    <h1>altcap.io - Virtual Host</h1>
  </body>
</html>

CTRL+X to save as index.html (^X Exit) ## Restart Nginx
```

```
sudo service nginx restart
```

Now you have setup a simple web server. DigitalOcean has a great [article](#) for more information on Virtual Host setup.

Install letsencrypt

```
sudo apt-get install letsencrypt
```

Obtain your SSL certificate

```
sudo letsencrypt certonly --webroot -w /var/www/altcap.io/public_html -d altcap.io
```

Follow the instructions and add an email address

Check your certificate

```
sudo ls -l /etc/letsencrypt/live/altcap.io
# and check it will update
sudo letsencrypt renew --dry-run --agree-tos
sudo letsencrypt renew
```

Setup a renew cronjob for your new SSL certificate

```
sudo crontab -e
```

Add this line to run the job every 6 hours on the 16th minute

```
16 */6 * * * /usr/bin/letsencrypt renew >> /var/log/letsencrypt-renew.log  
CTRL+X to save (^X Exit)
```

```
# check your crontab  
sudo crontab -l
```

Generate Strong Diffie-Hellman Group cert

```
sudo openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```

Add SSL to Nginx settings

Make a copy of altcap.io just in case.

```
cp altcap.io altcap.io.no.ssl
```

Edit altcap.io

```
nano altcap.io
```

```
##### altcap.io #####
server {
    listen 80;
    server_name altcap.io;
    #rewrite ^ https://$server_name$request_uri? permanent;
    rewrite ^ https://altcap.io$uri permanent;
    #
    root /var/www/altcap.io/public_html;
    # Add index.php to the list if you are using PHP
    index index.html index.htm;
    #
    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }
}

#####
 altcap.io websockets

upstream websockets {
    server localhost:8090;
```

```

}

#####
# altcap.io ssl
server {
    listen 443 ssl;
    #
    server_name altcap.io;
    #
    root /var/www/altcap.io/public_html;
    # Add index.php to the list if you are using PHP
    index index.html index.htm;
    #
    ssl_certificate /etc/letsencrypt/live/altcap.io/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/altcap.io/privkey.pem;
    #
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_prefer_server_ciphers on;
    ssl_dhparam /etc/ssl/certs/dhparam.pem;
    ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
    ↵AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-
    ↵AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
    ↵SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-
    ↵ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-
    ↵SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-
    ↵AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-
    ↵SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!
    ↵EXPORT:!DES:!RC4:!MD5:!PSK!:aECDH!:EDH-DSS-DES-CBC3-SHA!:EDH-RSA-DES-CBC3-SHA!:KRB5-
    ↵DES-CBC3-SHA';
    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:50m;
    ssl_stapling on;
    ssl_stapling_verify on;
    add_header Strict-Transport-Security max-age=15768000;
    #
    # Note: You should disable gzip for SSL traffic.
    # See: https://bugs.debian.org/773332
    #
    # Read up on ssl_ciphers to ensure a secure configuration.
    # See: https://bugs.debian.org/765782
    #
    # Self signed certs generated by the ssl-cert package
    # Don't use them in a production server!
    #
    # include snippets/snakeoil.conf;
    #
    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }
    location ~ /ws/? {
        access_log off;
        proxy_pass http://websockets;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_http_version 1.1;
    }
}

```

```
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
}

#####
# altcap.io #####
CTRL+X to save (^X Exit)
```

You have now setup an SSL secured web server with a WebSocket connected to your local BitShares witness_node (listening on port 8090 - see [this post](#) for more information) ###Update altcap.io www virtual host

```
sudo cp altcap.io /etc/nginx/sites-available/altcap.io
```

Restart Nginx

```
sudo service nginx restart
```

Now you have setup an SSL web server. More information on SSL setup can be found here. [DigitalOcean letsencrypt SSL LetsEncrypt CertBot](#)

Install BitShares web gui

Install NVM (Node Version Manager)

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.30.2/install.sh | bash
```

exit bash (terminal) and reconnect

```
ssh bitshares@altcap.io
nvm install v5
nvm use v5
```

Download BitShares gui

```
git clone https://github.com/bitshares/bitshares-2-ui.git
```

Setup light wallet

```
cd /bitshares-2-ui/
```

Before building the light wallet, you need to edit **SettingsStore.js** line 19 and 99 wss WebSocket.

```
nano /bitshares-2-ui/dl/src/stores/SettingsStore.js
```

Change line 19

```
connection: "wss://altcap.io/ws",
```

Add your new wss WebSocket to line 99

```
connection: [
    "wss://altcap.io/ws",
    "wss://bitshares.openledger.info/ws",
    "wss://bitshares.dacplay.org:8089/ws",
    "wss://dele-puppy.com/ws",
    "wss://valen-tin.fr:8090/ws"
```

CTRL+X to save (^X Exit)

```
###Setup install
cd dl; npm install
cd ../web; npm install
```

Link web root to gui build folder

```
ln -s ~/public_html/ dist
```

Build light wallet

```
npm run build
```

You have now created another Access point to the BitShares Decentralised Exchange. **The more the merrier.** Please remember to check your firewall and SSH is up-to-date and configured correctly. DigitalOcean has [firewall](#) and [Secure SSH](#) tutorials for more help.

SSL test

You can also check how secure your new web server is compared to your bank. Add this link to a web browser and wait for the results.

```
https://www.ssllabs.com/ssltest/analyze.html?d=altcap.io
```

Now change altcap.io to your local bank's domain name in the link and post the results below. >**Thank you 'svk <<https://steemit.com/@svk>>' for your advice and guidance.**

MUSE is a public ledger specifically tailored for the Music Industry that keeps track of worldwide copyrights. It is an ownerless, automated, globally distributed, Peer-to-Peer network that is both transparent and open to all.

2.2 MUSE

2.2.1 Installation

This section describes the installation procedure and the build process for those that want to compile from the source.

Downloads

Precompile Executables

Compiled executables are available for [download from github](#).

Sources

The sources are located at [github](#) and can be downloaded with *git*.

```
git clone https://github.com/peertracksinc/muse
```

Since the repository makes use of so called *submodules* which are repositories on their own, we need to refresh those.

```
git submodule update --init --recursive
```

Building from Sources

Downloading the sources

The sources can be downloaded from github as described [here](#).

Dependencies

Development Toolkit

The following dependencies were necessary for a clean install of Ubuntu 14.04 LTS:

```
sudo apt-get install gcc-4.9 g++-4.9 cmake make \
    libbz2-dev libdbd++-dev libdbd-dev \
    libssl-dev openssl libreadline-dev \
    autoconf libtool git
```

Boost 1.57

The Boost which ships with Ubuntu 15.04 is too old. You need to download the Boost tarball for Boost 1.57.0 (Note, 1.58.0 requires C++14 and will not build on Ubuntu LTS; this requirement was an accident, see [this mailing list post](#)).

```
BOOST_ROOT=$HOME/opt/boost_1_57_0
sudo apt-get update
sudo apt-get install autotools-dev build-essential \
    g++ libbz2-dev libicu-dev python-dev
wget -c 'http://sourceforge.net/projects/boost/files/boost/1.57.0/boost_1_57_0.tar.
↳bz2/download'\ \
    -O boost_1_57_0.tar.bz2
sha256sum boost_1_57_0.tar.bz2
# "910c8c022a33ccecf7f088bd65d4f14b466588dda94ba2124e78b8c57db264967"
tar xjf boost_1_57_0.tar.bz2
cd boost_1_57_0/
./bootstrap.sh "--prefix=$BOOST_ROOT"
./b2 install
```

QT 5.5

Qt 5.5 is a dependency if you wish to build *light_client*. Building it from source requires the following dependencies on Ubuntu LTS:

```
sudo apt-get install libxcb1 libxcb1-dev libx11-xcb1 libx11-xcb-dev \
    libxcb-keysyms1 libxcb-keysyms1-dev libxcb-image0 libxcb-image0-
    dev \
    libxcb-shm0 libxcb-shm0-dev libxcb-icccm4 libxcb-icccm4-dev \
    libxcb-sync1 \
    libxcb-sync-dev
```

Qt 5.5 can be built as follows:

```
QT_ROOT=$HOME/opt/qt5.5.0

wget http://download.qt.io/official_releases/qt/5.5/5.5.0/single/qt-everywhere-
    opensource-src-5.5.0.tar.gz
sha256sum qt-everywhere-opensource-src-5.5.0.tar.gz
# "bf3cfc54696fe7d77f2cf33ade46c2cc28841389e22a72f77bae606622998e82"
tar xzf qt-everywhere-opensource-src-5.5.0.tar.gz
cd qt-everywhere-opensource-src-5.5.0
./configure -prefix "$QT_ROOT" -opensource -nomake tests
make # -j4
make install
```

Next we need to tell *cmake* where to find them. If you have ever run CMake in this tree before, we must first delete some leftovers:

```
make clean
rm -f CMakeCache.txt
find . -name CMakeFiles | xargs rm -Rf
```

To actually run *cmake* we now need the following parameters:

```
cmake -DCMAKE_PREFIX_PATH="$QT_ROOT" \
    -DCMAKE_MODULE_PATH="$QT_ROOT/lib/cmake/Qt5Core" \
    -DQT_QMAKE_EXECUTABLE="$QT_ROOT/bin/qmake" -DBUILD_QT_GUI=TRUE \
    -DGRAPHENE_EGENESIS_JSON="$GENESIS_JSON" \
    -DBOOST_ROOT="$BOOST_ROOT" -DCMAKE_BUILD_TYPE=Debug .
cd ..
```

Building MUSE

After downloading the muse sources according to [the download page](#), we can run *cmake* for configuration and compile with *make*:

```
cmake -DBOOST_ROOT="$BOOST_ROOT" -DCMAKE_BUILD_TYPE=Release .
make
```

Note that the environmental variable `$BOOST_ROOT` should point to your install directory of boost if you have installed it manually.

Distribution Specific Settings

Ubuntu 14.04

As g++-4.9 isn't available in 14.04 LTS, you need to do this first:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
```

If you get build failures due to abi incompatibilities, just use gcc 4.9

```
CC=gcc-4.9 CXX=g++-4.9 cmake .
```

Ubuntu 15.04

Ubuntu 15.04 uses gcc 5, which has the c++11 ABI as default, but the boost libraries were compiled with the cxx11 ABI (this is an issue in many distros). If you get build failures due to abi incompatibilities, just use gcc 4.9:

```
CC=gcc-4.9 CXX=g++-4.9 cmake .
```

Upgrading

Recompiling from Sources

For upgrading from source you only need to execute:

```
git fetch
git checkout <version>
git submodule update --init --recursive
cmake .
make
```

2.2.2 Howto Redeem your MUSE/NOTE

This migration tutorial is relevant only to those customers and investors that have participated in the MUSE pre-sale. We here give assistance for claiming your funds in MUSE.

Note: The *NOTE blockchain* has been renamed to **MUSE blockchain** and the corresponding blockchain token is now called **MUSE**.

Exporting your wallet

In this tutorial, we assist you to extract the required data to later redeem your MUSE token. We distinguish between users that have participated in the Pre-Sale and those that have bought their tokens on the BitShares 1.0 decentralized exchange.

From Bitcoin Pre-Sale

Those that participated in the NOTES pre-sale using Bitcoin can obtain the required information to redeem their corresponding amount of MUSE (rebranding of NOTE) as described in the following guide:

- [Read more ...](#)

From BitShares Decentralized Exchange

If you have bought or traded the NOTE token within the BitShares DEX, you can obtain the required information as described in the following guide:

- [Read more ...](#)

Bitcoin Pre-Sale

If you have bought MUSE/NOTE token from the pre-sale paying bitcoin to the address:

37X8DHpfiimB7PU5y35rfBcg5Vxj2R6umL

all you need to get access to your MUSE/NOTE token is the *private key* associated with the address that was used to send the bitcoin to that address above. We will here assist you in how to obtain the correct private key in the correct format from the bitcoin wallet you used.

Unless you know which Bitcoin address you have used specifically to participate in the pre-sale, we need to identify it in order to extract the correct private key. If you already know the used address, you can skip this section and go over to actually *exporting* the corresponding private key.

1. Identify your pre-sale transaction in the bitcoin client you used to participate in the presale
2. Copy&Paste the *transaction id*
3. Go to either the a block explorer of your choice and display the details for your transaction id
4. Copy and paste the **first** address used to pay for the transaction
5. Then export the private key of that address using your bitcoin client

Step-By-Step

1. Search for your transaction id using <http://btc.blockr.io>
2. Click on the **first** address on the right hand-side

The screenshot shows the blockr.io website interface. At the top, there's a navigation bar with links for API, APPS, CHARTS, TRIVIA, DOCS, and BOOKMARKS. On the left, a sidebar features various icons for different blockchain networks like Bitcoin, Ethereum, Ripple, and others. The main content area is titled "Transaction". Below the title, it shows the URL "Home / Block: 332904 / Tx: 790f5d6da440573560dfdd5be907a7f3fd7187df1cf3c00e2843b430730001af". To the right, there's a "Share" button with links to Twitter, Facebook, Google+, and BitShares. The transaction details table includes fields for Hash (790f5d6da440573560dfdd5be907a7f3fd7187df1cf3c00e2843b430730001af), Time (2014-12-05 00:40:53), Sum of incoming txs (0.15080799 BTC), and Sum of outgoing txs (0.15070799 BTC). To the right of the table, a transaction history section shows inputs and outputs. One input (1DtVCmu465TGfAAM1WgqqaFVd2VTApBzM4) is circled in orange. The transaction history table has columns for Address, Amount, and Fee. The total transaction sum is listed as 0.0000807... BTC.

Hash	790f5d6da440573560dfdd5be907a7f3fd7187df1cf3c00e2843b430730001af
Time	2014-12-05 00:40:53
Sum of incoming txs	0.15080799 BTC
Sum of outgoing txs	0.15070799 BTC

Address	Amount	Fee
1DtVCmu465TGfAAM1WgqqaFVd2VTApBzM4	-0.15	0.0000807...
1ND558dL4hLnH5CJNCoAVevf3YQuPqu...	-0.0000807...	0.0000000...
1LdkY2ex5wnxmDnzDhHxRUoIMhoH...	0.0000000...	0.0000000...

3. Copy&Paste the full address

The screenshot shows the blockr.io website interface, similar to the previous one but for a specific Bitcoin address. The main content area is titled "Bitcoin Address". Below the title, it shows the URL "Home / Address: 1DtVCmu465TGfAAM1WgqqaFVd2VTApBzM4". To the right, there's a "Share" button with links to Twitter, Facebook, Google+, and BitShares. The address details table includes fields for Hash (1DtVCmu465TGfAAM1WgqqaFVd2VTApBzM4), Balance (0.00000000 BTC), Total received (0.15000000 BTC), and Transactions (2). To the right, there's a tab navigation for "Main info", "Transactions", and "Transaction tree". Under "Main info", it shows the "First Transaction" (Time: 2014-12-05 00:40:53, Block: 332904) and the "Last Transaction" (Time: 2014-12-05 00:40:53, Block: 332904).

Hash	1DtVCmu465TGfAAM1WgqqaFVd2VTApBzM4
Balance	0.00000000 BTC
Total received	0.15000000 BTC
Transactions	2

Main info	Transactions	Transaction tree
First Transaction		Last Transaction
Time 2014-12-05 00:40:53		Time 2014-12-05 00:40:53
Block 332904		Block 332904

Now we are ready to export the private key(s) from your bitcoin wallet:

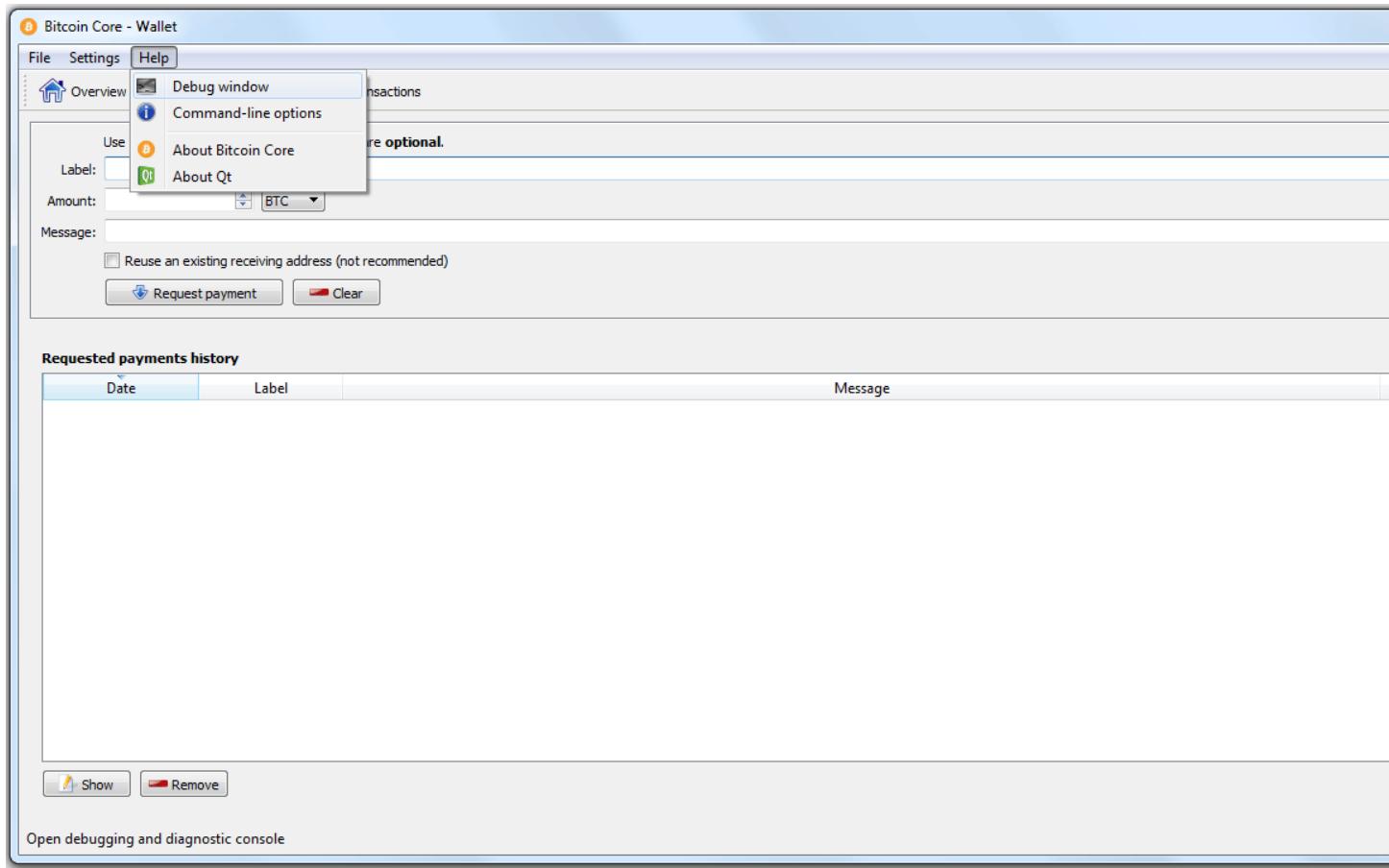
Exporting Private Key from Bitcoin Clients

The private keys required are in the so called *wallet import format* (wif), they usually start with a 5.

Here is, how you export your private keys in the most common bitcoin clients/wallets:

Bitcoin-QT

For Bitcoin-qt, we first need to access the *console* via the menu bar:



After that we can unlock the wallet with the passphrase and extract the private key with:

```
wallet passphrase <passphrase> 9999
dumpprivatekey <bitcoinaddress>
```

Blockchain.info

The advanced settings of blockchain.info offer to export an unencrypted version of the private key:

The screenshot shows the Blockchain.com wallet interface. At the top, there's a navigation bar with links for Home, Charts, Stats, Markets, API, and Wallet. Below this, the main title "My Wallet" is followed by the tagline "Be Your Own Bank." and a balance of "0.00". A horizontal menu bar includes "Wallet Home", "My Transactions", "Send Money", "Receive Money", and "Import / Export", with "Import / Export" being the active tab and circled in orange. On the left, a sidebar titled "IMPORT / EXPORT" lists "Import", "Import Backup", "Import Wallet", "Export Encrypted", "Export Unencrypted" (which is circled in blue), and "Paper Wallet". The main content area is titled "Export Unencrypted" and contains a warning about the unencrypted nature of the private keys. It features a dropdown menu set to "Bitcoin-Qt Format" (also circled in orange). Below the dropdown is a JSON-formatted string representing the private key data, with the "priv" field highlighted in green.

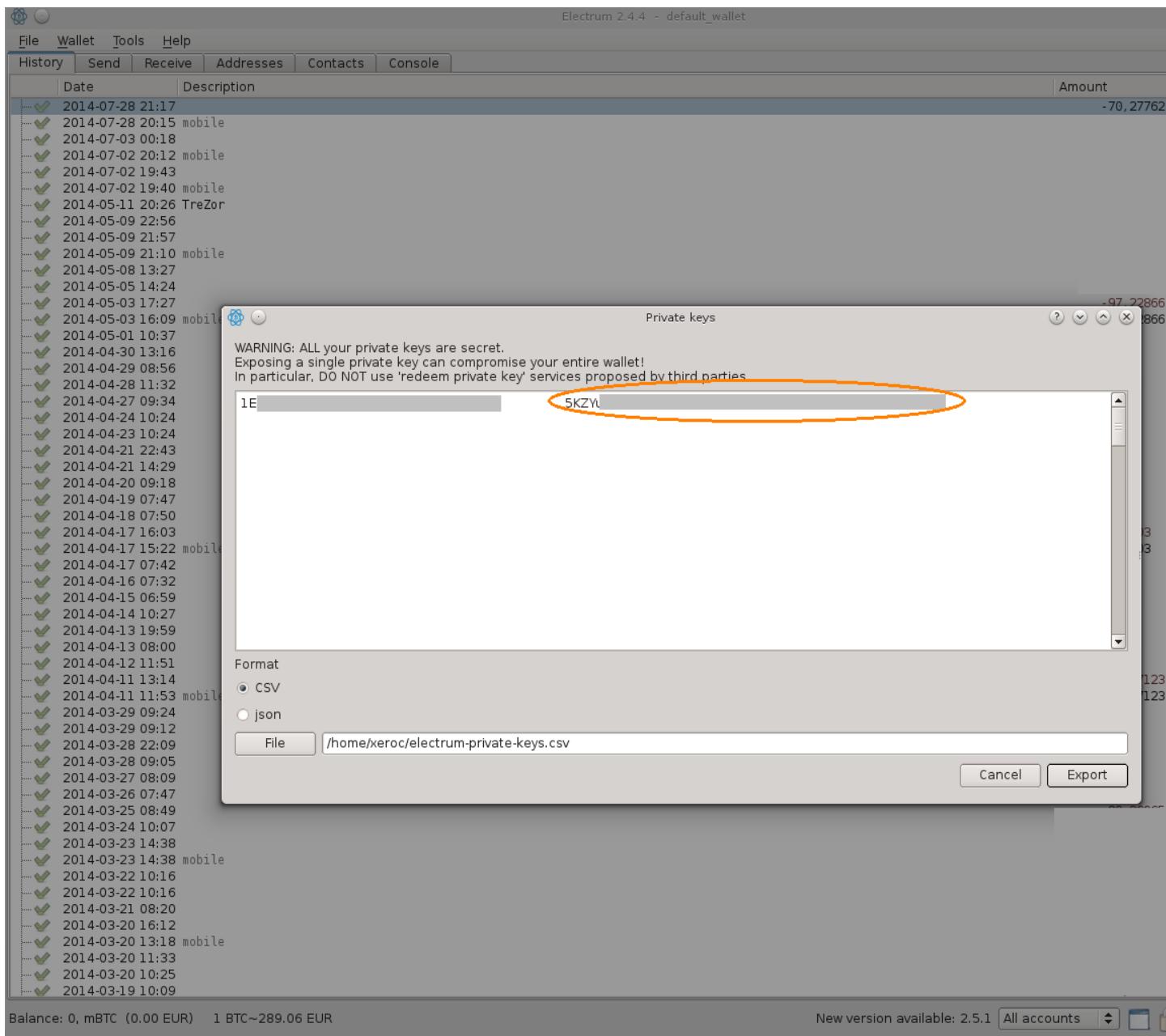
```
{
    "guid": "7611657d-25b2-4197-9d24-cdb41da2524e",
    "sharedKey": "8ab31b4b-5c91-45fb-b86a-c40e85ff2770",
    "options": {
        "pbkdf2_iterations": 10,
        "fee_policy": 0,
        "html5_notifications": false,
        "logout_time": 600000,
        "tx_display": 0,
        "always_keep_local_backup": false,
        "transactions_additional_seeds": []
    },
    "keys": [
        {
            "addr": "1MnE1rseVBgyaXKs3mScnnXpDM4ND1T3Wm",
            "priv": "5KgIMdeieZrqAqFsTkiEEEdKnkBEBpeec433Rc6XvL2YMsFUpjtik",
            "created_time": 0,
            "cre
e": "javascript_web",
            "created_device_version": "1.0"
        }
    ]
}
```

Electrum

In Electrum we need to go through the menu:

```
Wallet -> Private Keys -> Export
```

There you can identify your private key.



Armory

For armory users, the private key can be located by double-clicking your wallet in the Armory main window, click “Backup this wallet”, select “Export Key Lists” and click the button of the same name. After having supplied your password, you’ll be presented with your private key in different encodings. You can remove all checkboxes, except “Private Key (Plain Base58)”. Check the “Omit spaces in key data” box. Now select the key string and copy it to the clipboard.

All Wallet Keys

The textbox below shows all keys that are part of this wallet, which includes both permanent keys and imported keys. If you simply want to backup your wallet and you have no imported keys then all data below is reproducible from a plain paper backup.

If you have imported addresses to backup, and/or you would like to export your private keys to another wallet service or application, then you can save this data to disk, or copy&paste it into the other application.

Warning: The text box below contains the plaintext (unencrypted) private keys for each of the addresses in this wallet. This information can be used to spend the money associated with those addresses, so please protect it like you protect the rest of your wallet.

<p>Created: 28-07-2014 21:56 Wallet ID: 2TxzqrV9y Wallet Name: SJCX wallet</p> <p>PrivBase58: 5JiaTwgkS8R1giMgbbcwpk8vu3NKnFMkRX5UwGw4F12ybTyky4X</p>	<input type="checkbox"/> Address String <input type="checkbox"/> Hash160 <input checked="" type="checkbox"/> Private Key (Plain Base58) <input type="checkbox"/> Private Key (Plain Hex) <input type="checkbox"/> Public Key (BE) <input type="checkbox"/> Chain Index <hr/> <input type="checkbox"/> Imported Addresses Only <input type="checkbox"/> Include Unused Addresses <input type="checkbox"/> Include Paper Backup
--	---

[<<< Go Back](#) Omit spaces in key data [Copy to Clipboard](#) [Save to File](#)

BitShares' NOTE Token

Since the snapshot has taken place already, all you need to do now to get access to your funds in MUSE is described in the following. Note that the name of the asset in BitShares 1.0 was **NOTE** instead of **MUSE**.

Firstly, you need to upgrade your BitShares client to version 0.9.3c. To do the upgrade you need to:

- download the installation file from the [bitshares webpage](#)
- uninstall your previous version of the BitShares client
- install the new version

Attempt to sync with the blockchain (this is only necessary with if you think that since the last time you did the syncing there have been some new transactions involving any of your accounts).

You can see the syncing progress from the status bar or from the `info` command in the console (account list->advanced settings->console). After having *synced* the blockchain, your wallet will automatically

attempt to rescan the blockchain for new transactions. Depending on the amount of accounts in your wallet, this step should only take very few minutes.

Since BitShares 0.9.3c, we have a Graphene compatible Export Keys function that can be accessed in two ways:

- by accessing it in the main menu
- by issuing a command in the console.

Note that your private keys will be encrypted and you will be required to provide the corresponding pass phrase when importing your funds into MUSE.

Using the main menu

Just select File Menu -> Export Wallet and you'll be asked to select a file location where the keys will be exported.

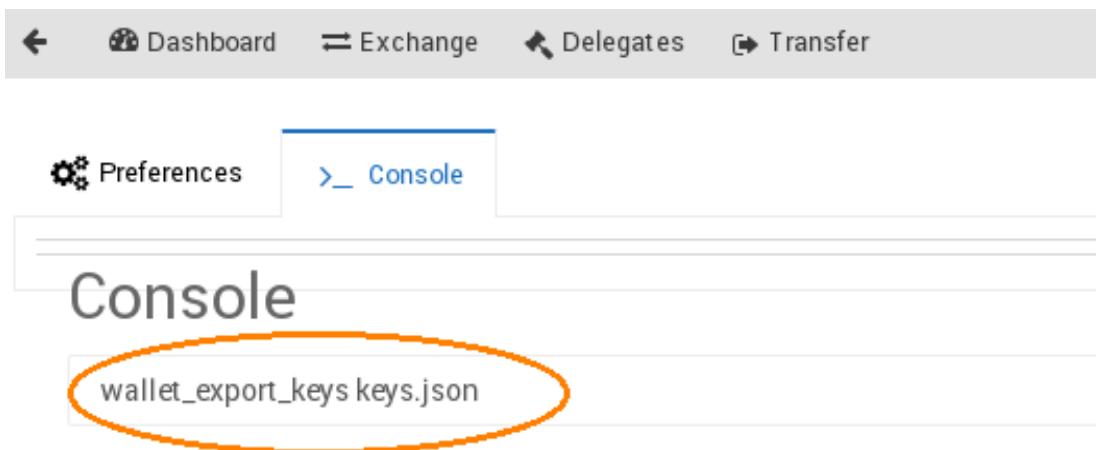
Note: Due to a known bug, if you are on Windows the only option that will work for you is the console command - the file exported using the menu will not be compatible with BTS 2.0. This refers to Windows only.

Using the console

- navigate to the console: Account List -> Advanced Settings -> Console
- type: `wallet_export_keys [full path to the file]/[file name].json` e.g. on Windows: `wallet_export_keys C:\Users\[your user name]\Desktop\keys.json` e.g. on Mac: `wallet_export_keys /Users/[your user name]/Desktop/keys.json` e.g. on Linux: `wallet_export_keys /home/[your user name]/Desktop/keys.json`
- Please replace [your user name] with your Windows account name.
- and hit Enter

Note: The exported wallet file will be encrypted with your pass phrase! Make sure to remember it when trying to use that file again!

Note: If you are on Windows and your file path tries to access the C drive directly (e.g. C:\keys.json) you might need to run the BitShares client as an administrator. So the least complicated option will be to aim for the desktop as in the example above.



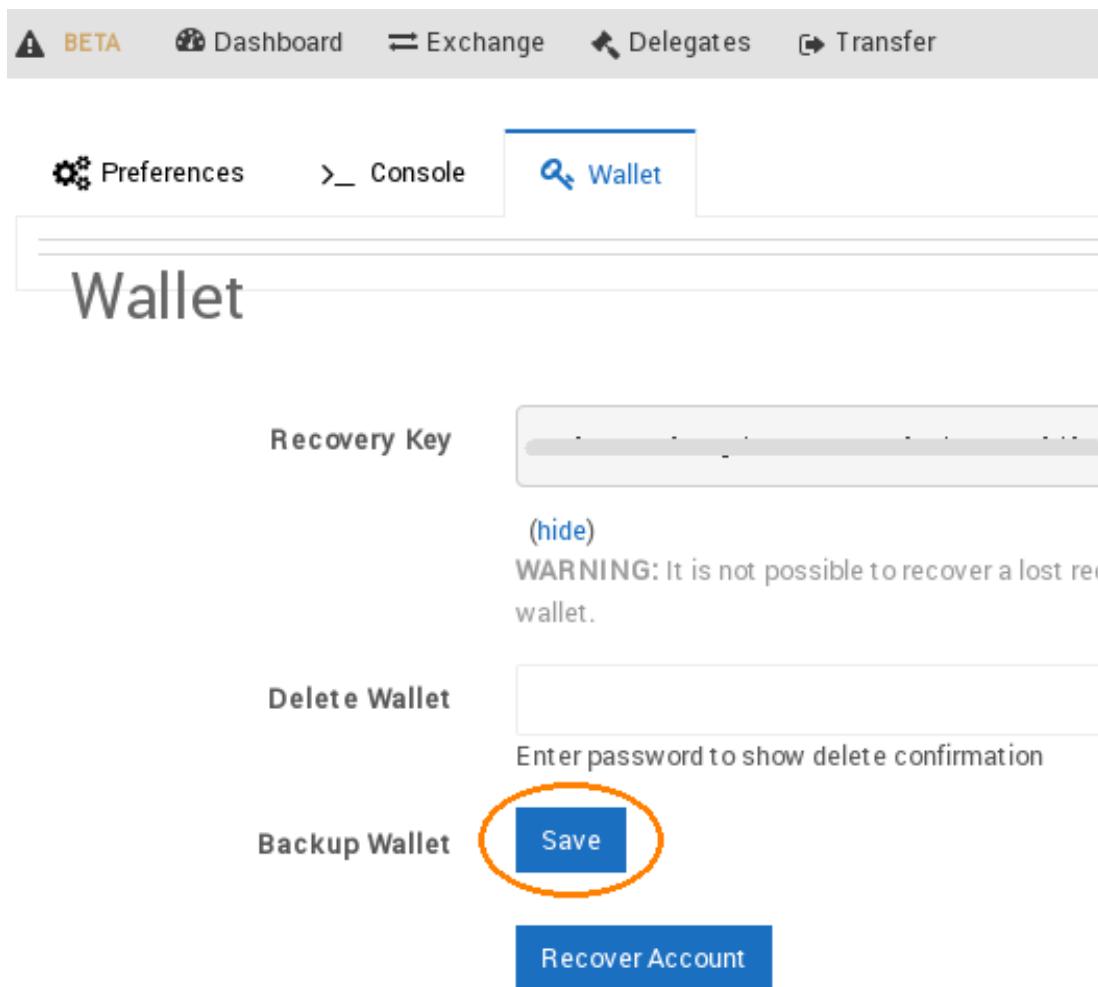
The screenshot shows the Graphene web wallet interface. At the top, there is a navigation bar with links: Dashboard, Exchange, Delegates, Transfer, Preferences, and Console. The 'Console' link is underlined, indicating it is the active tab. Below the navigation bar, the word 'Console' is displayed in a large, bold font. Underneath, a command is entered into the console: 'wallet_export_keys keys.json'. This command is circled with a red oval. The rest of the console output is a list of available commands.

```
>> help

clear_console
[command_name]? (alias for: help [command_name])
about
approve_register_account <account_salt> <paying_account_name>
batch <method_name> <parameters_list>
batch_authenticated <method_name> <parameters_list>
blockchain_broadcast_transaction <trx>
blockchain_calculate_debt <asset> [include_interest]
blockchain_calculate_max_supply [average_delegate_pay_rate]
blockchain_calculate_supply <asset>
blockchain_export_fork_graph [start_block] [end_block] [file]
```

wallet.bitshares.org

The keys of the [web wallet](#) can be exported simply by downloading a backup wallet. It can be obtained from the web wallet's preferences: (*Account List->Advanced Settings->Wallet*).

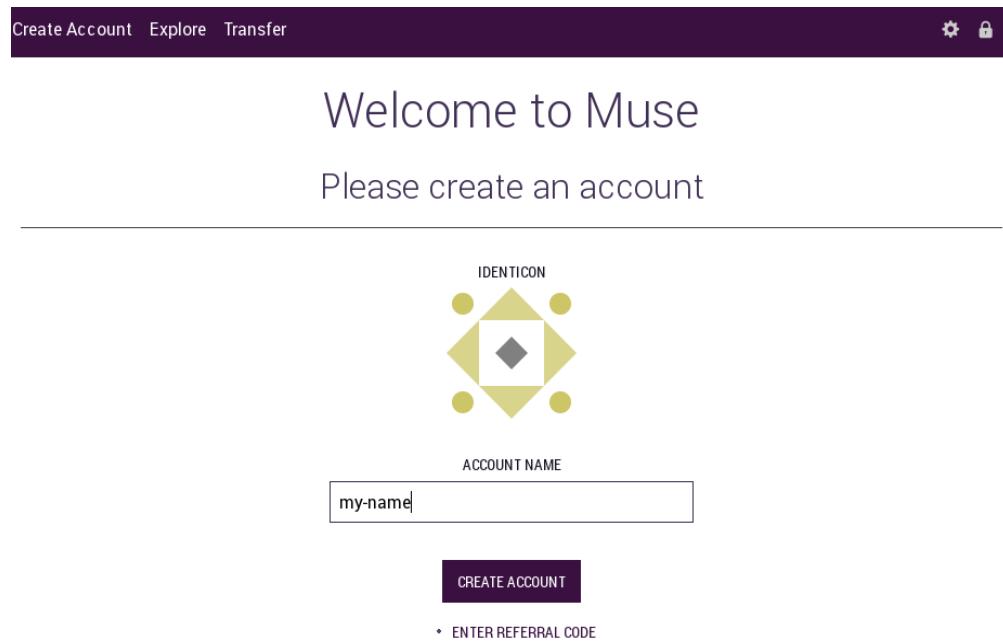


Creating a MUSE account

In order to use MUSE, you will need to register an account. All you need to provide is

- an account name and
- a password to protect your (default) wallet.

The identicon at the top can be used to verify your account name to third parties. It is derived from your account name and gives a second verification factor. And this is how you register your account:

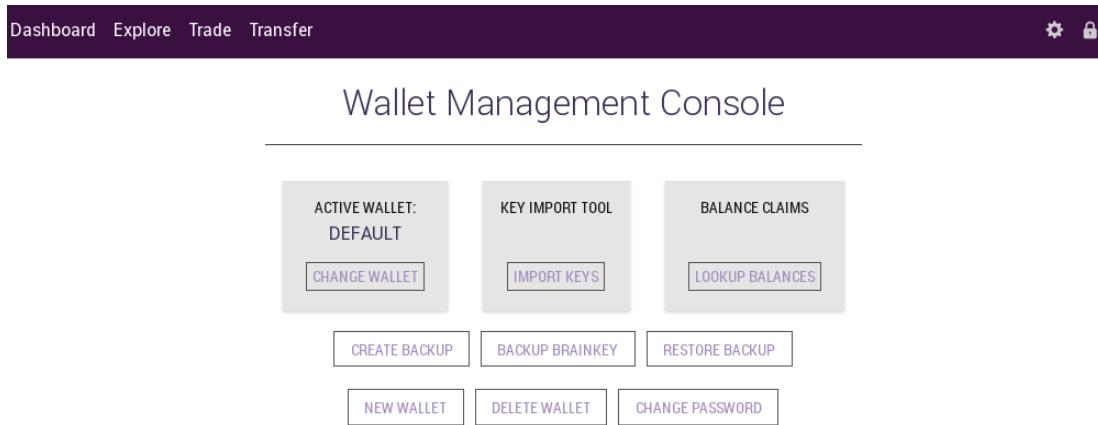


Note that, in contrast to any other platform you have ever used: Your account name can be seen similar to a mail address in such that it is **unique** and every participant in the MUSE network can interact with you independent of the actual partner providing the wallet.

Importing Your Balance

Using the Web Wallet (recommended)

The web wallet of MUSE has a **Wallet management Console**. that will help you import your funds. It can be access via *MUSE: Settings -> Wallets*

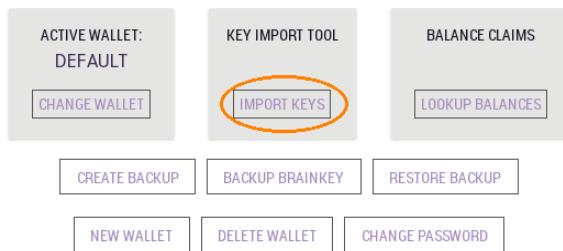


In order to import your existing accounts and claim all your funds you need to choose Import Keys.

Note: If loading the file files with invalid format please ensure that you have followed the steps described *Exporting your wallet* and make sure to click Import Keys and **not** Restore Backup.



Wallet Management Console



Here you can provide the wallet backup file produced from BitShares 0.9.3c and the pass phrase. Depending on the size of your import file, this step may take some time to auto-complete. Please be patient.



Wallet Management Console

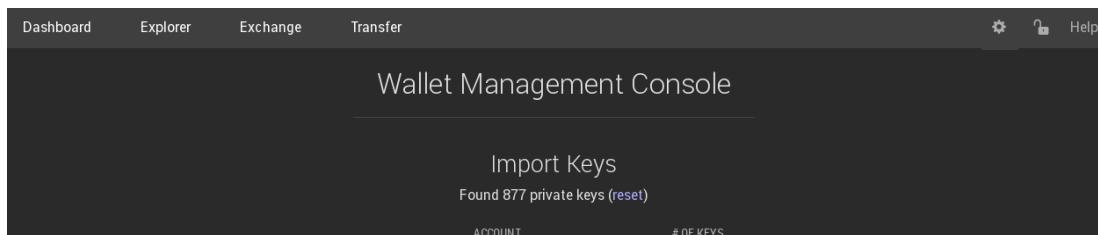
Import Keys

BTS 0.9.X KEY EXPORT FILE

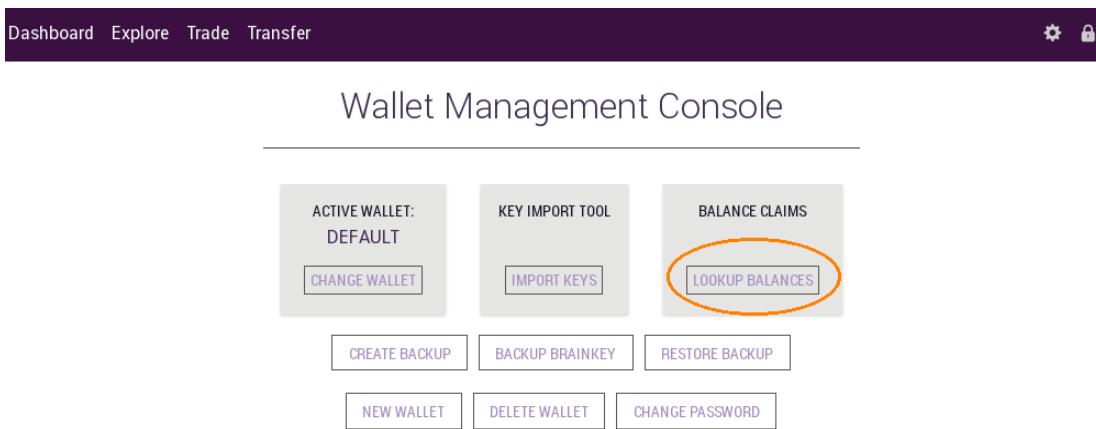
No file selected.

PASTE PRIVATE KEYS (WALLET IMPORT FORMAT - WIF)

The wallet will list all of your accounts including the number of private keys stored in the account names accordingly. The more often you have used your account, the higher this number should be. Confirm by pressing Import.



The wallet management console will now give an overview over unclaimed balances.



If you click on Balance Claim you will be brought to this screen.

The screenshot shows the 'Claim balances' screen. It has a dropdown menu 'Select Account...' (circled in orange) and a vertical list of accounts on the left. The main table has columns for 'UNCLAIMED', 'UNCLAIMED (VESTING)', and 'ACCOUNT'. The data in the table is as follows:

UNCLAIMED	UNCLAIMED (VESTING)	ACCOUNT
RCOIN		sharedrops.xeroc
EWBIE		sharedrops.xeroc
16 USD		sharedrops.xeroc
1 CORE		xeroc
1 FREE		xeroc
14 EUR		
1 CORE		xeroc
1 BTC		xeroc
TEAUX		xeroc
RCOIN		
1 CORE		
TSATM		
1 CORE		
11 USD		
1 NOTE		
1 CORE		
1 CORE		
RCOIN		

At the bottom, it says 'Graphene' and 'BACKUP RECOMMENDED Head block #73124'.

You are asked to define where to put your individual balances if you have multiple accounts.

After confirming all required steps, your accounts and the balances should appear accordingly.

Note: After importing your accounts and balances, we recommend to make a new backup of your wallet that will then contain access to your newly imported accounts and corresponding balances.

Using the Console Client (advanced users)

The wallet backup file can be imported by

```
>>> import_accounts <path to exported json> <password of wallet you exported from>
```

Note that this doesn't automatically claim the balances.

Claiming Balances

For each account <my_account_name> in your wallet (run `list_my_accounts` to see them)::

```
>>> import_account_keys /path/to/keys.json <my_password> <my_account_name> <my_
˓→account_name>
```

Note: In the release tag, this will create a full backup of the wallet after every key it imports. If you have thousands of keys, this is quite slow and also takes up a lot of disk space. Monitor your free disk space during the import and, if necessary, periodically erase the backups to avoid filling your disk. The latest code only saves your wallet after all keys have been imported.

The command above will only import your keys into the wallet and will **not** claim your funds. In order to claim the funds you need to execute::

```
>>> import_balance <my_account_name> [ "*" ] true
```

Note: If you would like to preview this claiming transaction, you can replace the `true` with a `false`. That way, the transaction will not be broadcast.

To verify the results, you can run::

```
>>> list_account_balances <my_account_name>
```

Manually claim balances

Balances can be imported one by one. The proper syntax to do so is:

```
>>> import_balance <account name> <private key> true
```

But I always import my accounts and then use the GUI to import my balances cause it's way easier.

CHAPTER 3

Integration Guide

3.1 Integration Guide

Banks, exchanges and merchants are integrating with the Graphene technology already to power instant cross-border remittance, corporate payments, voting, and decentralized trading. This page serves as a technical documentation for integrating **any** Graphene based technology to profiting from

- its real-time blockchain technology,
- existing user base,
- its network effect, and
- existing ecosystem.

Note: We offer a *low noise* skype group for important announcements for our partners, including exchanges, merchants and integrated businesses. If you would like to be added, please send a short mail to [Fabian](#) describing your service.

3.1.1 Basic Knowledge

We here illustrate the steps necessary to securely operate as a merchant, trader, exchange, or fiat-gateway. Some basic knowledge should be known to all of them before starting to integrate.

What is Different in BitShares

Here we give a brief overview of what is different in BitShares 2.0 when compared to satoshi-based blockchains such as Bitcoin, Litecoin, etc.. from the perspective of an exchange.

Several Tokens

In contrast to all satoshi-based clients, BitShares 2.0 offers a variety of blockchain tokens. There is not just the BTS (core token) but many others. Hence, as an exchange you need to distinguish different assets, either by their id (1.3.0 (BTS), 1.3.1 (USD), ...) or by there symbol.

Registered Identities

All participants in BitShares 2.0 are required to have a registered unique name. This is similar to mail addresses and are used to address recipients for transfers. As an exchange you will only ever need to tell your customers your BitShares account name and they will be able to send you funds.

No More Addresses

In BitShares 2.0, we have separated the permissions from the identity. Hence, as an exchange you don't need to ever deal with addresses again. In fact, you actually cannot possibly use an address because they only define so called *authorities* that can control the funds (or the account name). This should greatly simplify integration as you don't need to store thousands of addresses and their corresponding private keys.

Memos

In order to distinguish customers, we make use of so called *memos* similar to BitShares 1, which are encrypted. In contrast to BitShares 1.0, we now have a separated memo key that is only capable of decoding your memo and cannot spend funds. Hence, in order to monitor deposits to the exchange you no longer need to expose the private key to an internet connected machine. Instead you only decode the memo and leave the funds where they are.

Securing Funds

Funds can be secured by *hierarchical cooperate accounts*. In practise, they are (Threshold) Multi-Signature accounts from which funds can only be spend if several signatures are valid. In contrast to mostly every other crypto currency, you can propose a transaction on the blockchain and don't need other means of communications to add your approval to a certain transactions. You can find more details about these account types in

- [..../user/account-memberships](#)
- [Securing Funds](#)

Full Nodes and Clients

We have rewritten the core components from scratch and separated the core P2P and blockchain components from the wallet. Hence, you can run a full node without a wallet and connect your wallet to any public (or non-public) full-node (executable *witness_node*). The communication can be established securely but the private keys never leave the wallet.

Object IDs

Since BitShares 2.0 offers a variety of features to its users that are different in many ways, we have decided to address them using *object ids*. For instance:

Object ID translates to	
1.3.1	asset USD
1.3.0	asset BTS
1.2.<id>	user with id <id>
1.6.<id>	block signer <id>
1.11.<id>	operation with id <id>

Read more in the distinct [..../blockchain/Objects](#) page.

Blockchain Interaction

To interface your existing platform with BitShares, you can make use of [Remote Procedure Calls](#) and [WebSocket Calls & Notifications](#) to either a full node (for monitoring only) or the CLI wallet (for accessing funds).

All API calls are formated in JSON and return JSON only. You can read more about the API in the separated [API documentation](#).

The Graphene toolkit comprises several tools that allow interaction with the blockchain on different levels and shall thus be briefly described in the following sections.

Full Node

We here distinguish between full nodes (a.k.a. *non-block producing* witness nodes) and *block producing* witness nodes. Both are implemented by the same executable but the latter requires some additional parameters to be defined and the corresponding witness voted active by the shareholders.

Both represent nodes in the network that verify all transactions and blocks against the current state of the overall network. Hence, we recommend all service providers to run and maintain their own **full nodes** for reliability and security reasons.

Full Nodes

The full node is launched according to:

```
./programs/witness_node/witness_node
```

It takes an optional `-data-dir` parameter to define a working and data directory to store the configuration, blockchain and local databases (defaults to `witness_node_data_dir`). Those will be automatically created with default settings if they don't exist locally set.

Configuration

The configuration file `config.ini` in your data directory is commented and contains the following essential settings:

- **p2p-endpoint** Endpoint for P2P node to listen on
- **seed-node** P2P nodes to connect to on startup (may specify multiple times)
- **checkpoint** Pairs of [BLOCK_NUM,BLOCK_ID] that should be enforced as checkpoints.
- **rpc-endpoint** Endpoint for websocket RPC to listen on (e.g. 0.0.0.0:8090)
- **rpc-tls-endpoint** Endpoint for TLS websocket RPC to listen on

- **server-pem** The TLS certificate file for this server
- **server-pem-password** Password for this certificate
- **genesis-json** File to read Genesis State from
- **api-access** JSON file specifying API permissions
- **enable-stale-production** Enable block production, even if the chain is stale. (unless for private test-nets should be `false`)
- **required-participation** Percent of witnesses (0-99) that must be participating in order to produce blocks
- **allow-consecutive** Allow block production, even if the last block was produced by the same witness.
- **witness-id** ID of witness controlled by this node (e.g. “1.6.5”, quotes are required, may specify multiple times)
- **private-key** Tuple of [PublicKey, WIF private key] (may specify multiple times)
- **track-account** Account ID to track history for (may specify multiple times)
- **bucket-size** Track market history by grouping orders into buckets of equal size measured in seconds specified as a JSON array of numbers
- **history-per-size** How far back in time to track history for each bucket size, measured in the number of buckets (default: 1000)

Note: Folders and files are considered to be relative to the working directory (i.e. the directory from which the executables are launched from)

Enabling Remote Procedure Calls (RPC)

In order to allow RPC calls for blockchain operations you need to modify the following entry in the configuration file::

```
rpc-endpoint = 0.0.0.0:8090
```

This will open the port 8090 for global queries only. Since the witness node only maintains the blockchain and (unless you are an actively block producing witness) no private keys are involved, it is safe to expose your witness to the internet.

Restarting the witness node

When restarting the witness node, it may be required to append the `-replay-blockchain` parameter to regenerate the local (in-memory) blockchain state.

Enabling Block Production

For block production, the required parameters to be defined are

- **witness-id** and
- **private-key** as a pair of public key and wif private key.

The witness-id and public key can be obtain via::

```
>>> get_witness <accountname>
{
    [...]
    "id": "1.6.10",
    "signing_key": "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
    [...]
}
```

Assuming we want to maintain the witness with id 1.6.10, the corresponding setting would be::

```
witness_id = "1.6.10"
```

The required private keys can be exported from most wallets (e.g. `dump_private_keys`) for configuration according to::

```
private_key = ["BTS7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
    ↵ "5JGi7DM7J8fSTizZ4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"]
```

Delayed Full Node

The delayed full node node will provide us with a delayed and several times confirmed and verified blockchain. Even though DPOS is more resistant against forks than most other blockchain consensus schemes, we delay the blockchain here to reduces the risk of forks even more. In the end, the delayed full node is supposed to never enter an invalid fork.

The delayed full node will need the IP address and port of the p2p-endpoint from the trusted full node and the number of blocks that should be delayed. We also need to open the RPC/Websocket port (to the local network!) so that we can interface using RPC-JSON calls.

For our example and for 10 blocks delaye (i.e. 30 seconds for 3 second block intervals), we need:

```
./programs/delayed_node/delayed_node --trusted-node="192.168.0.100:8090" \
--delay-block-count=10 \
--rpc-endpoint="192.168.0.101:8090"
```

CLI Wallet

The following will explain how to use the console wallet (not GUI).

Launching

The `cli_wallet` creates a local `wallet.json` file that contains the encrypted private keys required to access the funds in your account. It **requires** a running witness node (not necessarily locally) and connects to it on launch:

```
programs/cli_wallet/cli_wallet -s ws://127.0.0.1:8090
```

Depending on the actual chain that you want to connect to your may need to specify `-chain-id`.

Enabling Remote Procedure Calls (RPC)

In order to allow RPC calls for wallet operations (spend, buy, sell, ...) you can choose between pure RPC or RPC-HTTP requests. In this tutorial, the latter is prefered since well established libraries make use of the RPC-HTTP protocol.

The cli-wallet can open a RPC port so that you can interface your application with it. You have the choices of * websocket RPC via the `-r` parameter, and * HTTP RPC via the `-H` parameter:

To enable RPC-HTTP in your wallet you need to run:

```
# recommended for use with python, or curl:  
programs/cli_wallet/cli_wallet --rpc-http-endpoint="127.0.0.1:8092"  
# or  
programs/cli_wallet/cli_wallet --rpc-endpoint="127.0.0.1:8092"
```

depending on the kind of RPC protocol.

This will open the port 8092 for local queries only. It is not recommended to publicly expose your wallet!

A tutorial for the BitShares cli-wallet can be found in the BitShares tutorials.

Web Wallet

The web wallet is a wallet implemented solely in Javascript. It makes use of modern Web development tools and libraries – to just a few:

- Coffee-Script
- LESS
- React-JS
- WebPack
- LoDash
- Foundation
- Highcharts
- Mocha
- ...

The webwallet (per default) connects to a full node (non-block-producing witness node) on the same host via web-sockets.

Download

The sources can be downloaded from github.

```
git clone https://github.com/cryptonomex/graphene-ui
```

They consist of libraries, a JS-console, and the wallet, as well as other tools.

Installing Dependencies

First, we need to install the dependencies via `npm`:

```
for I in cli dl web; do cd $I; npm install; cd ..; done
```

Bundling

We now bundle the web wallet into JavaScript, CSS, and HTML assets.

```
cd ./web  
npm run build
```

The resulting assets can be found in the *dist* folder.

Testing Bundle

```
npm test
```

Live Development

```
npm start
```

Network and Wallet Configuration

Similar to other crypto currencies, it is recommended to wait for several confirmations of a transaction. Even though the consensus scheme of Graphene is a lot more secure than regular proof-of-work or other proof-of-stake schemes, we still support exchanges that require more confirmations for deposits.

Components

P2P network

The BitShares client uses a peer-to-peer network to connect and broadcasts transactions there. A block producing full node will eventually catch your transaction and validate it by adding it into a new block.

Trusted Full Node

We will use a Full node to connect to the network directly. We call it *trusted* since it is supposed to be under our control.

Wallet

The wallet is used to initiate transfers (customer withdrawals) and connects to the trusted full node.

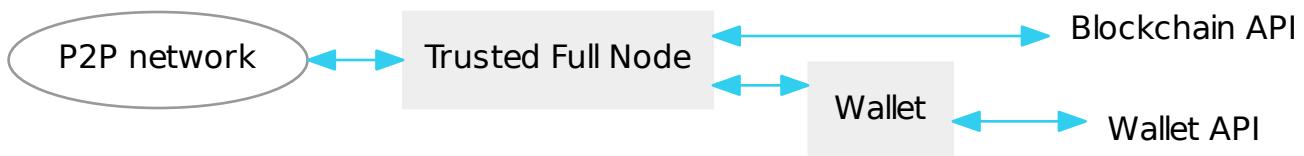
Wallet API

Since we have a delayed full node that we can fully trust, we will interface with this node to query the blockchain and receive notifications from it on balance changes. Hence, we use this API to watch deposits of users into the exchange's account. Because the delayed node only knows about irreversible blocks all transactions are at this point irreversible as well. For customer withdrawals, we will interface with the wallet to initiate transfers to the accounts of the customers on request. As we are connected to the trusted node directly, there will not be any delay on withdrawals.

Network Setups

General Setup

For general purpose setups, we recommend a reduced complexity setup that looks as follows

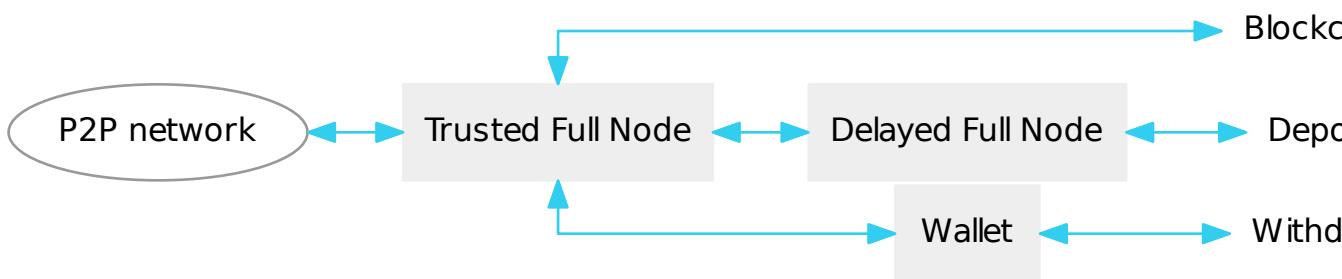


A tutorial to setup this network can be found here:

- [tutorials/general-network-setup](#)

High Security Setup

For high security, we provide a so called *delayed* full node which accepts the parameter `trusted-node` for an RPC endpoint of a trusted validating node. The trusted-node is a regular full node directly connected to the P2P network that works as a proxy. The delayed node will delay blocks until they are **irreversible**. Depending on the block interval and the number of witnesses, this may lead to a few minutes of delay.



A tutorial to setup this network can be found here:

- [tutorials/secure-network-setup](#)

Often used API Calls

Table of Contents

- *Often used API Calls*
 - *Overview*
 - *list_account_balances <account>*
 - *transfer <from> <to> <amount> <asset> "<memo>" <broadcast>*
 - *transfer2 <from> <to> <amount> <asset> "<memo>"*
 - *get_account_history <account> <limit>*
 - *get_object "1.11.<id>"*
 - *get_asset <USD>*
 - *Correspondences with BitShares 1.0 Calls*

Overview

Some API calls that are most interesting for exchanges and gateways are listed in the following table. They are compared to their corresponding API calls in BitShares 1.0.

We will now take a look at some sample outputs for some of the API calls in the table above. We recommend to read the following articles:

Objects and IDs

In contrast to most cryptocurrency wallets, the BitShares 2.0 has a different model to represent the blockchain, its transactions and accounts. This chapter wants to give an introduction to the concepts of *objects* as they are used by the BitShares 2.0 client. Furthermore, we will briefly introduce the API and show how to subscribe to object changes (such as new blocks or incoming deposits). Afterwards, we will show how exchanges may monitor their accounts and credit incoming funds to their corresponding users.

Objects

On the BitShares blockchains there are no addresses, but objects identified by a unique *id*, an *type* and a *space* in the form::

```
space.type.id
```

List of commonly used objects

ID	Object Type
1.1.x	base object
1.2.x	account object
1.3.x	asset object
1.4.x	force settlement object
1.5.x	committee member object
1.6.x	witness object
1.7.x	limit order object
1.8.x	call order object
1.9.x	custom object
1.10.x	proposal object
1.11.x	operation history object
1.12.x	withdraw permission object
1.13.x	vesting balance object
1.14.x	worker object
1.15.x	balance object
2.0.x	global_property_object
2.1.x	dynamic_global_property_object
2.3.x	asset_dynamic_data
2.4.x	asset_bitasset_data
2.5.x	account_balance_object
2.6.x	account_statistics_object
2.7.x	transaction_object
2.8.x	block_summary_object
2.9.x	account_transaction_history_object
2.10.x	blinded_balance_object
2.11.x	chain_property_object
2.12.x	witness_schedule_object
2.13.x	budget_record_object
2.14.x	special_authority_object

Examples

To get a feeling about what these objects do, we recommend to obtain these exemplary objects.

- 2.0.0 (global blockchain parameters)
- 2.1.0 (current blockchain data)
- 1.2.0 (committee-account details)
- 1.3.0 (core asset details)

Definitions

For advanced users that want to deal with the C++ code of graphene, we here list the definition of `object_type` and `impl_object_type`:

Protocol Space (1.x.x)

```
enum graphene::chain::object_type
```

List all object types from all namespaces here so they can be easily reflected and displayed in debug output. If a 3rd party wants to extend the core code then they will have to change the packed_object::type field from enum_type to uint16 to avoid warnings when converting packed_objects to/from json.

Values:

```
null_object_type
base_object_type
account_object_type
asset_object_type
force_settlement_object_type
committee_member_object_type
witness_object_type
limit_order_object_type
call_order_object_type
custom_object_type
proposal_object_type
operation_history_object_type
withdraw_permission_object_type
vesting_balance_object_type
worker_object_type
balance_object_type
OBJECT_TYPE_COUNT
```

Sentry value which contains the number of different object types.

Implementation Space (2.x.x)

```
enum graphene::chain::impl_object_type
```

Values:

```
impl_global_property_object_type
impl_dynamic_global_property_object_type
impl_reserved0_object_type
impl_asset_dynamic_data_type
impl_asset_bitasset_data_type
impl_account_balance_object_type
impl_account_statistics_object_type
impl_transaction_object_type
impl_block_summary_object_type
```

```
impl_account_transaction_history_object_type
impl_blinded_balance_object_type
impl_chain_property_object_type
impl_witness_schedule_object_type
impl_budget_record_object_type
impl_special_authority_object_type
impl_buyback_object_type
impl_fba_accumulator_object_type
```

Wallet API Calls

The wallet (`cli_wallet`) requires a running full node to connect to because it does not offer P2P or blockchain capabilities directly.

If you have not set up your wallet yet, you can find more information on the [CLI Wallet](#) and the [CLI Wallet FAQ](#) pages.

Table of Contents

- *Wallet API Calls*
 - *General Calls*
 - *Wallet Calls*
 - *Account Calls*
 - *Trading Calls*
 - *Asset Calls*
 - *Governance*
 - *Privacy Mode*
 - *Blockchain Inspection*
 - *Transaction Builder*

General Calls

```
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 multi-line string suitable for displaying on a terminal

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

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

Wallet Calls

`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

`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

`void graphene::wallet::wallet_api::lock()`

Locks the wallet immediately.

`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()`

`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.

`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

`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

```
map<string, bool> graphene::wallet::wallet_api::import_accounts (string filename, string password)
bool graphene::wallet::wallet_api::import_account_keys (string filename, string password, string src_account_name, string dest_account_name)
vector<signed_transaction> graphene::wallet::wallet_api::import_balance (string ac-
count_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.

`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).

Return a suggested brain_key

```
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.

`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.

```
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

```
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

```
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.

Account Calls

```
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

```
map<string, account_id_type> graphene::wallet::wallet_api::list_accounts(const string  
&lower-  
bound,  
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)

```
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’s balances

Parameters

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

```
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

```
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

```
signed_transaction graphene::wallet::wallet_api::create_account_with_brain_key(string
    brain_key,
    string
    ac-
    count_name,
    string
    reg-
    is-
    trar_account,
    string
    re-
    fer-
    rer_account,
    bool
    broad-
    cast
    =
    false)
```

Creates a new account and registers it on the blockchain.

See [suggest_brain_key\(\)](#)

See [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

```
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

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

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

```
signed_transaction graphene::wallet::wallet_api::whitelist_account(string authoriz-  
ing_account, string account_to_list, ac-  
count_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

```
vector<vesting_balance_object_with_info> graphene::wallet::wallet_api::get_vesting_balances(string  
ac-  
count_name)
```

Get information about a vesting balance object.

Parameters

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

```
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

```
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

```
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

```
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)

```
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

Trading Calls

```
signed_transaction graphene::wallet::wallet_api::sell_asset (string seller_account, string  
                                         amount_to_sell,           string  
                                         symbol_to_sell,         string  
                                         min_to_receive,        string sym-  
                                         bol_to_receive,       uint32_t time-  
                                         out_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 BTS with USD, you should sell USD for BTS.

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 block chain.

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

```
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

```
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

```
signed_transaction graphene::wallet::wallet_api::settle_asset (string    account_to_settle,
                                                               string  amount_to_settle,
                                                               string symbol, bool broad-
                                                               cast = 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'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`.

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 settle on
- `broadcast`: `true` to broadcast the transaction on the network

```
vector<bucket_object> graphene::wallet::wallet_api::get_market_history(string symbol,
                                                                     string sym-
                                                                     bol2, uint32_t
                                                                     bucket,
                                                                     fc::time_point_sec
                                                                     start,
                                                                     fc::time_point_sec
                                                                     end) const

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

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

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

Asset Calls

```
vector<asset_object> graphene::wallet::wallet_api::list_assets(const string &lower-
                                                               bound, 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)

```
signed_transaction graphene::wallet::wallet_api::create_asset(string issuer, string sym-
                                                               bol, uint8_t precision,
                                                               asset_options common,
                                                               fc::optional<bitasset_options>
                                                               bitasset_opts, bool broad-
                                                               cast = 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

```
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

```
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

```
signed_transaction graphene::wallet::wallet_api::update_asset_feed_producers(string
    sym-
    bol,
    flat_set<string>
    new_feed_producers,
    bool
    broad-
    cast
    =
    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

```
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 eachother, 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

```
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

```
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 block chain

Parameters

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

```
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

```
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

```
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

```
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

Governance

```
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

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 block chain

Parameters

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

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 block chain

Parameters

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

map<string, witness_id_type> graphene::wallet::wallet_api::**list_witnesses** (const

string

&lower-

bound,

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 witnessss, 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_witnesses()* call.

Return a list of witnessss 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 witnessss to return (max: 1000)

map<string, committee_member_id_type> graphene::wallet::wallet_api::**list_committee_members** (const

string

&lower-

bound,

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)

```
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

```
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.

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.

```
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.

```
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.

```
signed_transaction graphene::wallet::wallet_api::vote_for_committee_member (string
vot-
ing_account,
string
com-
mit-
tee_member,
bool
ap-
prove,
bool
broad-
cast    =
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

```
signed_transaction graphene::wallet::wallet_api::vote_for_witness (string          vot-
ing_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

```
signed_transaction graphene::wallet::wallet_api::set_voting_proxy(string      ac-
                                                               count_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

```
signed_transaction graphene::wallet::wallet_api::set_desired_witness_and_committee_member_count(s
```

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

```
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

```
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

Privacy Mode

```
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

```
string graphene::wallet::wallet_api::get_key_label (public_key_type key) const  
public_key_type graphene::wallet::wallet_api::get_public_key (string label) const  
Return the public key associated with the given label  
map<string, public_key_type> graphene::wallet::wallet_api::get_blind_accounts ()  
const  
Return all 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
```

```
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

```
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.

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

Parameters

- *to_amounts*: map from key or label to amount

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

```
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 as-set_symbol,
    bool broadcast =
    false)
```

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

```
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

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

Return all blind receipts to/form a particular account

```
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

- *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*
- *confirmation_receipt*: - a base58 encoded stealth confirmation

Blockchain Inspection

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

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

Returns the number of accounts registered on the blockchain

Return the number of registered accounts

```
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

Return the 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

Return the dynamic global properties

```
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

Transaction Builder

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

```
void graphene::wallet::wallet_api::add_operation_to_builder_transaction(transaction_handle_type trans-
ac-
tion_handle,
const
oper-
ation
&op)
```

```

void graphene::wallet::wallet_api::replace_operation_in_builder_transaction(transaction_handle_type
    han-
    dle,
    un-
    signed
    op-
    er-
    a-
    tion_index,
const
    op-
    er-
    a-
    tion
    &new_op)

asset graphene::wallet::wallet_api::set_fees_on_builder_transaction(transaction_handle_type
    handle,
    string
    fee_asset =
    GRAPHENE_SYMBOL)

transaction graphene::wallet::wallet_api::preview_builder_transaction(transaction_handle_type
    handle)

signed_transaction graphene::wallet::wallet_api::sign_builder_transaction(transaction_handle_type
    transac-
    tion_handle,
    bool
    broad-
    cast =
    true)

signed_transaction graphene::wallet::wallet_api::propose_builder_transaction(transaction_handle_type
    han-
    dle,
    time_point_sec
    ex-
    pi-
    ra-
    tion
    =
    time_point::now() + fc::minutes
    uint32_t
    re-
    view_period_seconds
    = 0,
    bool
    broad-
    cast
    =
    true)

```

```
signed_transaction graphene::wallet::wallet_api::propose_builder_transaction2(transaction_handle_type
                                         han-
                                         dle,
                                         string
                                         ac-
                                         count_name_or_id,
                                         time_point_sec
                                         ex-
                                         pi-
                                         ra-
                                         tion
                                         =
                                         time_point::now() + fc::minu
                                         uint32_t
                                         re-
                                         view_period_seconds
                                         =
                                         0,
                                         bool
                                         broad-
                                         cast
                                         =
                                         true)

void graphene::wallet::wallet_api::remove_builder_transaction(transaction_handle_type
                                                               handle)

string graphene::wallet::wallet_api::serialize_transaction(signed_transaction      tx)
                                                               const
                                         Converts a signed_transaction in JSON form to its binary representation.

                                         TODO: I don't see a broadcast_transaction() function, do we need one?
```

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

```
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

```
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")

```
list_account_balances <account>
```

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.list_account_balances("dan")
print(json.dumps(res, indent=4))
```

Result

```
[  
  {  
    "asset_id": "1.3.0",  
    "amount": "331104701530"  
  },  
  {  
    "asset_id": "1.3.511",  
    "amount": 3844848635  
  },  
  {  
    "asset_id": "1.3.427",  
    "amount": 8638  
  },  
  {  
    "asset_id": "1.3.536",  
    "amount": 31957981  
  }]
```

Reference

```
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's balances

Parameters

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

```
transfer <from> <to> <amount> <asset> "<memo>" <broadcast>
```

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.transfer("fromaccount", "toaccount", "10", "USD", "$10 gift", True);
print(json.dumps(res, indent=4))
```

The final parameter `True` states that the signed transaction will be broadcast. If this parameter is `False` the transaction will be signed but not broadcast, hence not executed.

Result

```
{
    "ref_block_num": 18,
    "ref_block_prefix": 2320098938,
    "expiration": "2015-10-13T13:56:15",
    "operations": [
        0, {
            "fee": {
                "amount": 2089843,
                "asset_id": "1.3.0"
            },
            "from": "1.2.17",
            "to": "1.2.7",
            "amount": {
                "amount": 10000000,
                "asset_id": "1.3.0"
            },
            "memo": {
                "from": "GPH6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
                "to": "GPH6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
                "nonce": "16430576185191232340",
                "message": "74d0e455e2e5587b7dc85380102c3291"
            },
            "extensions": []
        }
    ],
    "extensions": [],
    "signatures": [
        "1f147aed197a2925038e4821da54bd7818472ebe25257ac9a7ea66429494e7242d0dc13c55c6840614e6da6a5bf65ae609",
        "1f147aed197a2925038e4821da54bd7818472ebe25257ac9a7ea66429494e7242d0dc13c55c6840614e6da6a5bf65ae609"
    ]
}
```

Reference

```
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 <from> <to> <amount> <asset> "<memo>"
```

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.transfer2("fromaccount","toaccount","10", "USD", "$10 gift");
print(json.dumps(res, indent=4))
```

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

Result

```
[b546a75a891b5c51de6d1aaf40d10e91a717bb3, {
    "ref_block_num": 18,
    "ref_block_prefix": 2320098938,
    "expiration": "2015-10-13T13:56:15",
    "operations": [
        0, {
            "fee": {
                "amount": 2089843,
                "asset_id": "1.3.0"
            },
            "from": "1.2.17",
            "to": "1.2.7",
            "amount": {
                "amount": 10000000,
                "asset_id": "1.3.0"
            }
        }
    ]
}]
```

```
        },
        "memo": {
            "from": "GPH6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
            "to": "GPH6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
            "nonce": "16430576185191232340",
            "message": "74d0e455e2e5587b7dc85380102c3291"
        },
        "extensions": []
    }
],
"extensions": [],
"signatures": [
    "1f147aed197a2925038e4821da54bd7818472ebe25257ac9a7ea66429494e7242d0dc13c55c6840614e6da6a5bf65ae609"
]
}
```

Reference

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

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

```
get_account_history <account> <limit>
```

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.get_account_history("dan", 1)
print(json.dumps(res, indent=4))
```

Result

```
[{
    "description": "fill_order_operation dan fee: 0 CORE",
```

```

    "op": {
      "block_num": 28672,
      "op": [
        4,
        {
          "pays": {
            "asset_id": "1.3.536",
            "amount": 20000
          },
          "fee": {
            "asset_id": "1.3.0",
            "amount": 0
          },
          "order_id": "1.7.1459",
          "account_id": "1.2.21532",
          "receives": {
            "asset_id": "1.3.0",
            "amount": 50000000
          }
        }
      ],
      "id": "1.11.213277",
      "trx_in_block": 0,
      "virtual_op": 47888,
      "op_in_trx": 0,
      "result": [
        0,
        {}
      ]
    },
    "memo": ""
  }
]

```

Reference

```

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)

```
get_object "1.11.<id>"
```

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.get_object("1.11.213277")
print(json.dumps(res, indent=4))
```

Result

```
{
    "trx_in_block": 0,
    "id": "1.11.213277",
    "block_num": 28672,
    "op": [
        4,
        {
            "fee": {
                "asset_id": "1.3.0",
                "amount": 0
            },
            "receives": {
                "asset_id": "1.3.0",
                "amount": 50000000
            },
            "pays": {
                "asset_id": "1.3.536",
                "amount": 20000
            },
            "account_id": "1.2.21532",
            "order_id": "1.7.1459"
        }
    ],
    "result": [
        0,
        {}
    ],
    "op_in_trx": 0,
    "virtual_op": 47888
}
```

Reference

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

get_asset <USD>

Script

```
import json
from grapheneapi import GrapheneAPI
client = GrapheneAPI("localhost", 8092, "", "")
res = client.get_asset("USD")
print(json.dumps(res, indent=4))
```

Result

```
{  
    "symbol": "USD",  
    "issuer": "1.2.1",  
    "options": {  
        "description": "1 United States dollar",  
        "whitelistAuthorities": [],  
        "flags": 0,  
        "extensions": [],  
        "core_exchange_rate": {  
            "quote": {  
                "asset_id": "1.3.536",  
                "amount": 11  
            },  
            "base": {  
                "asset_id": "1.3.0",  
                "amount": 22428  
            }  
        },  
        "whitelist_markets": [],  
        "max_supply": "10000000000000000000",  
        "blacklist_markets": [],  
        "issuer_permissions": 79,  
        "market_fee_percent": 0,  
        "max_market_fee": "10000000000000000000",  
        "blacklistAuthorities": []  
    },  
    "dynamic_asset_data_id": "2.3.536",  
    "bitasset_data_id": "2.4.32",  
    "id": "1.3.536",  
    "precision": 4  
}
```

Reference

```
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 block chain

Parameters

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

Correspondences with BitShares 1.0 Calls

BitShares 1.0 Calls	BitShares 2.0 Calls
<code>wallet_open</code>	<code>n.A. (default wallet.json)</code>
<code>wallet_unlock</code>	<code>unlock <password></code>
<code>wallet_account_balance</code>	<code>get_account_balances <account></code>
<code>wallet_address_create</code>	<code>a.A. no addresses available for sending</code>
<code>wallet_account_transaction_history</code>	<code>get_transaction_history <account> <limit></code>
<code>wallet_transfer</code>	<code>transfer <from> <to> <amount> <asset> "<memo>" <broadcast></code> <code>transfer2 <from> <to> <amount> <asset> "<memo>"</code>
<code>n.A.</code>	<code>get_transaction_id(const signed_transaction & trx)</code>
<code>blockchain_get_transaction</code>	<code>get_object 1.11.<id>(<id> integer)</code>
<code>blockchain_get_asset</code>	<code>get_asset <symbol> or get_object 1.3.<id>(<id> integer)</code>
<code>info</code>	<code>info</code>

3.1.2 Use-Cases

Exchange, Bridges, and Gateways represent business that trade or exchange assets that are located **inside** a Graphene network (e.g. BitShares) against assets that are **located** outside the blockchain network. For instance, exchanges trade `BTS:BTC` while bridges exchange `bitBTC:BTC`.

Exchanges, Bridges, and Gateways

We here illustrate the steps necessary to securely operate as exchange or gateway. Gateways take Fiat and convert them to their corresponding bitAsset at a fee and vice versa. For instance:

1. A customer requests 100 bitUSD from a gateway
2. The gateway sends an invoice with bank account details
3. When the funds arrive at the gateway a percentage is taken as a fee and the rest is transferred as bitUSD directly into the BitShares wallet of the customer.

For exchanges we recommend to also read [What is Different in BitShares](#) and [Often used API Calls](#).

Integration Instructions

Step-By-Step Instructions for Exchanges

We here describe how to interface your exchange with BitShares step-by-step. We will link to a more detailed description where appropriate.

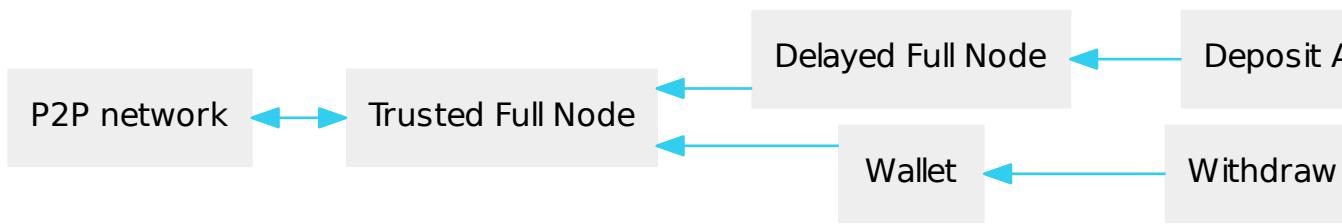
Installation

In this step-by-step instruction we assume you have successfully built from the sources according to:

- *Building from Sources*

Running Daemons and Wallet

For security reasons we will run two daemons and a wallet according to these diagram:



In this tutorial we will run all deamons and the wallet on the same machine and use different ports to distinguish them:

- port 8090: trusted full node
- port 8091: delayed node
- port 8092: wallet

[Read more details](#)

Trusted Full Node

The trusted full node is your entry point to the BitShares P2P network. It will hold the blockchain, connect to other peers, and will receive new blocks in *real-time*.

```
./programs/witness_node/witness_node --data-dir=trusted_node/ --rpc-endpoint="127.0.0.1:8090"
```

Note: Until the genesis block is integrated into the binary/sources, you may additionally need to download the genesis block from github and add the parameter `--genesis-json <genesis.json>`. (See [Release Page](#))

Note: Unless the seed nodes are encoded into the binary, you may need to add a known seed node with `-s xxx.xxx.xxx:yyy` in order to initially connect to the P2P network. (See [Release Page](#))

Note: To start a node with reduced RAM please see [Memory reduction for nodes](#)

Delayed Node

The delayed full node node will provide us with a delayed and several times confirmed and verified blockchain. All transactions that are confirmed by the delayed node are **irreversible**.

```
./programs/delayed_node/delayed_node --trusted-node="127.0.0.1:8090" \
--rpc-endpoint="127.0.0.1:8091"
-d delayed_node \
-s "0.0.0.0:0" \
--p2p-endpoint="0.0.0.0:0" \
--seed-nodes "[]"
```

We will use this node for notifications of customer deposits.

Wallet

The wallet will be used to transfer assets to the customers. It connects to the trusted full node and has spending privileges for the hot wallet.

```
./programs/cli_wallet/cli_wallet --server-rpc-endpoint="ws://127.0.0.1:8090" \
--rpc-http-endpoint="127.0.0.1:8092"
```

Query blockchain for required data

We now use the open `cli_wallet` to issue transfers and query the blockchain for more information. First of all, we create a new wallet and set a pass phrase::

```
>>> set_password <password>
```

Existing BitShares 1 Account

We assume that you already have an account on the BitShares blockchain and show how to export it from the BitShares 1 client.

We first get the account statistics ID (2.6.*.) of the deposit account to monitor deposits, the memo key for later decoding of memos and the active key for being able to spend funds of that accounts::

```
>>> get_account <account-name>
{
  [...]
  "active": {
    "key_auths": [
      "<active_key>",
      1
    ],
    [...]
  }
  "memo_key": "<memo_key>",
  [...]
  "statistics": "<statistics>",
  [...]
}
```

We now need to export the corresponding private keys from BitShares 1.0 and import the keys into the `cli_wallet`::

```
BitShares 1: >>> wallet_dump_private_key <memo_key>
    "<memo_private_key>"
BitShares 1: >>> wallet_dump_private_key <active_key>
    "<active_private_key>"
```

Import the active key into BitShares 2 wallet:

```
BitShares 2: >>> import_key <account-name> <active_private_key>
```

This gives access to the funds stored in <account-name>. We will need the memo private key later when watching deposits.

Claiming BitShares 1.0 funds

We now describe how to claim your funds from the Bitshares 1 blockchain so you can use them in BitShares 2.

For **Coldstorage** and plain private keys, we recommend to use:

```
>>> import_balance <accountname> <private_key> false
```

to import all balances that are locked in the private key into the account named <accountname>. As long as the last argument is `false` the transaction will only be printed for audit and not be broadcasted or executed. **Only** after changing `false` to `true` will the balances be claimed!

For your hot wallet (or any other active wallet running in the BitShares 1 client) we recommend to use the GUI to claim your funds from hot wallet as described [here](#).

Watching Deposits with Python

For watching deposits, we recommend pybitshares' *Notify* module. The full documentation is available on [pybitshares.com](#).

Executing Transfers for Withdrawals

For transferring funds, we recommend pybitshares. This python module enables all features required to operate on/with BitShares. The full documentation is available on [pybitshares.com](#).

Supporting Features

BitShares 2.0 offers some features that will make your integration easier and more secure:

User Issued Assets

Any participant can create and issue new (user-issued) assets. The potential use cases for so called user-issued assets (UIA) are innumerable. On the one hand, UIAs can be used as simple event tickets deposited on the customers mobile phone to pass the entrance of a concert. On the other hand, they can be used for crowd funding, ownership tracking or even to sell equity of a company in form of stock.

Obviously, the regulations that apply to each kind of token vary widely and are often different in every jurisdiction. Hence, BitShares comes with tools that allow issuers to remain compliant with all applicable regulations when issuing

assets assuming regulators allow such assets in the first place. We will discuss the tools and optional administrative rights given to the issuers of a given UIA and provide a subset of possible use-cases in more detail.

Whitelists and Blacklists

Some 3rd party service providers may want to select which customers are allowed to hold their assets , e.g. after verified their identity for KYC/AML. Those services can use so called *whitelists* (or, alternatively, *blacklists*) of their assets that will prevent unauthorized participants to use this particular asset.

In BitShares 2.0, account names (life-time members only) and also user-issued assets have their individual whitelists. Hence, if you issue an IOU on the blockchain, you can define who can hold and trade your tokens, if you wish.

User whitelists on contrast can be used by independent KYC/AML providers to state proper verification. An asset issuer may then use those providers to outsource identity verification completely.

Hierarchical Corporate Accounts

BitShares designs permissions around people, rather than around cryptography, making it easy to use. Every account can be controlled by any weighted combination of other accounts and private keys. This creates a hierarchical structure that reflects how permissions are organized in real life, and makes multi-user control over funds easier than ever. Multi-user control is the single biggest contributor to security, and, when used properly, it can virtually eliminate the risk of theft due to hacking.

In BitShares 2.0 there is no real need for cold storage solutions. Just construct your spending authority using a set of people in your company such as CFO, CTO, and members of accounting and freely chose how much they can do.

Using White- and Black-lists

White- Black-Lists exist for assets and for (lifetime member) accounts. While the latter can only be used to cast an opinion about another account, lists for assets serve a very practical need.

User Whitelists

Any live-time member can cast an opinion about other accounts using white- and black-lists. They **do not** prevent anyone from interacting with your account but serve as a basis for *list authorities*.

Examples

A user `white` can be added to the white-list of account `provider` with::

```
>>> whitelist_account provider white white_listed true
```

In contrast a `black` user can be added to its blacklist with::

```
>>> whitelist_account provider black black_listed true
```

Both can be removed from their lists with::

```
>>> whitelist_account provider black no_listing true  
>>> whitelist_account provider white no_listing true
```

Definition

White- and Black-listing of accounts works with the following API call:

```
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

It expects a `new_listing_status` from

```
enum graphene::chain::account_whitelist_operation::account_listing
Values:
no_listing = 0x0
    No opinion is specified about this account.

white_listed = 0x1
    This account is whitelisted, but not blacklisted.

black_listed = 0x2
    This account is blacklisted, but not whitelisted.

white_and_black_listed = white_listed | black_listed
    This account is both whitelisted and blacklisted.
```

Asset User Whitelists

Asset User white- and black-lists serve the need for companies to restrict service to a subset of accounts. For instance, a fiat gateway may require to follow KYC/AML regulations and can hence only deal with those customers that have been verified accordingly. If the issuer of an user-issued asset desires, he may set a restriction so that only users on the white-list (and/or **not** on the blacklist) are allowed to hold his token.

Instead of putting all verified accounts into the respective asset's white-list directly, BitShares 2.0 allows to define one or several white-list *authorities*. In practice, the white- and black-lists of these accounts are combined and serve as white- and black-lists for the asset.

This allows for easy out-sourcing of KYC/AML verification to 3rd-party providers.

Note: By removing a user from the whitelist, funds can effectively be frozen.

Example

Let's assume user `alice` wants to own a gateways IOUs called `G.USD` which are restricted by a whitelists. Before being able to own `G.USD`, `alice` needs to be white-listed by one of the authorities of `G.USD`.

Defining an asset's list authorities

We now define the authorities (i.e. accounts) that define the white- and blacklist of the asset `G.USD`. We add `g-issuer` and `kycprovider` to the white- and black-list::

```
>>> update_asset G.USD "" "{blacklist_authorities:[g-issuer, kycprovider], whitelist_
->authorities:[g-issuer, kycprovider], flags:white_list}" true
```

Note: The third argument for `update_asset` replaces the existing settings. Make sure to have all desired settings present.

Adding `alice` to a whitelist

Let's assume the only authority is the issuer `g-issuer` himself for simplicity. The issuer now needs to add `alice` to `g-issuer`'s account whitelist::

```
>>> whitelist_account g-issuer alice white_listed true
```

Definition

White- and Black-listing of assets works with the following API call:

```
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

struct `graphene::chain::asset_options`

The `asset_options` struct contains options available on all assets in the network.

Note Changes to this struct will break protocol compatibility

Public Functions

`void validate() const`

Perform internal consistency checks.

Exceptions

- `fc::exception`: if any check fails

Public Members

`share_type max_supply = GRAPHENE_MAX_SHARE_SUPPLY`

The maximum supply of this asset which may exist at any given time. This can be as large as `GRAPHENE_MAX_SHARE_SUPPLY`

`uint16_t market_fee_percent = 0`

When this asset is traded on the markets, this percentage of the total traded will be exacted and paid to the issuer. This is a fixed point value, representing hundredths of a percent, i.e. a value of 100 in this field means a 1% fee is charged on market trades of this asset.

`share_type max_market_fee = GRAPHENE_MAX_SHARE_SUPPLY`

Market fees calculated as `market_fee_percent` of the traded volume are capped to this value.

`uint16_t issuer_permissions = UIA_ASSET_ISSUER_PERMISSION_MASK`

The flags which the issuer has permission to update. See `asset_issuer_permission_flags`.

`uint16_t flags = 0`

The currently active flags on this permission. See `asset_issuer_permission_flags`.

`price core_exchange_rate`

When a non-core asset is used to pay a fee, the blockchain must convert that asset to core asset in order to accept the fee. If this asset's fee pool is funded, the chain will automatically deposit fees in this asset to its accumulated fees, and withdraw from the fee pool the same amount as converted at the core exchange rate.

`flat_set<account_id_type> whitelistAuthorities`

A set of accounts which maintain whitelists to consult for this asset. If `whitelistAuthorities` is non-empty, then only accounts in `whitelistAuthorities` are allowed to hold, use, or transfer the asset.

`flat_set<account_id_type> blacklistAuthorities`

A set of accounts which maintain blacklists to consult for this asset. If `flags & white_list` is set, an account may only send, receive, trade, etc. in this asset if none of these accounts appears in its `account_object::blacklisting_accounts` field. If the account is blacklisted, it may not transact in this asset even if it is also whitelisted.

```
flat_set<asset_id_type> whitelist_markets
    defines the assets that this asset may be traded against in the market

flat_set<asset_id_type> blacklist_markets
    defines the assets that this asset may not be traded against in the market, must not overlap whitelist

string description
    data that describes the meaning/purpose of this asset, fee will be charged proportional to size of description.

enum graphene::chain::asset_issuer_permission_flags
    Values:

charge_market_fee = 0x01
    an issuer-specified percentage of all market trades in this asset is paid to the issuer

white_list = 0x02
    accounts must be whitelisted in order to hold this asset

override_authority = 0x04
    issuer may transfer asset back to himself

transfer_restricted = 0x08
    require the issuer to be one party to every transfer

disable_force_settle = 0x10
    disable force settling

global_settle = 0x20
    allow the bitasset issuer to force a global settling this may be set in permissions, but not flags

disable_confidential = 0x40
    allow the asset to be used with confidential transactions

witness_fed_asset = 0x80
    allow the asset to be fed by witnesses

committee_fed_asset = 0x100
    allow the asset to be fed by the committee
```

Asset Market Whitelists

An issuer of an user-issued-asset may want to restrict trading partners for his assets for legal reasons. For instance, a gateway for US dollar may not be allowed to let his customers trade USD against CNY because additional licenses would be required. Hence, in BitShares 2.0 we let issuers chose to restrict trading partners with white- and black-lists.

Example

A gateway with IOU G.USD that wants to prevent his customers from trading G.USD against bitCNY can do so by adding bitCNY to the blacklist of G.USD by issuing::

```
>>> update_asset G.USD "" "{blacklist_markets:[CNY]}" true
```

Alternatively, if an issuer may want to only open the market G.USD : bitUSD with his asset, he can do so as well with::

```
>>> update_asset G.USD "" "{whitelist_markets:[USD]}" true
```

Note: The third argument for `update_asset` replaces the existing settings. Make sure to have all desired settings present.

Definition

Asset Market white-lists work with the following API call:

```
signed_transaction graphene::wallet::wallet_api::update_asset(string      symbol,      op-
                                                               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

```
struct graphene::chain::asset_options
```

The `asset_options` struct contains options available on all assets in the network.

Note Changes to this struct will break protocol compatibility

Public Functions

```
void validate() const
```

Perform internal consistency checks.

Exceptions

- `fc::exception`: if any check fails

Public Members

```
share_type max_supply = GRAPHENE_MAX_SHARE_SUPPLY
```

The maximum supply of this asset which may exist at any given time. This can be as large as `GRAPHENE_MAX_SHARE_SUPPLY`

```
uint16_t market_fee_percent = 0
```

When this asset is traded on the markets, this percentage of the total traded will be exacted and paid to the

issuer. This is a fixed point value, representing hundredths of a percent, i.e. a value of 100 in this field means a 1% fee is charged on market trades of this asset.

share_type **max_market_fee** = GRAPHENE_MAX_SHARE_SUPPLY

Market fees calculated as *market_fee_percent* of the traded volume are capped to this value.

uint16_t issuer_permissions = UIA_ASSET_ISSUER_PERMISSION_MASK

The flags which the issuer has permission to update. See *asset_issuer_permission_flags*.

uint16_t flags = 0

The currently active flags on this permission. See *asset_issuer_permission_flags*.

price core_exchange_rate

When a non-core asset is used to pay a fee, the blockchain must convert that asset to core asset in order to accept the fee. If this asset's fee pool is funded, the chain will automatically deposit fees in this asset to its accumulated fees, and withdraw from the fee pool the same amount as converted at the core exchange rate.

flat_set<account_id_type> whitelistAuthorities

A set of accounts which maintain whitelists to consult for this asset. If *whitelistAuthorities* is non-empty, then only accounts in *whitelistAuthorities* are allowed to hold, use, or transfer the asset.

flat_set<account_id_type> blacklistAuthorities

A set of accounts which maintain blacklists to consult for this asset. If *flags & white_list* is set, an account may only send, receive, trade, etc. in this asset if none of these accounts appears in its *account_object::blacklisting_accounts* field. If the account is blacklisted, it may not transact in this asset even if it is also whitelisted.

flat_set<asset_id_type> whitelistMarkets

defines the assets that this asset may be traded against in the market

flat_set<asset_id_type> blacklistMarkets

defines the assets that this asset may not be traded against in the market, must not overlap whitelist

string description

data that describes the meaning/purpose of this asset, fee will be charged proportional to size of description.

enum graphene::chain::asset_issuer_permission_flags

Values:

charge_market_fee = 0x01

an issuer-specified percentage of all market trades in this asset is paid to the issuer

white_list = 0x02

accounts must be whitelisted in order to hold this asset

override_authority = 0x04

issuer may transfer asset back to himself

transfer_restricted = 0x08

require the issuer to be one party to every transfer

disable_force_settle = 0x10

disable force settling

global_settle = 0x20

allow the bitasset issuer to force a global settling this may be set in permissions, but not flags

disable_confidential = 0x40

allow the asset to be used with confidential transactions

witness_fed_asset = 0x80

allow the asset to be fed by witnesses

```
committee_fed_asset = 0x100
allow the asset to be fed by the committee
```

Securing Funds

Since BitShares 2.0 offers hierarchical corporate accounts to secure your account name and your funds. With this, you can build hierarchies of accounts (so called *authorities*) and a condition that has to be fulfilled in order for a transaction to become valid.

(under construction)

Libraries

Supporting Libraries

General Python Library

pybitshares offers many modules specifically for BitShares. It is well documented and has it's own documentation page.

Python Module for the DEX

pybitshares offers a Trading module specifically for the decentralized exchange (DEX). It is well documented and has it's own documentation page.

API Guide

This page serves as a comprehensive reference for the Graphene API and its blockchain, such as *BitShares 2.0* and *MUSE*.

APIs are separated into two categories, namely

- the **Blockchain API** which is used to query blockchain data (account, assets, trading history, etc.) and
- the **Wallet API** which has your private keys loaded and is required when interacting with the blockchain with new transactions.

Note: In order to interface with *the wallet*, you need to run the *CLI Wallet*. Neither the **light-wallet**, nor the **hosted web wallet** will provide you with an API.

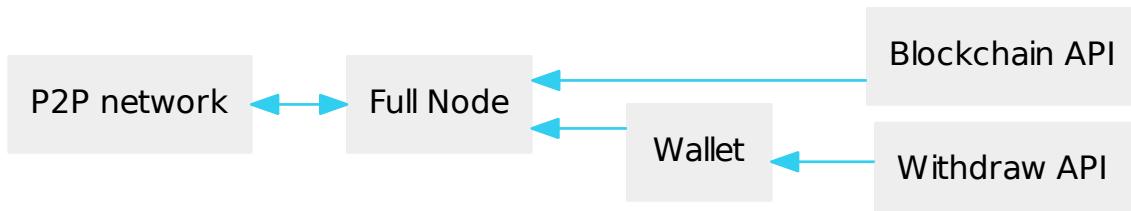
In contrast to many existing ecosystems, there is no centralized service that lets you access private API calls after successful authentication. Instead, you run your wallet (and optionally a full node) **locally** and are with **your own API service provider**. This obviously has the advantage that you don't need to give access to your funds to any third party but has the slight disadvantage that you need to run a local *wallet application*, that however does not download the whole blockchain for verification. If you run a sensitive business, we recommend to also run a local full node to download and verify the blockchain and interface your wallet with your local full node.

This page will give you a detailed description of both API categories, the Remote Procedure Calls and Websockets, and will give an introduction to many available calls.

Interfacing with Graphene

The set of available calls depends on whether you connect to a full node (`witness_node`) or the wallet (`cli_wallet`). Both support RPC-JSON. The full node also supports the websocket protocol with notifications.

Which blockchain network you connect to (BitShares, MUSE, ..) depends on the configuration of the full node and the wallet. If you run a full node, we recommend to connect your wallet to your local full node even though it could be connected to any other public full node as well.



For sensitive businesses that want to ensure that deposits are irreversible, we recommend the use of the [High Security Setup](#). That contains a *delayed node* to pass only irreversible transactions to the API.

Note: All API calls are formated in JSON and return JSON only.

Wallet API

This chapter introduces the calls available via wallet API. If you have not set up your wallet yet, you can find more information on the [CLI Wallet](#) and the [CLI Wallet FAQ](#) pages.

Remote Procedure Calls

Prerequisites

This page assumes that you either have a full node or a wallet running and listening to port 8090, locally.

Note: The set of available commands depends on application you connect to.

Call Format

In Graphene, RPC calls are state-less and accessible via regular JSON formated RPC-HTTP-calls. The correct structure of the JSON call is

```
{  
  "jsonrpc": "2.0",
```

```

"id": 1
"method": "get_accounts",
"params": [["1.2.0", "1.2.1"]],
}

```

The `get_accounts` call is available in the Full Node's database API and takes only one argument which is an array of account ids (here: `["1.2.0", "1.2.1"]`).

Example Call with *curl*

Such a call can be submitted via `curl`:

```
curl --data '{"jsonrpc": "2.0", "method": "get_accounts", "params": [["1.2.0", "1.2.1"]], "id": 1}' http://127.0.0.1:8090/rpc
```

Successful Calls

The API will return a properly 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
  "error": {
    "data": {
      "code": error-code,
      "name": " .. name of exception .."
      "message": " .. message of exception ..",
      "stack": [ .. stack trace .. ],
    },
    "code": 1,
  },
}
```

Remarks

Wallet specific commands, such as `transfer` and market orders, are only available if connecting to `cli_wallet` because only the wallet has the private keys and signing capabilities and some calls will only execute if the wallet is unlocked.

The full node offers a set of API(s), of which only the database calls are available via RPC. Calls that are restricted by default (i.e. `network_node_api`) or have been restricted by configuration are not accessible via RPC because a statefull protocol (websocket) is required for login.

Blockchain API(s)

The blockchain API can be used to obtain any kind of data stored in the blockchain. Besides data stores in the blockchain itself (blocks, transactions, etc. ...), higher level objects (such as accounts, balances, etc. ...) can be retrieved through the full node's database.

It is not required to run a local full node if you want to query a particular blockchain or database, but you can also query any existing public node for information.

Websocket Calls & Notifications

Prerequisites

This page assumes that you have a full node running and listening to port 8090, locally.

Note: If you also want to run a wallet, please pick reasonable different ports and make sure you do not try to call methods at the wallet that are only available to the blockchain API.

Call Format

In Graphene, Websocket calls are stateful and accessible via regular JSON formated websocket connection. The correct structure of the JSON call is

```
{  
  "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 (see more details below).

Example Call with wscat

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://127.0.0.1:8090  
> {"id":1, "method":"call", "params": [0, "get_accounts", [[["1.2.0"]]]]}
```

Successful Calls

The API will return a properly JSON formated 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
  "error": {
    "data": {
      "code": error-code,
      "name": " .. name of exception .."
      "message": " .. message of exception ..",
      "stack": [ .. stack trace .. ],
    },
    "code": 1,
  },
}
```

Requesting API access

The Graphene 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

Find below a list of available APIs:

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](#).

Table of Contents

- *Database API*
 - *Objects*

- *Subscriptions*
- *Blocks and transactions*
- *Globals*
- *Keys*
- *Accounts*
- *Balances*
- *Assets*
- *Markets / feeds*
- *Witnesses*
- *Committee members*
- *Workers*
- *Votes*
- *Authority / Validation*
- *Proposed Transactions*
- *Blinded balances*

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

Subscriptions

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

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.

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

```
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
    trans-
    ac-
    tion_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.

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.
```

Keys

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

Accounts

```
vector<optional<account_object>> graphene::app::database_api::get_accounts (const vec-
tor<account_id_type>
&ac-
count_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
vec-
tor<string>
&names_or_ids,
bool
sub-
scribe)
```

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
ac-
count_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
vec-
tor<string>
&ac-
count_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.

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 ac-  
count_id) const
```

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> &sym-  
bols_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

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,           as-
set_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<call_order_object> graphene::app::database_api::get_call_orders(asset_id_type a,
uint32_t limit)
const
```

Get call orders in a given asset.

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

Parameters

- a: ID of asset being called
- 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
ac-
count_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 vari-  
ant&  
> 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_volume graphene::app::database_api::get_24_volume (const string &base, const  
string &quote) const
```

Returns the 24 hour volume for the market assetA:assetB.

Return The market 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 trasactions to retrieve, capped at 100
- start: Start time as a UNIX timestamp

Witnesses

```
vector<optional<witness_object>> graphene::app::database_api::get_witnesses (const
                                                                      vec-
                                                                      tor<witness_id_type>
                                                                      &wit-
                                                                      ness_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
                                                                           ac-
                                                                           count)
                                                                           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.

Committee members

```
vector<optional<committee_member_object>> graphene::app::database_api::get_committee_members (const  
vec-  
tor<commi-  
&com-  
mit-  
tee_membe  
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 (ac  
ac-  
cc  
co
```

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
```

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

Workers

```
vector<worker_object> graphene::app::database_api::get_workers_by_account (account_id_type  
ac-  
count)  
const
```

WORKERS.

Return the worker objects associated with this account.

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.

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 if 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

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.

Blinded balances

```
vector<blinded_balance_object> graphene::app::database_api::get_blinded_balances (const flat_set<commitment_type> &com- mit- ments) const
```

Return the set of blinded balance objects by commitment ID

Account History API

The history API is available from the full node via websockets.

If you have not set up your websockets connection, please read [this article](#).

Table of Contents

- *Account History API*
 - *Account History*
 - *Market History*

Account History

```
vector<operation_history_object> graphene::app::history_api::get_account_history(account_id_type
    ac-
    count,
    op-
    era-
    tion_history_id_type
    stop
    =
    op-
    era-
    tion_history_id_type
    () ,
    un-
    signed
    limit
    =
    100,
    op-
    era-
    tion_history_id_type
    start
    =
    op-
    era-
    tion_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

Market History

```
vector<order_history_object> graphene::app::history_api::get_fill_order_history(asset_id_type
    a,
    as-
    set_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
```

Network Broadcast API

The network broadcast API is available from the full node via websockets.

If you have not set up your websockets connection, please read [this article](#).

Table of Contents

- *Network Broadcast API*
 - *Transactions*
 - *Block*

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.

Block

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

Network Nodes API

The network node API is available from the full node via websockets.

If you have not set up your websockets connection, please read [this article](#).

Table of Contents

- *Network Nodes API*
 - *Obtain Network Information*
 - *Change Network Settings*

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.
```

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

1. Login

The first thing we need to do is to *login*:

```
> {"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 the quotes, accordingly. Furthermore, you should verify, that the result give positive confirmation about your login.

2. Requesting Access to an API

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

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

3. 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!

4. 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"}]]}
```

Database Notifications

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

We have the following subscriptions available:

- **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(int identifier)**: Notifications for incoming unconfirmed transactions.
- **set_block_applied_callback(blockid)**: Gives a notification whenever the block blockid is applied to the blockchain.
- **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(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.

Let's first get a global scubscription callback to disctinguish 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 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": ... }
      ]
    ],
  ]
}
```

Example Session

Here is an example of a full session::

```
> {"method": "call", "params": [1, "login", ["", ""]], "id": 2}
< {"id":2,"result":true}
> {"method": "call", "params": [1, "database", []], "id": 3}
< {"id":3,"result":2}
> {"method": "call", "params": [1, "history", []], "id": 4}
< {"id":4,"result":3}
> {"method": "call", "params": [2, "set_subscribe_callback", [5, false]], "id": 6}
< {"id":6,"result":null}
> {"method": "call", "params": [2, "get_objects", [["2.1.0"]]], "id": 7}
(plenty of data coming in from this point on)
```

Blockchain API

The Graphene full node distinguishes several different APIs that can be accessed as described [the websockets documentation](#).

Crypto API

The crypto API is available from the full node via websockets.

If you have not set up your websockets connection, please read [this article](#).

Table of Contents

- [Crypto API](#)

- *Blinding and Un-Blinding*
- *Rage Proofs*
- *Verification*

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)
```

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)
```

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_
&com-
mit,
const
std::vector<char>
&proof)
```

Access to some APIs may be **restricted** and requires login with username and passphrase. More detailed description about this can be found on the access page.

Blockchain Objects and their Identifiers

In contrast to many other projects, the Graphene technology distinguishes different kinds of objects, in the protocol and implementation space.

In the protocol space, there are raw objects such as, accounts, assets, committee members as well as orders, proposals and balances. The implementation space is used to gain access to higher abstraction layers for instance content of the current database state (these include, current global blockchain properties, dynamic asset data, transaction histories as well as account statistics and budget records).

Merchants make use of the currency-denominated assets of a Graphene network (e.g. BitShares). Similar to traditional payment solutions they let their customers pay using bitUSD, bitEUR, or any other *stable* blockchain asset.

Merchants

We here illustrate the steps necessary to securely operate as merchant. Merchants take funds from customers on the blockchain and deliver a good. Hence, a merchant should monitor the blockchain and be notified on incoming transactions. Thanks to (encrypted) transaction memos attachable to each transfer the merchant can easily distinguish different customers.

For exchanges we recommend to also read [What is Different in BitShares](#) and [Often used API Calls](#).

Protocols/API

Wallet Login Protocol

The idea behind the login protocol is to allow another party to verify that you are the owner of a particular account. Traditionally login is performed via a password that is sent to the server, but this method is subject to [Phishing Attacks](#). Instead of a password, Graphene uses a cryptographic challenge/response to verify that a user controls a particular account.

For the purpose of this document, we will assume <https://merchant.org> is the service that will be logged into and that <https://wallet.org> is the wallet provider that will be assisting the user with their login.

Step 1 - Merchant Login Button

The merchant must provide the user with a login button that links to [https://wallet.org/login#\\${args}](https://wallet.org/login#${args}) where \${args} is a JSON object containing following information and serialized as described below:

```
{  
  "onetimekey" : "${SERVER_PUBLIC_KEY}",  
  "account" : "${OPT_ACCOUNT_NAME}",  
  "callback" : "https://merchant.org/login_callback"  
}
```

The merchant server will need to save the \${SERVER_PRIVATE_KEY} associated with the \${SERVER_PUBLIC_KEY} in the user's web session in order to verify the login.

Step 2 - Compress your JSON representation

Using LZMA-JS library to compress the JSON into a binary array. This will be the most compact form of the data. After running the compression the example JSON was reduced to 281 bytes from 579 bytes.

Step 3 - Convert to Base58

Using the bs58 library encode the compressed data in base58. Base58 is URL friendly and size efficient. After converting to base58 the string will be 385 characters which can easily be passed in a URL and easily support much larger invoices.

Step 4 - Wallet Confirmation

When the user loads `https://wallet.org/login#${args}` they will be prompted to confirm the login request by selecting an account that they wish to login with. If “account” was specified in the \${args} then it will default to that account.

After the account is identified enough keys to authorize a account must participate in the login process in the following way.

The wallet generates a WALLET_ONETIMEKEY and derives a shared secret with the SERVER_PUBLIC_KEY provided by the `https://merchant.org` via \${args}. This shared secret is a provably “random” 512 bits of data that is only known to the wallet at this point in time. The wallet then gathers signatures on the shared secret from enough keys to authorize the account. In the simple case this will be a single signature, but in more complex cases multi-factor authentication may be required.

After gathering all of the signatures the wallet redirects the user to `https://merchant.org/login_callback?a=${result}` where result is an encoded JSON object containing the following information:

```
{  
  "account": "Graphene Account Name",  
  "server_key": "${SERVER_PUBLIC_KEY}",  
  "account_key": "${WALLET_ONETIMEKEY}",  
  "signatures" : [ "SIG1", "SIG2", ... ]  
}
```

Step 5 - Server Verifies Authority

Upon receiving the result from the wallet, `https://merchant.org` will lookup \${SERVER_PRIVATE_KEY} in the user's session data and then combine it with \${WALLET_ONETIMEKEY} to generate the *shared secret* that was used by the wallet. Once this shared secret has been recovered, it can be used to recover the public keys that correspond to the provided signatures.

The last step is to verify that the public keys provided by the signatures are sufficient to authorize the account given the current state of the graphene blockchain. This can be achieved using the witness API call::

```
verify_account_authority( account_name_or_id, [public_keys...] )
```

The `verify_account_authority` call will return `true` if the provided keys have sufficient authority to authorize the account, otherwise it will return `false`

Wallet Merchant Protocol

The purpose of this protocol is to enable a merchant to request payment from the user via a hosted wallet provider or via a browser plugin. We will assume that the wallet is hosted at `https://wallet.org` and that the merchant is hosted at `https://merchant.org`.

Privacy Concerns

The goal of this protocol is to maintain user and merchant privacy from the wallet provider which should never have direct access to the invoice data.

To securely pass data from `https://merchant.org` to the javascript wallet hosted at `https://wallet.org`, the data will have to be passed after the `#`. Assuming the wallet provider is not serving up pages designed to compromise your privacy, only your web browser will have access to the invoice data.

Step 1: Define your Invoice via JSON

This invoice provides all of the data needed by the wallet to display an invoice to the user.

```
{
  "to" : "merchant_account_name",
  "to_label" : "Merchant Name",
  "memo" : "Invoice #1234",
  "currency": "BTS",
  "line_items" : [
    { "label" : "Something to Buy", "quantity": 1, "price" : "1000.00 SYMBOL" },
    { "label" : "10 things to Buy", "quantity": 10, "price" : "1000.00 SYMBOL" },
    { "label" : "User Specified Price", "quantity": 1, "price" : "CUSTOM SYMBOL" }
  ],
  { "label" : "User Asset and Price", "quantity": 1, "price" : "CUSTOM" }
],
  "note" : "Something the merchant wants to say to the user",
  "callback" : "https://merchant.org/complete"
}
```

By itself this data is 579 characters which after URL encoding is 916 characters, with a 2000 character limit this approach doesn't scale as well as we would like.

Step 2: Compress your JSON representation

Using [LZMA-JS](#) library to compress the JSON into a binary array. This will be the most compact form of the data. After running the compression the example JSON was reduced to 281 bytes from 579 bytes.

Step 3: Convert to Base58

Using the `bs58` library encode the compressed data in base58. Base58 is URL friendly and size efficient. After converting to base58 the string will be 385 characters which can easily be passed in a URL and easily support much larger invoices.

Step 4: Pass to Wallet

Once the Base58 data is known, it can be passed to the wallet with the following URL::

```
https://wallet.org/#/invoice/BASE58BLOB
```

Step 5: Receive Callback from Wallet

After the wallet has signed a transaction, broadcast it, and gotten confirmation from <https://wallet.org> that the transaction was included in block 12345 as transaction 4 wallet will direct the user to <https://merchant.org/complete?block=12345&trx=4>

The merchant will then request that transaction from <https://wallet.org/api?block=12345&trx=4> which will respond with the transaction that was included in the blockchain. The merchant will decrypt the memo from the transaction and use memo content to confirm payment for the invoice.

Step 6: Payment Complete

At this point the user has successfully made a payment and the merchant has verified the payment has been received without having to maintain a full node.

Example Python script

```
import json
import lzma
from graphenebase.base58 import base58encode, base58decode
from binascii import hexlify, unhexlify

invoice = {
    "to": "bitshareseurope",
    "to_label": "BitShares Europe",
    "currency": "EUR",
    "memo": "Invoice #1234",
    "line_items": [
        {"label": "Something to Buy", "quantity": 1, "price": "10.00"},
        {"label": "10 things to Buy", "quantity": 10, "price": "1.00"}
    ],
    "note": "Payment for reading awesome documentation",
    "callback": "https://bitshares.eu/complete"
}

compressed = lzma.compress(bytes(json.dumps(invoice), 'utf-8'), format=lzma.FORMAT_ALONE)
b58 = base58encode(hexlify(compressed).decode('utf-8'))
url = "https://bitshares.openledger.info/#/invoice/%s" % b58
```

```
print(url)
```

Libraries

Traders make use of the API provided to interact with a Graphene network (e.g. BitShares) and provide market makers and liquidity. The APIs can be easily used to implement automated robots for trading algorithms.

Traders

Blockchain-based decentralized exchanges (DEX) are slightly different to centralized exchanges and as a consequence, dealing with the DEX programmatically via APIs differs from centralized approaches as well. However, our developers have put quite some efforts into making the DEX as easy to use as their centralized counterparts and offer an API to the **public exchange data** that is very similar. However, **private exchange APIs** are different due to the fact that no entity except yourself can access your funds. For this reason, trading in the DEX requires that you have the private key to your account installed in an application that can construct and sign the corresponding transactions for you. One of these applications is the *CLI Wallet* which, after installation and configuration, offers a **your own private API**.

For exchanges we recommend to also read [What is Different in BitShares](#) and [Often used API Calls](#).

Public API

The best way to get public data on markets is via websocket connection to a public full-node which provides traders with

- a ticker
- order books
- trade history
- and more.

BitShares Public Full Nodes:

wss://bitshares.openledger.info/ws
wss://bitshares.dacplay.org:8089/ws
wss://dele-puppy.com/ws

A detailed description of how to interface with Graphene-based blockchain (e.g. BitShares) and a list of available calls can be found here:

Private API

As briefly mentioned above, trading in the DEX programmatically requires that you run your own *CLI Wallet*. The following tutorials gives a brief introduction on how to use the CLI wallet and configure it properly so that it can be used as a **private API server**:

Howto prepare a CLI wallet for trading

The CLI wallet is used to interact with the blockchain. Everything that adds new data to the blockchain requires a signature from a private key. These signed transactions can be produced by the CLI wallet.

Download and Installation

For most graphene-based blockchain projects, there should be a separated `cli-tools` download available for many platforms. If not, the CLI wallet can be compile manually via

```
make cli_wallet
```

Executing the cli-wallet

All it takes for the cli-wallet to run is a trusted **public API server** to interface with the blockchain. These public API servers are run by businesses and *individuals*:

BitShares Public Full Nodes:

wss://bitshares.openledger.info/ws
wss://bitshares.dacplay.org:8089/ws
wss://dele-puppy.com/ws

We can let the wallet know which API server to use by:

```
./programs/cli_wallet/cli_wallet --server-rpc-endpoint=<URL> -H 127.0.0.1:8092 -r 127.  
→ 0.0.1:8093
```

The cli-wallet will open two RPC ports so that you can interface your application with it. You have the choices of

- websocket RPC via the `-r` parameter, and
- HTTP RPC via the `-H` parameter:

The command above will open the cli-wallet and, unless you already have a local wallet, will ask you to provide a passphrase for your local wallet. Once a wallet has been created (default wallet file is `wallet.json`), it will prompt with

```
locked >>>
```

In order for the wallet to be able to sign trading orders, you need to unlock it by providing the passphrase:

Note: The passphrase is given in clear text and is echoed by the wallet!

```
locked >>> unlock "supersecretpassphrase"  
null  
unlocked >>>
```

After this point, you can issue *any command available to the cli-wallet* or construct your *own transaction manually*.

You can get more detailed information either by pressing `Tab`, twice, or by issuing `help`. Detailed explanations for most calls are available via

```
unlocked >> gethelp <command>
```

Note: Many calls have a obligatory `broadcast`-flag as last argument. If this flag is `False`, the wallet will construct and sign, but **not** broadcast the transaction. This can be very useful for a cold storage setup or to verify transactions.

After installation and configuration of the private API, we can use RPC to create orders, cancel orders, create and adjust call orders, and more. The CLI wallet offers a wide range of calls that can be used to mange your account and trade in the DEX:

Libraries

Libraries have been developed that simply the interaction with both, the full node and the CLI wallet and make interactions with the blockchain and the DEX very easy:

For **businesses**, a Graphene network (e.g. BitShares) can be used to issue their own token and let customers trade their shares either against predefined assets, or freely against any other asset. This guide serves the purpose to introduce the possibilities of

Businesses

Since BitShares allows everyone to establish their coin or business on the same blockchain, they can benefit from synergies. Every customer of a competing business or coin that is part of BitShares already has an account capable of dealing with your business or holding your coins. Besides a shared user base, we also have shared markets. You can think of shared markets like a market that trades BitStampUSD versus KrakenUSD, both being IOUs for fiat USD at the corresponding platform. Suddenly, every user of one exchange can be a potential user for a competing business, not just for the exchange business.

Libraries

CHAPTER 4

API Guide

CHAPTER 5

Development

5.1 Development Guide

5.1.1 Blockchain Specifications

The technical specifications of blockchain objects and their serialization is part of the following sections.

5.1.2 Namespaces

Graphene::App

```
namespace graphene::app
```

Unnamed Group

```
template <typename T>
T jsonify(const string &s)
```

Some useful tools for boost::program_options arguments using vectors of JSON strings

Functions

```
vector<account_id_type> get_relevant_accounts(const object *obj)
```

```
void operation_get_impacted_accounts(const operation &op, flat_set<account_id_type>
&result)
```

```
void transaction_get_impacted_accounts(const transaction &tx,
flat_set<account_id_type> &result)
```

```
void operation_get_impacted_accounts(const graphene::chain::operation &op,
                                         fc::flat_set<graphene::chain::account_id_type>
                                         &result)

void transaction_get_impacted_accounts(const graphene::chain::transaction &tx,
                                         fc::flat_set<graphene::chain::account_id_type>
                                         &result)

class abstract_plugin
#include <plugin.hpp> Subclassed by graphene::app::plugin
```

Public Functions

virtual void plugin_initialize(const boost::program_options::variables_map &options) = 0

Perform early startup routines and register plugin indexes, callbacks, etc.

Plugins MUST supply a method initialize() which will be called early in the application startup. This method should contain early setup code such as initializing variables, adding indexes to the database, registering callback methods from the database, adding APIs, etc., as well as applying any options in the options map

This method is called BEFORE the database is open, therefore any routines which require any chain state MUST NOT be called by this method. These routines should be performed in startup() instead.

Parameters

- *options*: The options passed to the application, via configuration files or command line

virtual void plugin_startup() = 0

Begin normal runtime operations.

Plugins MUST supply a method startup() which will be called at the end of application startup. This method should contain code which schedules any tasks, or requires chain state.

virtual void plugin_shutdown() = 0

Cleanly shut down the plugin.

This is called to request a clean shutdown (e.g. due to SIGINT or SIGTERM).

virtual void plugin_set_app(application *a) = 0

Register the application instance with the plugin.

This is called by the framework to set the application.

**virtual void plugin_set_program_options(boost::program_options::options_description &command_line_options,
 boost::program_options::options_description &config_file_options) = 0**

Fill in command line parameters used by the plugin.

This method populates its arguments with any command-line and configuration file options the plugin supports. If a plugin does not need these options, it may simply provide an empty implementation of this method.

Parameters

- *command_line_options*: All options this plugin supports taking on the command-line
- *config_file_options*: All options this plugin supports storing in a configuration file

class application

#include <application.hpp>

Public Members

```
boost::signals2::signal<void ()> syncing_finished
    Emitted when syncing finishes (is_finished_syncing will return true)
```

```
class block_api
    #include <api.hpp> Block api.
```

```
class database_api
    #include <database_api.hpp> The database_api class implements the RPC API for the chain database.
```

This API exposes accessors on the database which query state tracked by a blockchain validating node. This API is read-only; all modifications to the database must be performed via transactions. Transactions are broadcast via the *network_broadcast_api*.

Public Functions

```
fc::variants 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

```
void cancel_all_subscriptions ()
```

Stop receiving any notifications.

This unsubscribes from all subscribed markets and objects.

```
optional<block_header> 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

```
map<uint32_t, optional<block_header>> get_block_header_batch (const vector<uint32_t> &block_nums) const
```

Retrieve multiple block header by block numbers.

Return array of headers of the referenced blocks, or null if no matching block was found

Parameters

- *block_num*: vector containing heights of the block whose header should be returned

```
optional<signed_block> 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

```
processed_transaction get_transaction (uint32_t block_num, uint32_t trx_in_block) const
```

used to fetch an individual transaction.

```
optional<signed_transaction> 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.

```
chain_property_object get_chain_properties() const
```

Retrieve the chain_property_object associated with the chain.

```
global_property_object get_global_properties() const
```

Retrieve the current global_property_object.

```
fc::variant_object get_config() const
```

Retrieve compile-time constants.

```
chain_id_type get_chain_id() const
```

Get the chain ID.

```
dynamic_global_property_object get_dynamic_global_properties() const
```

Retrieve the current dynamic_global_property_object.

```
bool is_public_key_registered(string public_key) const
```

Determine whether a textual representation of a public key (in Base-58 format) is *currently* linked to any *registered* (i.e. non-stealth) account on the blockchain

Return Whether a public key is known

Parameters

- public_key: Public key

```
vector<optional<account_object>> 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> 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

```
vector<account_id_type> 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>> 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> 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

```
vector<asset> 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> 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> get_balance_objects (const vector<address> &addrs) const
```

Return all unclaimed balance objects for a set of addresses

```
uint64_t get_account_count () const
```

Get the total number of accounts registered with the blockchain.

```
vector<optional<asset_object>> 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> 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>> 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

```
vector<limit_order_object> 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<call_order_object> **get_call_orders** (asset_id_type *a*, uint32_t *limit*) **const**

Get call orders in a given asset.

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

Parameters

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

vector<force_settlement_object> **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> **get_margin_positions** (**const** account_id_type &*id*) **const**

Return all open margin positions for a given account id.

void **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 **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 **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_volume **get_24_volume** (**const** string &*base*, **const** string &*quote*) **const**

Returns the 24 hour volume for the market assetA:assetB.

Return The market volume over the past 24 hours

Parameters

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

```
order_book 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<market_trade> 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 trasactions to retrieve, capped at 100
- *start*: Start time as a UNIX timestamp

```
vector<optional<witness_object>> 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> 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> 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 get_witness_count () const
```

Get the total number of witnesses registered with the blockchain.

```
vector<optional<committee_member_object>> 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> 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> 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

```
vector<worker_object> get_workers_by_account (account_id_type account) const  
WORKERS.
```

Return the worker objects associated with this account.

```
vector<variant> 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.

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

```
set<public_key_type> 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> 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.

```
bool verify_authority (const signed_transaction &trx) const
```

Return true if the trx has all of the required signatures, otherwise throws an exception

```
bool 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 validate_transaction(const signed_transaction &trx) const
    Validates a transaction against the current state without broadcasting it on the network.
```

```
vector<fc::variant> 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
```

```
vector<proposal_object> get_proposed_transactions(account_id_type id) const
    Return the set of proposed transactions relevant to the specified account id.
```

```
vector<blinded_balance_object> get_blinded_balances(const
                                                       flat_set<commitment_type> &com-
                                                       mitments) const
    Return the set of blinded balance objects by commitment ID
```

```
class database_api_impl
    Inherits from std::enable_shared_from_this<database_api_impl>
```

Public Functions

```
vector<vector<account_id_type>> get_key_references(vector<public_key_type> key) const
    Return all accounts that referr to the key or account id in their owner or active authorities.
```

```
vector<limit_order_object> get_limit_orders(asset_id_type a, asset_id_type b, uint32_t limit) const
    Return the limit orders for both sides of the book for the two assets specified up to limit number on
    each side.
```

```
vector<proposal_object> get_proposed_transactions(account_id_type id) const
    TODO: add secondary index that will accelerate this process
```

```
void on_objects_new(const vector<object_id_type> &ids, const flat_set<account_id_type>
                    &impacted_accounts)
    called every time a block is applied to report the objects that were changed
```

```
void on_applied_block()
    note: this method cannot yield because it is called in the middle of apply a block.
```

```
struct get_required_fees_helper
    Container method for mutually recursive functions used to implement get_required_fees() with potentially
    nested proposals.
```

```
class history_api
    #include <api.hpp> The history_api class implements the RPC API for account history.

    This API contains methods to access account histories
```

Public Functions

```
vector<operation_history_object> 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> get_account_history_operations(account_id_type account, int operation_id, operation_history_id_type start = operation_history_id_type(), operation_history_id_type stop = operation_history_id_type(), unsigned limit = 100) const
```

Get only asked 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
- *operation_id*: The ID 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> 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.

```
class login_api
```

#include <api.hpp> The *login_api* class implements the bottom layer of the RPC API.

All other APIs must be requested from this API.

Public Functions

```
bool login(const string &user, const string &password)
```

Authenticate to the RPC server.

Return True if logged in successfully; false otherwise

Note This must be called prior to requesting other APIs. Other APIs may not be accessible until the client has successfully authenticated.

Parameters

- *user*: Username to login with
- *password*: Password to login with

```
fc::api<block_api> block() const
```

Retrieve the network block API.

```
fc::api<network_broadcast_api> network_broadcast() const
```

Retrieve the network broadcast API.

```
fc::api<database_api> database() const
```

Retrieve the database API.

```
fc::api<history_api> history() const
```

Retrieve the history API.

```
fc::api<network_node_api> network_node() const
```

Retrieve the network node API.

```
fc::api<crypto_api> crypto() const
```

Retrieve the cryptography API.

```
fc::api<asset_api> asset() const
```

Retrieve the asset API.

```
fc::api<graphene::debug_witness::debug_api> debug() const
```

Retrieve the debug API (if available)

```
void enable_api(const string &api_name)
```

Called to enable an API, not reflected.

```
class network_broadcast_api
```

#include <api.hpp> The *network_broadcast_api* class allows broadcasting of transactions.

Inherits from std::enable_shared_from_this< network_broadcast_api >

Public Functions

```
void 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 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.

```
fc::variant broadcast_transaction_synchronous(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.

```
void on_applied_block(const signed_block &b)
```

Not reflected, thus not accessible to API clients.

This function is registered to receive the applied_block signal from the chain database when a block is received. It then dispatches callbacks to clients who have requested to be notified when a particular txid is included in a block.

```
class network_node_api
```

#include <api.hpp> The `network_node_api` class allows maintenance of p2p connections.

Public Functions

```
fc::variant_object get_info() const
```

Return general network information, such as p2p port.

```
void 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

```
std::vector<net::peer_status> get_connected_peers() const
```

Get status of all current connections to peers.

```
fc::variant_object get_advanced_node_parameters() const
```

Get advanced node parameters, such as desired and max number of connections.

```
void 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

```
std::vector<net::potential_peer_record> get_potential_peers() const
```

Return list of potential peers.

```
class plugin
```

#include <plugin.hpp> Provides basic default implementations of `abstract_plugin` functions.

Inherits from `graphene::app::abstract_plugin`

Public Functions

`void plugin_initialize(const boost::program_options::variables_map &options)`

Perform early startup routines and register plugin indexes, callbacks, etc.

Plugins MUST supply a method initialize() which will be called early in the application startup. This method should contain early setup code such as initializing variables, adding indexes to the database, registering callback methods from the database, adding APIs, etc., as well as applying any options in the options map

This method is called BEFORE the database is open, therefore any routines which require any chain state MUST NOT be called by this method. These routines should be performed in startup() instead.

Parameters

- `options`: The options passed to the application, via configuration files or command line

`void plugin_startup()`

Begin normal runtime operations.

Plugins MUST supply a method startup() which will be called at the end of application startup. This method should contain code which schedules any tasks, or requires chain state.

`void plugin_shutdown()`

Cleanly shut down the plugin.

This is called to request a clean shutdown (e.g. due to SIGINT or SIGTERM).

`void plugin_set_app(application *a)`

Register the application instance with the plugin.

This is called by the framework to set the application.

`void plugin_set_program_options(boost::program_options::options_description &command_line_options,`

`boost::program_options::options_description &config_file_options)`

Fill in command line parameters used by the plugin.

This method populates its arguments with any command-line and configuration file options the plugin supports. If a plugin does not need these options, it may simply provide an empty implementation of this method.

Parameters

- `command_line_options`: All options this plugin supports taking on the command-line
- `config_file_options`: All options this plugin supports storing in a configuration file

`namespace detail`

Functions

`genesis_state_type create_example_genesis()`

`class application_impl`

Inherits from node_delegate

Public Functions

virtual bool has_item(const net::item_id &id)

If delegate has the item, the network has no need to fetch it.

**virtual bool handle_block(const graphene::net::block_message &blk_msg,
bool sync_mode, std::vector<fc::uint160_t> &contained_transaction_message_ids)**

allows the application to validate an item prior to broadcasting to peers.

Return true if this message caused the blockchain to switch forks, false if it did not

Parameters

- sync_mode: true if the message was fetched through the sync process, false during normal operation

Exceptions

- exception: if error validating the item, otherwise the item is safe to broadcast on.

**virtual std::vector<item_hash_t> get_block_ids(const std::vector<item_hash_t>
&blockchainSynopsis, uint32_t
&remaining_item_count, uint32_t
limit)**

Assuming all data elements are ordered in some way, this method should return up to limit ids that occur *after* the last ID in synopsis that we recognize.

On return, remaining_item_count will be set to the number of items in our blockchain after the last item returned in the result, or 0 if the result contains the last item in the blockchain

virtual message get_item(const item_id &id)

Given the hash of the requested data, fetch the body.

**virtual std::vector<item_hash_t> get_blockchainSynopsis(const item_hash_t
&reference_point,
uint32_t num-
ber_of_blocks_after_reference_point)**

Returns a synopsis of the blockchain used for syncing. This consists of a list of block hashes at intervals exponentially increasing towards the genesis block. When syncing to a peer, the peer uses this data to determine if we're on the same fork as they are, and if not, what blocks they need to send us to get us on their fork.

In the over-simplified case, this is a straightforward synopsis of our current preferred blockchain; when we first connect up to a peer, this is what we will be sending. It looks like this: If the blockchain is empty, it will return the empty list. If the blockchain has one block, it will return a list containing just that block. If it contains more than one block: the first element in the list will be the hash of the highest numbered block that we cannot undo the second element will be the hash of an item at the half way point in the undoable segment of the blockchain the third will be ~3/4 of the way through the undoable segment of the block chain the fourth will be at ~7/8... &c. the last item in the list will be the hash of the most recent block on our preferred chain so if the blockchain had 26 blocks labeled a - z, the synopsis would be: a n u x z the idea being that by sending a small (<30) number of block ids, we can summarize a huge blockchain. The block ids are more dense near the end of the chain where because we are more likely to be almost in sync when we first connect, and forks are likely to be short. If the peer we're syncing with in our example is on a fork that started at block 'v', then they will reply to our synopsis with a list of all blocks starting from block 'u', the last block they know that we had in common.

In the real code, there are several complications.

First, as an optimization, we don't usually send a synopsis of the entire blockchain, we send a synopsis of only the segment of the blockchain that we have undo data for. If their fork doesn't

build off of something in our undo history, we would be unable to switch, so there's no reason to fetch the blocks.

Second, when a peer replies to our initial synopsis and gives us a list of the blocks they think we are missing, they only send a chunk of a few thousand blocks at once. After we get those block ids, we need to request more blocks by sending another synopsis (we can't just say "send me the next 2000 ids" because they may have switched forks themselves and they don't track what they've sent us). For faster performance, we want to get a fairly long list of block ids first, then start downloading the blocks. The peer doesn't handle these follow-up block id requests any different from the initial request; it treats the synopsis we send as our blockchain and bases its response entirely off that. So to get the response we want (the next chunk of block ids following the last one they sent us, or, failing that, the shortest fork off of the last list of block ids they sent), we need to construct a synopsis as if our blockchain was made up of:

1. the blocks in our block chain up to the fork point (if there is a fork) or the head block (if no fork)
2. the blocks we've already pushed from their fork (if there's a fork)
3. the block ids they've previously sent us Segment 3 is handled in the p2p code, it just tells us the number of blocks it has (in `number_of_blocks_after_reference_point`) so we can leave space in the synopsis for them. We're responsible for constructing the synopsis of Segments 1 and 2 from our active blockchain and fork database. The `reference_point` parameter is the last block from that peer that has been successfully pushed to the blockchain, so that tells us whether the peer is on a fork or on the main chain.

virtual void sync_status (uint32_t item_type, uint32_t item_count)

Call this after the call to `handle_message` succeeds.

Parameters

- `item_type`: the type of the item we're synchronizing, will be the same as `item` passed to the `sync_from()` call
- `item_count`: the number of items known to the node that haven't been sent to `handle_item()` yet. After `item_count` more calls to `handle_item()`, the node will be in sync

virtual void connection_count_changed (uint32_t c)

Call any time the number of connected peers changes.

virtual fc::time_point_sec get_block_time (const item_hash_t &block_id)

Returns the time a block was produced (if `block_id` = 0, returns genesis time). If we don't know about the block, returns `time_point_sec::min()`

Graphene::Chain

```
namespace graphene::chain
```

Typedefs

```
typedef multi_index_container<account_balance_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, >>> account_balance_index
typedef generic_index<account_balance_object, account_balance_object_multi_index_type> account_balance_index
typedef multi_index_container<account_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, >>> account_index
typedef generic_index<account_object, account_multi_index_type> account_index
typedef multi_index_container<asset_bitasset_data_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, >>> asset_bitasset_data_index
typedef generic_index<asset_bitasset_data_object, asset_bitasset_data_object_multi_index_type> asset_bitasset_data_index
```

```
typedef multi_index_container<asset_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> asset_index

using graphene::chain::balance_multi_index_type = typedef multi_index_container< balance_object, balance_index_type>

using graphene::chain::balance_index = typedef generic_index<balance_object, balance_index_type>

typedef multi_index_container<buyback_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> buyback_index

using graphene::chain::committee_member_multi_index_type = typedef multi_index_container< committee_member_object, committee_member_index_type>

using graphene::chain::committee_member_index = typedef generic_index<committee_member_object, committee_member_index_type>

typedef multi_index_container<blinded_balance_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> blinded_balance_index

typedef generic_index<blinded_balance_object, blinded_balance_object_multi_index_type> blinded_balance_index

typedef shared_ptr<fork_item> item_ptr

typedef multi_index_container<limit_order_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> limit_order_index

typedef generic_index<limit_order_object, limit_order_multi_index_type> limit_order_index

typedef multi_index_container<call_order_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> call_order_index

typedef generic_index<force_settlement_object, force_settlement_object_multi_index_type> force_settlement_index

typedef multi_index_container<account_transaction_history_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> account_transaction_history_index

typedef generic_index<account_transaction_history_object, account_transaction_history_multi_index_type> account_transaction_history_index

typedef boost::multi_index_container<proposal_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type, &object>>> proposal_index

typedef generic_index<proposal_object, proposal_multi_index_container> proposal_index
```

typedef static_variant<account_name_eq_lit_predicate, asset_symbol_eq_lit_predicate, block_id_predicate> predicate

When defining predicates do not make the protocol dependent upon implementation details.

typedef fc::static_variant<void_result, object_id_type, asset> operation_result

typedef static_variant<void_t> future_extensions

For future expansion many structus include a single member of type extensions_type that can be changed when updating a protocol. You can always add new types to a static_variant without breaking backward compatibility.

typedef flat_set<future_extensions> extensions_type

A flat_set is used to make sure that only one extension of each type is added and that they are added in order.

Note static_variant compares only the type tag and not the content.

typedef static_variant parameter_extension

typedef transform_to_fee_parameters<operation>::type fee_parameters

typedef fee_schedule fee_schedule_type

typedef fc::static_variant<transfer_operation, limit_order_create_operation, limit_order_cancel_operation, call_order_update_operation>

Defines the set of valid operations as a discriminated union type.

typedef static_variant<no_special_authority, top_holders_special_authority> special_authority

```

typedef fc::ecc::private_key private_key_type
typedef fc::sha256 chain_id_type
typedef object_id<protocol_ids, account_object_type, account_object> account_id_type
typedef object_id<protocol_ids, asset_object_type, asset_object> asset_id_type
typedef object_id<protocol_ids, force_settlement_object_type, force_settlement_object> force_settlement_id_type
typedef object_id<protocol_ids, committee_member_object_type, committee_member_object> committee_member_id_type
typedef object_id<protocol_ids, witness_object_type, witness_object> witness_id_type
typedef object_id<protocol_ids, limit_order_object_type, limit_order_object> limit_order_id_type
typedef object_id<protocol_ids, call_order_object_type, call_order_object> call_order_id_type
typedef object_id<protocol_ids, custom_object_type, custom_object> custom_id_type
typedef object_id<protocol_ids, proposal_object_type, proposal_object> proposal_id_type
typedef object_id<protocol_ids, operation_history_object_type, operation_history_object> operation_history_id_type
typedef object_id<protocol_ids, withdraw_permission_object_type, withdraw_permission_object> withdraw_permission_id_type
typedef object_id<protocol_ids, vesting_balance_object_type, vesting_balance_object> vesting_balance_id_type
typedef object_id<protocol_ids, worker_object_type, worker_object> worker_id_type
typedef object_id<protocol_ids, balance_object_type, balance_object> balance_id_type
typedef object_id<implementation_ids, impl_global_property_object_type, global_property_object> global_property_id_type
typedef object_id<implementation_ids, impl_dynamic_global_property_object_type, dynamic_global_property_object> dynamic_global_property_id_type
typedef object_id<implementation_ids, impl_asset_dynamic_data_type, asset_dynamic_data_object> asset_dynamic_data_id_type
typedef object_id<implementation_ids, impl_asset_bitasset_data_type, asset_bitasset_data_object> asset_bitasset_data_id_type
typedef object_id<implementation_ids, impl_account_balance_object_type, account_balance_object> account_balance_id_type
typedef object_id<implementation_ids, impl_account_statistics_object_type, account_statistics_object> account_statistics_id_type
typedef object_id<implementation_ids, impl_transaction_object_type, transaction_object> transaction_obj_id_type
typedef object_id<implementation_ids, impl_block_summary_object_type, block_summary_object> block_summary_id_type
typedef object_id<implementation_ids, impl_account_transaction_history_object_type, account_transaction_history_object> account_transaction_history_id_type
typedef object_id<implementation_ids, impl_chain_property_object_type, chain_property_object> chain_property_id_type
typedef object_id<implementation_ids, impl_witness_schedule_object_type, witness_schedule_object> witness_schedule_id_type
typedef object_id<implementation_ids, impl_budget_record_object_type, budget_record_object> budget_record_id_type
typedef object_id<implementation_ids, impl_blinded_balance_object_type, blinded_balance_object> blinded_balance_id_type
typedef object_id<implementation_ids, impl_special_authority_object_type, special_authority_object> special_authority_id_type
typedef object_id<implementation_ids, impl_buyback_object_type, buyback_object> buyback_id_type
typedef object_id<implementation_ids, impl_fba_accumulator_object_type, fba_accumulator_object> fba_accumulator_id_type
typedef fc::array<char, GRAPHENE_MAX_ASSET_SYMBOL_LENGTH> symbol_type
typedef fc::ripemd160 block_id_type
typedef fc::ripemd160 checksum_type
typedef fc::ripemd160 transaction_id_type

```

```
typedef fc::sha256 digest_type
typedef fc::ecc::compact_signature signature_type
typedef safe<int64_t> share_type
typedef uint16_t weight_type
typedef fc::static_variant<linear_vesting_policy_initializer, cdd_vesting_policy_initializer> vesting_policy_initializer
typedef static_variant<refund_worker_initializer, vesting_balance_worker_initializer, burn_worker_initializer> worker_initializer
typedef multi_index_container<special_authority_object, indexed_by<ordered_unique<tag<by_id>, member<object, object>>> special_authority_index
typedef generic_index<special_authority_object, special_authority_multi_index_type> special_authority_index
typedef multi_index_container<transaction_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type>>> transaction_index
```

```
typedef generic_index<transaction_object, transaction_multi_index_type> transaction_index
typedef fc::static_variant<linear_vesting_policy, cdd_vesting_policy> vesting_policy
typedef multi_index_container<vesting_balance_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type>>> vesting_balance_index
typedef generic_index<vesting_balance_object, vesting_balance_multi_index_type> vesting_balance_index
typedef multi_index_container<withdraw_permission_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type>>> withdraw_permission_index
typedef generic_index<withdraw_permission_object, withdraw_permission_object_multi_index_type> withdraw_permission_index
using graphene::chain::witness_multi_index_type = typedef multi_index_container< witness_object, witness_multi_index_type>
using graphene::chain::witness_index = typedef generic_index<witness_object, witness_multi_index_type>
typedef static_variant<refund_worker_type, vesting_balance_worker_type, burn_worker_type> worker_type
typedef multi_index_container<worker_object, indexed_by<ordered_unique<tag<by_id>, member<object, object_id_type>>> worker_index
using graphene::chain::worker_index = typedef generic_index<worker_object, worker_index>
typedef boost::multiprecision::uint128_t uint128_t
typedef boost::multiprecision::int128_t int128_t
typedef fc::smart_ref<fee_schedule> smart_fee_schedule
```

Enums

```
enum graphene_fba_accumulator_id_enum
```

An object will be created at genesis for each of these FBA accumulators.

Values:

```

fba_accumulator_id_transfer_to_blind=0
fba_accumulator_id_blind_transfer
fba_accumulator_id_transfer_from_blind
fba_accumulator_id_count

enum asset_issuer_permission_flags
  Values:
    charge_market_fee = 0x01
      an issuer-specified percentage of all market trades in this asset is paid to the issuer
    white_list = 0x02
      accounts must be whitelisted in order to hold this asset
    override_authority = 0x04
      issuer may transfer asset back to himself
    transfer_restricted = 0x08
      require the issuer to be one party to every transfer
    disable_force_settle = 0x10
      disable force settling
    global_settle = 0x20
      allow the bitasset issuer to force a global settling this may be set in permissions, but not flags
    disable_confidential = 0x40
      allow the asset to be used with confidential transactions
    witness_fed_asset = 0x80
      allow the asset to be fed by witnesses
    committee_fed_asset = 0x100
      allow the asset to be fed by the committee

enum reserved_spaces
  Values:
    relative_protocol_ids = 0
    protocol_ids = 1
    implementation_ids = 2

enum object_type
  List all object types from all namespaces here so they can be easily reflected and displayed in debug output.
  If a 3rd party wants to extend the core code then they will have to change the packed_object::type field
  from enum_type to uint16 to avoid warnings when converting packed_objects to/from json.

  Values:
    null_object_type
    base_object_type
    account_object_type
    asset_object_type
    force_settlement_object_type
    committee_member_object_type
    witness_object_type

```

```
limit_order_object_type
call_order_object_type
custom_object_type
proposal_object_type
operation_history_object_type
withdraw_permission_object_type
vesting_balance_object_type
worker_object_type
balance_object_type
OBJECT_TYPE_COUNT
    Sentry value which contains the number of different object types.

enum impl_object_type
    Values:
        impl_global_property_object_type
        impl_dynamic_global_property_object_type
        impl_reserved0_object_type
        impl_asset_dynamic_data_type
        impl_asset_bitasset_data_type
        impl_account_balance_object_type
        impl_account_statistics_object_type
        impl_transaction_object_type
        impl_block_summary_object_type
        impl_account_transaction_history_object_type
        impl_blinded_balance_object_type
        impl_chain_property_object_type
        impl_witness_schedule_object_type
        impl_budget_record_object_type
        impl_special_authority_object_type
        impl_buyback_object_type
        impl_fba_accumulator_object_type
```

Functions

```
void verify_authority_accounts (const database &db, const authority &a)
void verify_account_votes (const database &db, const account_options &options)
share_type cut_fee (share_type a, uint16_t p)
```

```

void evaluate_buyback_account_options (const database &db, const buy-
                                         back_account_options &bbo)

void debug_apply_update (database &db, const fc::variant_object &vo)
template <typename Visitor>
void visit_special_authorities (const database &db, Visitor visit)

void update_top_n_authorities (database &db)

void split_fba_balance (database &db, uint64_t fba_id, uint16_t network_pct, uint16_t design-
                                         nated_asset_buyback_pct, uint16_t designated_asset_issuer_pct)

void distribute_fba_balances (database &db)

void create_buyback_orders (database &db)

void deprecate_annual_members (database &db)

bool maybe_cull_small_order (database &db, const limit_order_object &order)

fc::variant_object get_config ()

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (transfer)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (call_order_update)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (account_create)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (account_update)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (asset_reserve)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (proposal_create)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (balance_claim)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (override_transfer)

GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (blind_transfer)

graphene::chain::GRAPHENE_DECLARE_OP_EVALUATE_EXCEPTION (unknown_commitment, blind_trans-
                                         action)

bool isAuthorizedAsset (const database &d, const account_object &acct, const asset_
                                         object &asset_obj)

```

Return true if the account is whitelisted and not blacklisted to transact in the provided asset; false otherwise.

bool is_valid_name (const string &s)

Names must comply with the following grammar (RFC 1035): <domain> ::= <subdomain> | ” ” <subdomain> ::= <label> | <subdomain> “.” <label> <label> ::= <letter> [[<ldh-str>] <let-dig>] <ldh-str> ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str> <let-dig-hyp> ::= <let-dig> | “-” <let-dig> ::= <letter> | <digit>

Which is equivalent to the following:

<domain> ::= <subdomain> | ” ” <subdomain> ::= <label> (“.” <label>)* <label> ::= <letter> [[<let-dig-
 hyp>+] <let-dig>] <let-dig-hyp> ::= <let-dig> | “-” <let-dig> ::= <letter> | <digit>

I.e. a valid name consists of a dot-separated sequence of one or more labels consisting of the following rules:

- Each label is three characters or more

- Each label begins with a letter
- Each label ends with a letter or digit
- Each label contains only letters, digits or hyphens

In addition we require the following:

- All letters are lowercase
- Length is between (inclusive) GRAPHENE_MIN_ACCOUNT_NAME_LENGTH and GRAPHENE_MAX_ACCOUNT_NAME_LENGTH

```
bool is_cheap_name (const string &n)

bool operator== (const address &a, const address &b)

bool operator!= (const address &a, const address &b)

bool operator< (const address &a, const address &b)

price operator/ (const asset &base, const asset &quote)

price operator~ (const price &p)

bool operator< (const asset &a, const asset &b)

bool operator<= (const asset &a, const asset &b)

bool operator< (const price &a, const price &b)

bool operator<= (const price &a, const price &b)

bool operator> (const price &a, const price &b)

bool operator>= (const price &a, const price &b)

bool operator== (const price &a, const price &b)

bool operator!= (const price &a, const price &b)

asset operator* (const asset &a, const price &b)

bool is_valid_symbol (const string &symbol)
    Valid symbols can contain [A-Z0-9], and '.' They must start with [A, Z] They must end with [A, Z] They can contain a maximum of one '.'

void add_authority_accounts (flat_set<account_id_type> &result, const authority &a)
    Add all account members of the given authority to the given flat_set.

template <typename Stream, class T>
void operator<< (Stream &stream, const graphene::chain::extension<T> &value)

template <typename Stream, typename T>
void operator>> (Stream &s, graphene::chain::extension<T> &value)

void operation_get_required_authorities (const operation &op,
                                         flat_set<account_id_type> &active,
                                         flat_set<account_id_type> &owner, vector<authority> &other)
    Appends required authorities to the result vector. The authorities appended are not the same as those returned by get_required_auth
```

Return a set of required authorities for op

```

void operation_validate(const operation &op)
void validate_special_authority(const special_authority &auth)
void verify_authority(const vector<operation> &ops, const flat_set<public_key_type>
                      &sigs, const std::function<const authority*> account_id_type
                      > &get_active, const std::function<const authority*> account_id_type &get_owner, uint32_t
max_recursion = GRAPHENE_MAX_SIG_CHECK_DEPTH, bool allow_committe = false,
const flat_set<account_id_type> &active_aprovals = flat_set<account_id_type>(), const
flat_set<account_id_type> &owner_approvals = flat_set<account_id_type>())

bool is_relative(object_id_type o)
vote_id_type get_next_vote_id(global_property_object &gpo, vote_id_type::vote_type type)
bool operator==(const pts_address &a, const pts_address &b)
bool operator!=(const pts_address &a, const pts_address &b)
bool operator<(const pts_address &a, const pts_address &b)
void evaluate_special_authority(const database &db, const special_authority &auth)
bool operator==(const public_key_type &p1, const fc::ecc::public_key &p2)
bool operator==(const public_key_type &p1, const public_key_type &p2)
bool operator!=(const public_key_type &p1, const public_key_type &p2)
bool operator==(const extended_public_key_type &p1, const fc::ecc::extended_public_key
                  &p2)
bool operator==(const extended_public_key_type &p1, const extended_public_key_type &p2)
bool operator!=(const extended_public_key_type &p1, const extended_public_key_type &p2)
bool operator==(const extended_private_key_type &p1, const fc::ecc::extended_public_key
                  &p2)
bool operator==(const extended_private_key_type &p1, const extended_private_key_type &p2)
bool operator!=(const extended_private_key_type &p1, const extended_private_key_type &p2)
bool sum_below_max_shares(const asset &a, const asset &b)
VESTING_VISITOR(on_deposit)
VESTING_VISITOR(on_deposit_vested)
VESTING_VISITOR(on_withdraw)
graphene::chain::VESTING_VISITOR(is_deposit_allowed, const)
graphene::chain::VESTING_VISITOR(is_deposit_vested_allowed, const)
graphene::chain::VESTING_VISITOR(is_withdraw_allowed, const)
graphene::chain::VESTING_VISITOR(get_allowed_withdraw, const)

```

Variables

Public Members

```
account_id_type registrar
    This account pays the fee. Must be a lifetime member.

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
    Of the fee split between registrar and referrer, this percentage goes to the referrer. The rest goes to the
    registrar.

struct fee_parameters_type
    #include <account.hpp>
```

Public Members

```
uint64_t basic_fee = 5*GRAPHENE_BLOCKCHAIN_PRECISION
    the cost to register the cheapest non-free account

uint64_t premium_fee = 2000*GRAPHENE_BLOCKCHAIN_PRECISION
    the cost to register the cheapest non-free account

account_member_index
include <account_object.hpp> This secondary index will allow a reverse lookup of all accounts that a particular key or account is a potential signing authority.

inherits from secondary_index
```

Public Members

```
map<account_id_type, set<account_id_type>> account_to_account_memberships
    given an account or key, map it to the set of accounts that reference it in an active or owner authority
map<address, set<account_id_type>> account_to_address_memberships
    some accounts use address authorities in the genesis block

struct account_name_eq_lit_predicate
    #include <assert.hpp> Used to verify that account_id->name is equal to the given string literal.
```

Public Functions

```
bool validate() const
    Perform state-independent checks. Verify account_name is a valid account name.
```

```
class account_object
    #include <account_object.hpp> This class represents an account on the object graph
    Accounts are the primary unit of authority on the graphene system. Users must have an account in order to use assets, trade in the markets, vote for committee_members, etc.
    Inherits from graphene::db::abstract_object< account_object >
```

Unnamed Group

```
set<account_id_type> whitelisted_accounts
    Optionally track all of the accounts this account has whitelisted or blacklisted, these should be made
    Immutable so that when the account object is cloned no deep copy is required. This state is tracked
    for GUI display purposes.

    TODO: move white list tracking to its own multi-index container rather than having 4 fields on an
    account. This will scale better because under the current design if you whitelist 2000 accounts, then
    every time someone fetches this account object they will get the full list of 2000 accounts.
```

Public Functions

```
bool is_lifetime_member() const
    Return true if this is a lifetime member account; false otherwise.
```

```
bool is_basic_account(time_point_sec now) const
    Return true if this is a basic account; false otherwise.
```

```
bool is_annual_member(time_point_sec now) const
    Return true if the account is an unexpired annual member; false otherwise.
    Note This method will return false for lifetime members.
```

```
bool is_member(time_point_sec now) const
    Return true if the account is an annual or lifetime member; false otherwise.
```

Public Members

time_point_sec **membership_expiration_date**

The time at which this account's membership expires. If set to any time in the past, the account is a basic account. If set to time_point_sec::maximum(), the account is a lifetime member. If set to any time not in the past less than time_point_sec::maximum(), the account is an annual member.

See *is_lifetime_member*, *is_basic_account*, *is_annual_member*, and *is_member*

account_id_type **registrar**

The account that paid the fee to register this account. Receives a percentage of referral rewards.

account_id_type **referrer**

The account credited as referring this account. Receives a percentage of referral rewards.

account_id_type **lifetime_referrer**

The lifetime member at the top of the referral tree. Receives a percentage of referral rewards.

uint16_t **network_fee_percentage** = GRAPHENE_DEFAULT_NETWORK_PERCENT_OF_FEE

Percentage of fee which should go to network.

uint16_t **lifetime_referrer_fee_percentage** = 0

Percentage of fee which should go to lifetime referrer.

uint16_t **referrer_rewards_percentage** = 0

Percentage of referral rewards (leftover fee after paying network and lifetime referrer) which should go to referrer. The remainder of referral rewards goes to the registrar.

string **name**

The account's name. This name must be unique among all account names on the graph. May not be empty.

authority **owner**

The owner authority represents absolute control over the account. Usually the keys in this authority will be kept in cold storage, as they should not be needed very often and compromise of these keys constitutes complete and irrevocable loss of the account. Generally the only time the owner authority is required is to update the active authority.

authority **active**

The owner authority contains the hot keys of the account. This authority has control over nearly all operations the account may perform.

account_statistics_id_type **statistics**

The reference implementation records the account's statistics in a separate object. This field contains the ID of that object.

flat_set<account_id_type> **whitelisting_accounts**

This is a set of all accounts which have 'whitelisted' this account. Whitelisting is only used in core validation for the purpose of authorizing accounts to hold and transact in whitelisted assets. This account cannot update this set, except by transferring ownership of the account, which will clear it. Other accounts may add or remove their IDs from this set.

flat_set<account_id_type> **blacklisting_accounts**

This is a set of all accounts which have 'blacklisted' this account. Blacklisting is only used in core validation for the purpose of forbidding accounts from holding and transacting in whitelisted assets. This account cannot update this set, and it will be preserved even if the account is transferred. Other accounts may add or remove their IDs from this set.

optional<vesting_balance_id_type> **cashback_vb**

Vesting balance which receives cashback_reward deposits.

```
uint8_t top_n_control_flags = 0
```

This flag is set when the top_n logic sets both authorities, and gets reset when authority or special_authority is set.

```
optional<flat_set<asset_id_type>> allowed_assets
```

This is a set of assets which the account is allowed to have. This is utilized to restrict buyback accounts to the assets that trade in their markets. In the future we may expand this to allow accounts to e.g. voluntarily restrict incoming transfers.

```
struct account_options
```

#include <account.hpp> These are the fields which can be updated by the active authority.

Public Members

```
public_key_type memo_key
```

The memo key is the key this account will typically use to encrypt/sign transaction memos and other non-validated account activities. This field is here to prevent confusion if the active authority has zero or multiple keys in it.

```
account_id_type voting_account = GRAPHENE_PROXY_TO_SELF_ACCOUNT
```

If this field is set to an account ID other than GRAPHENE_PROXY_TO_SELF_ACCOUNT, then this account's votes will be ignored; its stake will be counted as voting for the referenced account's selected votes instead.

```
uint16_t num_witness = 0
```

The number of active witnesses this account votes the blockchain should appoint Must not exceed the actual number of witnesses voted for in *votes*

```
uint16_t num_committee = 0
```

The number of active committee members this account votes the blockchain should appoint Must not exceed the actual number of committee members voted for in *votes*

```
flat_set<vote_id_type> votes
```

This is the list of vote IDs this account votes for. The weight of these votes is determined by this account's balance of core asset.

```
class account_referrer_index
```

#include <account_object.hpp> This secondary index will allow a reverse lookup of all accounts that have been referred by a particular account.

Inherits from secondary_index

Public Members

```
map<account_id_type, set<account_id_type>> referred_by
```

maps the referrer to the set of accounts that they have referred

```
class account_statistics_object
```

#include <account_object.hpp> This object contains regularly updated statistical data about an account. It is provided for the purpose of separating the account data that changes frequently from the account data that is mostly static, which will minimize the amount of data that must be backed up as part of the undo history everytime a transfer is made.

Inherits from graphene::db::abstract_object< account_statistics_object >

Public Functions

```
void process_fees (const account_object &a, database &d) const
    Split up and pay out pending_fees and pending_vested_fees.
```

```
void pay_fee (share_type core_fee, share_type cashback_vesting_threshold)
    Core fees are paid into the account_statistics_object by this method
```

Public Members

account_transaction_history_id_type most_recent_op

Keep the most recent operation as a root pointer to a linked list of the transaction history.

uint32_t total_ops = 0

Total operations related to this account.

uint32_t removed_ops = 0

Total operations related to this account that has been removed from the database.

share_type total_core_in_orders

When calculating votes it is necessary to know how much is stored in orders (and thus unavailable for transfers). Rather than maintaining an index of [asset,owner,order_id] we will simply maintain the running total here and update it every time an order is created or modified.

share_type lifetime_fees_paid

Tracks the total fees paid by this account for the purpose of calculating bulk discounts.

share_type pending_fees

Tracks the fees paid by this account which have not been disseminated to the various parties that receive them yet (registrar, referrer, lifetime referrer, network, etc). This is used as an optimization to avoid doing massive amounts of uint128 arithmetic on each and every operation.

These fees will be paid out as vesting cash-back, and this counter will reset during the maintenance interval.

share_type pending_vested_fees

Same as *pending_fees*, except these fees will be paid out as pre-vested cash-back (immediately available for withdrawal) rather than requiring the normal vesting period.

class account_transaction_history_object

#include <operation_history_object.hpp> a node in a linked list of operation_history_objects

Account history is important for users and wallets even though it is not part of “core validation”. Account history is maintained as a linked list stored on disk in a stack. Each account will point to the most recent account history object by ID. When a new operation relativent to that account is processed a new account history object is allocoated at the end of the stack and intialized to point to the prior object.

This data is never accessed as part of chain validation and therefore can be kept on disk as a memory mapped file. Using a memory mapped file will help the operating system better manage / cache / page files and also accelerates load time.

When the transaction history for a particular account is requested the linked list can be traversed with relatively effecient disk access because of the use of a memory mapped stack.

Inherits from graphene::db::abstract_object< account_transaction_history_object >

Public Members

operation_history_id_type **operation_id**

the account this operation applies to

account_transaction_history_id_type **next**

the operation position within the given account

struct account_transfer_operation

#include <account.hpp> 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.

Inherits from *graphene::chain::base_operation*

class account_update_evaluator

#include <account_evaluator.hpp> Inherits from *graphene::chain::evaluator<account_update_evaluator>*

struct account_update_operation

#include <account.hpp> 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.

Inherits from *graphene::chain::base_operation*

Public Members

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.

class account_upgrade_evaluator

#include <account_evaluator.hpp> Inherits from *graphene::chain::evaluator<account_upgrade_evaluator>*

struct account_upgrade_operation

#include <account.hpp> 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.

Inherits from `graphene::chain::base_operation`

Public Members

`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.

`struct fee_parameters_type`

`#include <account.hpp>`

Public Members

`uint64_t membership_lifetime_fee = 10000 * GRAPHENE_BLOCKCHAIN_PRECISION`
the cost to upgrade to a lifetime member

`class account_whitelist_evaluator`

`#include <account_evaluator.hpp>` Inherits from `graphene::chain::evaluator<account_whitelist_evaluator>`

`struct account_whitelist_operation`

`#include <account.hpp>` 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 whitelistAuthorities and blacklisted by none of S's blacklistAuthorities. 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`.

Inherits from `graphene::chain::base_operation`

Public Members

`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`

The new white and blacklist status of account_to_list, as determined by authorizing_account This is a bitfield using values defined in the account_listing enum

class address

`#include <address.hpp>` a 160 bit hash of a public key

An address can be converted to or from a base58 string with 32 bit checksum.

An address is calculated as ripemd160(sha512(compressed_ecc_public_key))

When converted to a string, checksum calculated as the first 4 bytes ripemd160(address) is appended to the binary address before converting to base58.

Public Functions

address ()

constructs empty / null address

address (const std::string &base58str)

converts to binary, validates checksum

address (const fc::ecc::public_key &pub)

converts to binary

address (const fc::ecc::public_key_data &pub)

converts to binary

address (const pts_address &pub)

converts to binary

operator std::string() const

converts to base58 + checksum

class assert_evaluator

`#include <assert_evaluator.hpp>` Inherits from `graphene::chain::evaluator< assert_evaluator >`

struct assert_operation

`#include <assert.hpp>` 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.

Inherits from `graphene::chain::base_operation`

Public Functions

`share_type calculate_fee (const fee_parameters_type &k) const`

The fee for assert operations is proportional to their size, but cheaper than a data fee because they require no storage

class asset_bitasset_data_object

`#include <asset_object.hpp>` contains properties that only apply to bitassets (market issued assets)

Inherits from `graphene::db::abstract_object< asset_bitasset_data_object >`

Unnamed Group

`price settlement_price`

In the event of a black swan, the swan price is saved in the settlement price, and all margin positions are settled at the same price with the seized collateral being moved into the settlement fund. From this point on no further updates to the asset are permitted (no feeds, etc) and forced settlement occurs immediately when requested, using the settlement price and fund. Price at which force settlements of a black swanned asset will occur

`share_type settlement_fund`

Amount of collateral which is available for force settlement.

Public Functions

`share_type max_force_settlement_volume (share_type current_supply) const`

Calculate the maximum force settlement volume per maintenance interval, given the current share supply.

`bool has_settlement () const`

return true if there has been a black swan, false otherwise

Public Members

`bitasset_options options`

The tunable options for BitAssets are stored in this field.

`flat_map<account_id_type, pair<time_point_sec, price_feed>> feeds`

Feeds published for this asset. If issuer is not committee, the keys in this map are the feed publishing accounts; otherwise, the feed publishers are the currently active committee_members and witnesses and this map should be treated as an implementation detail. The timestamp on each feed is the time it was published.

`price_feed current_feed`

This is the currently active price feed, calculated as the median of values from the currently active feeds.

`time_point_sec current_feed_publication_time`

This is the publication time of the oldest feed which was factored into current_feed.

`bool is_prediction_market = false`

True if this asset implements a Prediction Market.

`share_type force_settled_volume`

This is the volume of this asset which has been force-settled this maintenance interval.

`class asset_claim_fees_evaluator`

`#include <asset_evaluator.hpp>` Inherits from `graphene::chain::evaluator<asset_claim_fees_evaluator>`

`struct asset_claim_fees_operation`

`#include <asset_ops.hpp>` used to transfer accumulated fees back to the issuer's balance.

Inherits from `graphene::chain::base_operation`

Public Members

```
extensions_type extensions
    amount_to_claim.asset_id->issuer must == issuer

class asset_create_evaluator
    #include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_create_evaluator>

struct asset_create_operation
    #include <asset_ops.hpp> Inherits from graphene::chain::base_operation
```

Public Members

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
Options common to all assets.

Note common_options.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.

optional<bitasset_options> bitasset_opts
Options only available for BitAssets. MUST be non-null if and only if the market_issued flag is set in common_options.flags

bool is_prediction_market = false
For BitAssets, set this to true if the asset implements a Prediction Market; false otherwise.

class asset_dynamic_data_object
#include <asset_object.hpp> tracks the asset information that changes frequently

Because the `asset_object` is very large it doesn't make sense to save an undo state for all of the parameters that never change. This object factors out the parameters of an asset that change in almost every transaction that involves the asset.

This object exists as an implementation detail and its ID should never be referenced by a blockchain operation.

Inherits from graphene::db::abstract_object<asset_dynamic_data_object>

Public Members

share_type current_supply
The number of shares currently in existence.

share_type confidential_supply
total asset held in confidential balances

share_type accumulated_fees
fees accumulate to be paid out over time

```
share_type fee_pool
in core asset

class asset_fund_fee_pool_evaluator
#include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_fund_fee_pool_evaluator>

struct asset_fund_fee_pool_operation
#include <asset_ops.hpp> Inherits from graphene::chain::base_operation
```

Public Members

```
asset fee
core asset

share_type amount
core asset

class asset_global_settle_evaluator
#include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_global_settle_evaluator>

struct asset_global_settle_operation
#include <asset_ops.hpp> 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.

Inherits from graphene::chain::base_operation
```

Public Members

```
account_id_type issuer
must equal asset_to_settle->issuer

class asset_issue_evaluator
#include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_issue_evaluator>

struct asset_issue_operation
#include <asset_ops.hpp> Inherits from graphene::chain::base_operation
```

Public Members

```
account_id_type issuer
Must be asset_to_issue->asset_id->issuer.

optional<memo_data> memo
user provided data encrypted to the memo key of the “to” account
```

```
class asset_object
#include <asset_object.hpp> tracks the parameters of an asset
```

All assets have a globally unique symbol name that controls how they are traded and an issuer who has authority over the parameters of the asset.

Inherits from `graphene::db::abstract_object<asset_object>`

Public Functions

`bool is_market_issued() const`

Return true if this is a market-issued asset; false otherwise.

`bool can_force_settle() const`

Return true if users may request force-settlement of this market-issued asset; false otherwise

`bool can_global_settle() const`

Return true if the issuer of this market-issued asset may globally settle the asset; false otherwise

`bool charges_market_fees() const`

Return true if this asset charges a fee for the issuer on market operations; false otherwise

`bool is_transfer_restricted() const`

Return true if this asset may only be transferred to/from the issuer or market orders

`asset amount(share_type a) const`

Helper function to get an asset object with the given amount in this asset's type.

`asset amount_from_string(string amount_string) const`

Convert a string amount (i.e. “123.45”) to an asset object with this asset’s type. The string may have a decimal and/or a negative sign.

`string amount_to_string(share_type amount) const`

Convert an asset to a textual representation, i.e. “123.45”.

`string amount_to_string(const asset &amount) const`

Convert an asset to a textual representation, i.e. “123.45”.

`string amount_to_pretty_string(share_type amount) const`

Convert an asset to a textual representation with symbol, i.e. “123.45 USD”.

`string amount_to_pretty_string(const asset &amount) const`

Convert an asset to a textual representation with symbol, i.e. “123.45 USD”.

`template <class DB>`

`share_type reserved(const DB &db) const`

The total amount of an asset that is reserved for future issuance.

Public Members

`string symbol`

Ticker symbol for this asset, i.e. “USD”.

`uint8_t precision = 0`

Maximum number of digits after the decimal point (must be ≤ 12)

`account_id_type issuer`

ID of the account which issued this asset.

`asset_dynamic_data_id_type dynamic_asset_data_id`

Current supply, fee pool, and collected fees are stored in a separate object as they change frequently.

`optional<asset_bitasset_data_id_type> bitasset_data_id`

Extra data associated with BitAssets. This field is non-null if and only if `is_market_issued()` returns true.

Public Static Functions

`static bool is_valid_symbol(const string &symbol)`

This function does not check if any registered asset has this symbol or not; it simply checks whether the symbol would be valid.

Return true if symbol is a valid ticker symbol; false otherwise.

`struct asset_options`

#include <asset_ops.hpp> The `asset_options` struct contains options available on all assets in the network.

Note Changes to this struct will break protocol compatibility

Public Functions

`void validate() const`

Perform internal consistency checks.

Exceptions

- `fc::exception`: if any check fails

Public Members

`share_type max_supply = GRAPHENE_MAX_SHARE_SUPPLY`

The maximum supply of this asset which may exist at any given time. This can be as large as `GRAPHENE_MAX_SHARE_SUPPLY`

`uint16_t market_fee_percent = 0`

When this asset is traded on the markets, this percentage of the total traded will be exacted and paid to the issuer. This is a fixed point value, representing hundredths of a percent, i.e. a value of 100 in this field means a 1% fee is charged on market trades of this asset.

`share_type max_market_fee = GRAPHENE_MAX_SHARE_SUPPLY`

Market fees calculated as `market_fee_percent` of the traded volume are capped to this value.

`uint16_t issuer_permissions = UIA_ASSET_ISSUER_PERMISSION_MASK`

The flags which the issuer has permission to update. See `asset_issuer_permission_flags`.

`uint16_t flags = 0`

The currently active flags on this permission. See `asset_issuer_permission_flags`.

`price core_exchange_rate`

When a non-core asset is used to pay a fee, the blockchain must convert that asset to core asset in order to accept the fee. If this asset's fee pool is funded, the chain will automatically deposit fees in this asset to its accumulated fees, and withdraw from the fee pool the same amount as converted at the core exchange rate.

`flat_set<account_id_type> whitelistAuthorities`

A set of accounts which maintain whitelists to consult for this asset. If `whitelistAuthorities` is non-empty, then only accounts in `whitelistAuthorities` are allowed to hold, use, or transfer the asset.

flat_set<account_id_type> **blacklist_authorities**

A set of accounts which maintain blacklists to consult for this asset. If flags & white_list is set, an account may only send, receive, trade, etc. in this asset if none of these accounts appears in its *account_object::blacklisting_accounts* field. If the account is blacklisted, it may not transact in this asset even if it is also whitelisted.

flat_set<asset_id_type> **whitelist_markets**

defines the assets that this asset may be traded against in the market

flat_set<asset_id_type> **blacklist_markets**

defines the assets that this asset may not be traded against in the market, must not overlap whitelist

string **description**

data that describes the meaning/purpose of this asset, fee will be charged proportional to size of description.

struct **asset_publish_feed_operation**

#include <asset_ops.hpp> 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 eachother, 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.

Inherits from *graphene::chain::base_operation*

Public Members

asset **fee**

paid for by publisher

asset_id_type **asset_id**

asset for which the feed is published

class **asset_publish_feeds_evaluator**

#include <asset_evaluator.hpp> Inherits from *graphene::chain::evaluator<asset_publish_feeds_evaluator>*

class **asset_reserve_evaluator**

#include <asset_evaluator.hpp> Inherits from *graphene::chain::evaluator<asset_reserve_evaluator>*

struct **asset_reserve_operation**

#include <asset_ops.hpp> used to take an asset out of circulation, returning to the issuer

Note You cannot use this operation on market-issued assets.

Inherits from *graphene::chain::base_operation*

struct **asset_settle_cancel_operation**

#include <asset_ops.hpp> Virtual op generated when force settlement is cancelled.

Inherits from *graphene::chain::base_operation*

Public Members

`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.

`class asset_settle_evaluator`

`#include <asset_evaluator.hpp>` Inherits from `graphene::chain::evaluator<asset_settle_evaluator>`

`struct asset_settle_operation`

`#include <asset_ops.hpp>` 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'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

Inherits from `graphene::chain::base_operation`

Public Members

`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.

`struct fee_parameters_type`

`#include <asset_ops.hpp>`

Public Members

`uint64_t fee = 100 * GRAPHENE_BLOCKCHAIN_PRECISION`

this fee should be high to encourage small settlement requests to be performed on the market rather than via forced settlement.

Note that in the event of a black swan or prediction market close out everyone will have to pay this fee.

`struct asset_symbol_eq_lit_predicate`

`#include <assert.hpp>` Used to verify that asset_id->symbol is equal to the given string literal.

Public Functions

`bool validate() const`

Perform state independent checks. Verify symbol is a valid asset symbol.

```
class asset_update_bitasset_evaluator
    #include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_update_bitasset_evaluator>
```

```
struct asset_update_bitasset_operation
    #include <asset_ops.hpp> 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 on an existing BitAsset.

Pre issuer MUST be an existing account and MUST match `asset_object::issuer` on `asset_to_update`

Pre `asset_to_update` MUST be a BitAsset, i.e. `asset_object::is_market_issued()` returns true

Pre fee MUST be nonnegative, and issuer MUST have a sufficient balance to pay it

Pre new_options SHALL be internally consistent, as verified by validate()

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

Inherits from `graphene::chain::base_operation`

```
class asset_update_evaluator
```

```
#include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_update_evaluator>
```

```
class asset_update_feed_producers_evaluator
```

```
#include <asset_evaluator.hpp> Inherits from graphene::chain::evaluator<asset_update_feed_producers_evaluator>
```

```
struct asset_update_feed_producers_operation
```

```
#include <asset_ops.hpp> 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.

Pre issuer MUST be an existing account, and MUST match `asset_object::issuer` on `asset_to_update`

Pre issuer MUST NOT be the committee account

Pre `asset_to_update` MUST be a BitAsset, i.e. `asset_object::is_market_issued()` returns true

Pre fee MUST be nonnegative, and issuer MUST have a sufficient balance to pay it

Pre Cardinality of new_feed_producers MUST NOT exceed `chain_parameters::maximum_asset_feed_publishers`

Post `asset_to_update` will have a set of feed producers matching new_feed_producers

Post 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

Inherits from `graphene::chain::base_operation`

```
struct asset_update_operation
```

```
#include <asset_ops.hpp> 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.

Pre issuer SHALL be an existing account and MUST match `asset_object::issuer` on `asset_to_update`

Pre fee SHALL be nonnegative, and issuer MUST have a sufficient balance to pay it

Pre new_options SHALL be internally consistent, as verified by validate()

Post asset_to_update will have options matching those of new_options

Inherits from *graphene::chain::base_operation*

Public Members

optional<account_id_type> new_issuer

If the asset is to be given a new issuer, specify his ID here.

class authority

#include <authority.hpp> Identifies a weighted set of keys and accounts that must approve operations.

Public Members

flat_map<address, weight_type> address_auths

needed for backward compatibility only

class balance_claim_evaluator

#include <balance_evaluator.hpp> Inherits from *graphene::chain::evaluator<balance_claim_evaluator>*

Public Functions

void_result do_apply (const balance_claim_operation &op)

Note the fee is always 0 for this particular operation because once the balance is claimed it frees up memory and it cannot be used to spam the network

struct balance_claim_operation

#include <balance.hpp> Claim a balance in a balance_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.

Inherits from *graphene::chain::base_operation*

class balance_object

#include <balance_object.hpp> Inherits from *graphene::db::abstract_object<balance_object>*

struct base_operation

#include <base.hpp> Subclassed by *graphene::chain::account_create_operation*,
graphene::chain::account_update_operation,
graphene::chain::account_upgrade_operation,
graphene::chain::assert_operation,
graphene::chain::asset_create_operation,
graphene::chain::asset_global_settle_operation,
graphene::chain::asset_publish_feed_operation,
graphene::chain::asset_settle_cancel_operation,
graphene::chain::asset_update_bitasset_operation, *graphene::chain::asset_update_feed_producers_operation*,
graphene::chain::asset_update_operation, *graphene::chain::balance_claim_operation*,

```

graphene::chain::blind_transfer_operation,
graphene::chain::committee_member_create_operation, graphene::chain::committee_member_update_global_parameters,
graphene::chain::committee_member_update_operation, graphene::chain::custom_operation,
graphene::chain::fba_distribute_operation, graphene::chain::fill_order_operation,
graphene::chain::limit_order_cancel_operation, graphene::chain::limit_order_create_operation,
graphene::chain::override_transfer_operation, graphene::chain::proposal_create_operation,
graphene::chain::proposal_delete_operation, graphene::chain::proposal_update_operation,
graphene::chain::transfer_from_blind_operation, graphene::chain::transfer_operation,
graphene::chain::transfer_to_blind_operation, graphene::chain::vesting_balance_create_operation,
graphene::chain::vesting_balance_withdraw_operation, graphene::chain::withdraw_permission_claim_operation,
graphene::chain::withdraw_permission_create_operation, graphene::chain::withdraw_permission_delete_operation,
graphene::chain::withdraw_permission_update_operation, graphene::chain::witness_create_operation,
graphene::chain::witness_update_operation, graphene::chain::worker_create_operation

```

struct bitasset_options

#include <asset_ops.hpp> The *bitasset_options* struct contains configurable options available only to BitAssets.

Note Changes to this struct will break protocol compatibility

Public Functions

void **validate()** const

Perform internal consistency checks.

Exceptions

- `fc::exception`: if any check fails

Public Members

`uint32_t feed_lifetime_sec = GRAPHENE_DEFAULT_PRICE_FEED_LIFETIME`

Time before a price feed expires.

`uint8_t minimum_feeds = 1`

Minimum number of unexpired feeds required to extract a median feed from.

`uint32_t force_settlement_delay_sec = GRAPHENE_DEFAULT_FORCE_SETTLEMENT_DELAY`

This is the delay between the time a long requests settlement and the chain evaluates the settlement.

`uint16_t force_settlement_offset_percent = GRAPHENE_DEFAULT_FORCE_SETTLEMENT_OFFSET`

This is the percent to adjust the feed price in the short's favor in the event of a forced settlement.

`uint16_t maximum_force_settlement_volume = GRAPHENE_DEFAULT_FORCE_SETTLEMENT_MAX_VOL`

Force settlement volume can be limited such that only a certain percentage of the total existing supply of the asset may be force-settled within any given chain maintenance interval. This field stores the percentage of the current supply which may be force settled within the current maintenance interval. If force settlements come due in an interval in which the maximum volume has already been settled, the new settlements will be enqueued and processed at the beginning of the next maintenance interval.

`asset_id_type short_backing_asset`

This specifies which asset type is used to collateralize short sales. This field may only be updated if the current supply of the asset is zero.

struct blind_input

#include <confidential.hpp>

Public Members

`authority owner`

provided to maintain the invariant that all authority required by an operation is explicit in the operation. Must match blinded_balance_id->owner

`struct blind_memo`

`#include <confidential.hpp>` This data is encrypted and stored in the encrypted memo portion of the blind output.

Public Members

`uint32_t check = 0`

set to the first 4 bytes of the shared secret used to encrypt the memo. Used to verify that decryption was successful.

`class blind_output`

`#include <confidential.hpp>` Defines data required to create a new blind commitment

The blinded output that must be proven to be greater than 0.

Public Members

`range_proof_type range_proof`

only required if there is more than one blind output

`class blind_transfer_evaluator`

`#include <confidential_evaluator.hpp>` Inherits from `graphene::chain::evaluator<blind_transfer_evaluator>`

Public Functions

`void pay_fee()`

Routes the fee to where it needs to go. The default implementation routes the fee to the `account_statistics_object` of the `fee_paying_account`.

Before `pay_fee()` is called, the fee is computed by `prepare_fee()` and has been moved out of the `fee_paying_account` and (if paid in a non-CORE asset) converted by the asset's fee pool.

Therefore, when `pay_fee()` is called, the fee only exists in `this->core_fee_paid`. So `pay_fee()` need only increment the receiving balance.

The default implementation simply calls `account_statistics_object->pay_fee()` to increment `pending_fees` or `pending_vested_fees`.

`struct blind_transfer_operation`

`#include <confidential.hpp>` Transfers from blind to blind.

There are two ways to transfer value while maintaining privacy:

1. account to account with amount kept secret
2. 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.

Inherits from [graphene::chain::base_operation](#)

Public Functions

```
account_id_type fee_payer() const
graphene TEMP account
```

If fee_payer = temp_account_id, then the fee is paid by the surplus balance of inputs-outputs and 100% of the fee goes to the network.

```
void validate() const
```

This method can be computationally intensive because it verifies that input commitments - output commitments add up to 0

```
struct fee_parameters_type
#include <confidential.hpp>
```

Public Members

```
uint64_t fee = 5*GRAPHENE_BLOCKCHAIN_PRECISION
the cost to register the cheapest non-free account
```

```
class blinded_balance_object
```

#include <confidential_object.hpp> tracks a blinded balance commitment

Inherits from graphene::db::abstract_object<blinded_balance_object>

```
struct block_header
```

#include <block.hpp> Subclassed by [graphene::chain::signed_block_header](#)

```
struct block_id_predicate
#include <assert.hpp> Used to verify that a specific block is part of the blockchain history. This helps
protect some high-value transactions to newly created IDs

The block ID must be within the last 2^16 blocks.

class block_summary_object
#include <block_summary_object.hpp> tracks minimal information about past blocks to implement
TaPOS

When attempting to calculate the validity of a transaction we need to lookup a past block and check its
block hash and the time it occurred so we can calculate whether the current transaction is valid and at what
time it should expire.

Inherits from graphene::db::abstract_object< block_summary_object >

class budget_record_object
#include <budget_record_object.hpp> Inherits from graphene::db::abstract_object< bud-
get_record_object >

struct burn_worker_type
#include <worker_object.hpp> A worker who permanently destroys all of his pay.

This worker sends all pay he receives to the null account.
```

Public Members

```
share_type total_burned
Record of how much this worker has burned in his lifetime.
```

```
struct buyback_account_options
#include <buyback.hpp>
```

Public Members

```
asset_id_type asset_to_buy
The asset to buy.
```

```
account_id_type asset_to_buy_issuer
Issuer of the asset. Must sign the transaction, must match issuer of specified asset.
```

```
flat_set<asset_id_type> markets
What assets the account is willing to buy with. Other assets will just sit there since the account has
null authority.
```

```
class buyback_object
#include <buyback_object.hpp> buyback_authority_object only exists to help with a specific indexing
problem. We want to be able to iterate over all assets that have a buyback program. However, assets
which have a buyback program are very rare. So rather than indexing asset_object by the buyback field
(requiring additional bookkeeping for every asset), we instead maintain a buyback_object pointing to each
asset which has buyback (requiring additional bookkeeping only for every asset which has buyback).
```

This class is an implementation detail.

```
Inherits from graphene::db::abstract_object< buyback_object >
```

class call_order_object

#include <market_object.hpp> tracks debt and call price information

There should only be one *call_order_object* per asset pair per account and they will all have the same call price.

Inherits from graphene::db::abstract_object< call_order_object >

Public Members***share_type collateral***

call_price.base.asset_id, access via get_collateral

share_type debt

call_price.quote.asset_id, access via get_collateral

price call_price

Debt / Collateral.

class call_order_update_evaluator

#include <market_evaluator.hpp> Inherits from graphene::chain::evaluator< call_order_update_evaluator >

struct call_order_update_operation

#include <market.hpp> 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.

Inherits from graphene::chain::base_operation

Public Members***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

struct fee_parameters_type

#include <market.hpp> this is slightly more expensive than limit orders, this pricing impacts prediction markets

struct cdd_vesting_policy

#include <vesting_balance_object.hpp> defines vesting in terms of coin-days accrued which allows for dynamic deposit/withdraw

The economic effect of this vesting policy is to require a certain amount of “interest” to accrue before the full balance may be withdrawn. Interest accrues as coindays (balance * length held). If some of the balance is withdrawn, the remaining balance must be held longer.

Public Functions

```
fc::uint128_t compute_coin_seconds_earned(const vesting_policy_context &ctx)  
    const  
Compute coin_seconds_earned. Used to non-destructively figure out how many coin seconds are available.  
  
void update_coin_seconds_earned(const vesting_policy_context &ctx)  
    Update coin_seconds_earned and coin_seconds_earned_last_update fields; called by both on_deposit() and on_withdraw().
```

Public Members

```
fc::time_point_sec start_claim  
while coindays may accrue over time, none may be claimed before first_claim date  
  
struct cdd_vesting_policy_initializer  
#include <vesting.hpp>
```

Public Members

```
fc::time_point_sec start_claim  
while coindays may accrue over time, none may be claimed before the start_claim time  
  
struct chain_parameters  
#include <chain_parameters.hpp>
```

Public Functions

```
void validate() const  
defined in fee_schedule.cpp
```

Public Members

```
smart_ref<fee_schedule> current_fees  
current schedule of fees  
  
using a smart ref breaks the circular dependency created between operations and the fee schedule  
  
uint8_t block_interval = GRAPHENE_DEFAULT_BLOCK_INTERVAL  
interval in seconds between blocks  
  
uint32_t maintenance_interval = GRAPHENE_DEFAULT_MAINTENANCE_INTERVAL  
interval in sections between blockchain maintenance events  
  
uint8_t maintenance_skip_slots = GRAPHENE_DEFAULT_MAINTENANCE_SKIP_SLOTS  
number of block_intervals to skip at maintenance time
```

```

uint32_t committee_proposal_review_period = GRAPHENE_DEFAULT_COMMITTEE_PROPOSAL_REVIE
    minimum time in seconds that a proposed transaction requiring committee authority may not be
    signed, prior to expiration

uint32_t maximum_transaction_size = GRAPHENE_DEFAULT_MAX_TRANSACTION_SIZE
    maximum allowable size in bytes for a transaction

uint32_t maximum_block_size = GRAPHENE_DEFAULT_MAX_BLOCK_SIZE
    maximum allowable size in bytes for a block

uint32_t maximum_time_until_expiration = GRAPHENE_DEFAULT_MAX_TIME_UNTIL_EXPIRATION
    maximum lifetime in seconds for transactions to be valid, before expiring

uint32_t maximum_proposal_lifetime = GRAPHENE_DEFAULT_MAX_PROPOSAL_LIFETIME_SEC
    maximum lifetime in seconds for proposed transactions to be kept, before expiring

uint8_t maximum_asset_whitelistAuthorities = GRAPHENE_DEFAULT_MAX_ASSET_WHITELIST_AU
    maximum number of accounts which an asset may list as authorities for its whitelist OR blacklist

uint8_t maximum_asset_feed_publishers = GRAPHENE_DEFAULT_MAX_ASSET_FEED_PUBLISHERS
    the maximum number of feed publishers for a given asset

uint16_t maximum_witness_count = GRAPHENE_DEFAULT_MAX_WITNESSES
    maximum number of active witnesses

uint16_t maximum_committee_count = GRAPHENE_DEFAULT_MAX_COMMITTEE
    maximum number of active committee_members

uint16_t maximum_authority_membership = GRAPHENE_DEFAULT_MAX_AUTHORITY_MEMBERSHIP
    largest number of keys/accounts an authority can have

uint16_t reserve_percent_of_fee = GRAPHENE_DEFAULT_BURN_PERCENT_OF_FEE
    the percentage of the network's allocation of a fee that is taken out of circulation

uint16_t network_percent_of_fee = GRAPHENE_DEFAULT_NETWORK_PERCENT_OF_FEE
    percent of transaction fees paid to network

uint16_t lifetime_referrer_percent_of_fee = GRAPHENE_DEFAULT_LIFETIME_REFERRER_PERCENT
    percent of transaction fees paid to network

uint32_t cashback_vesting_period_seconds = GRAPHENE_DEFAULT_CASHBACK_VESTING_PERIOD_SE
    time after cashback rewards are accrued before they become liquid

share_type cashback_vesting_threshold = GRAPHENE_DEFAULT_CASHBACK_VESTING_THRESHOLD
    the maximum cashback that can be received without vesting

bool count_non_member_votes = true
    set to false to restrict voting privileges to member accounts

bool allow_non_member_whitelists = false
    true if non-member accounts may set whitelists and blacklists; false otherwise

share_type witness_pay_per_block = GRAPHENE_DEFAULT_WITNESS_PAY_PER_BLOCK
    CORE to be allocated to witnesses (per block)

uint32_t witness_pay_vesting_seconds = GRAPHENE_DEFAULT_WITNESS_PAY_VESTING_SECONDS
    vesting_seconds parameter for witness VBO's

share_type worker_budget_per_day = GRAPHENE_DEFAULT_WORKER_BUDGET_PER_DAY
    CORE to be allocated to workers (per day)

uint16_t max_predicate_opcode = GRAPHENE_DEFAULT_MAX_ASSERT_OPCODE
    predicate_opcode must be less than this number

```

```
share_type fee_liquidation_threshold = GRAPHENE_DEFAULT_FEE_LIQUIDATION_THRESHOLD
    value in CORE at which accumulated fees in blockchain-issued market assets should be liquidated

uint16_t accounts_per_fee_scale = GRAPHENE_DEFAULT_ACCOUNTS_PER_FEE_SCALE
    number of accounts between fee scalings

uint8_t account_fee_scale_bitshifts = GRAPHENE_DEFAULT_ACCOUNT_FEE_SCALE_BITSHIFTS
    number of times to left bitshift account registration fee at each scaling

class chain_property_object
    #include <chain_property_object.hpp> Contains invariants which are set at genesis and never changed.

    Inherits from graphene::db::abstract_object<chain_property_object>

class committee_member_create_evaluator
    #include <committee_member_evaluator.hpp> Inherits from graphene::chain::evaluator< committee_member_create_evaluator >

struct committee_member_create_operation
    #include <committee_member.hpp> 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.

    Inherits from graphene::chain::base_operation
```

Public Members

```
account_id_type committee_member_account
    The account which owns the committee_member. This account pays the fee for this operation.

class committee_member_object
    #include <committee_member_object.hpp> tracks information about a committee_member account.

    A committee_member is responsible for setting blockchain parameters and has dynamic multi-sig control over the committee account. The current set of active committee_members has control.

    committee_members were separated into a separate object to make iterating over the set of committee_member easy.

    Inherits from graphene::db::abstract_object<committee_member_object>

class committee_member_update_evaluator
    #include <committee_member_evaluator.hpp> Inherits from graphene::chain::evaluator< committee_member_update_evaluator >

class committee_member_update_global_parameters_evaluator
    #include <committee_member_evaluator.hpp> Inherits from graphene::chain::evaluator< committee_member_update_global_parameters_evaluator >

struct committee_member_update_global_parameters_operation
    #include <committee_member.hpp> 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.

Inherits from [graphene::chain::base_operation](#)

```
struct committee_member_update_operation
#include <committee_member.hpp> Update a committee_member object.
```

Currently the only field which can be updated is the `url` field.

Inherits from [graphene::chain::base_operation](#)

Public Members

`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.

```
class custom_evaluator
```

#include <custom_evaluator.hpp> Inherits from [graphene::chain::evaluator<custom_evaluator>](#)

```
struct custom_operation
```

#include <custom.hpp> 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.

Inherits from [graphene::chain::base_operation](#)

```
class database
```

#include <database.hpp> tracks the blockchain state in an extensible manner

Inherits from `object_database`

Unnamed Group

`void globally_settle_asset (const asset_object &bitasset, const price &settle_price)`

Market Helpers

All margin positions are force closed at the swan price Collateral received goes into a force-settlement fund No new margin positions can be created for this asset No more price feed updates Force settlement happens without delay at the swan price, deducting from force-settlement fund No more asset updates may be issued.

```
bool apply_order (const limit_order_object &new_order_object, bool allow_black_swan =
    true)
```

Process a new limit order through the markets.

This function takes a new limit order, and runs the markets attempting to match it with existing orders already on the books.

Return true if order was completely filled; false otherwise

Parameters

- `order`: The new order to process

```
template <typename OrderType>
```

```
int match (const limit_order_object &bid, const OrderType &ask, const price &match_price)  
Matches the two orders,
```

0 - no orders were matched 1 - bid was filled 2 - ask was filled 3 - both were filled

Return a bit field indicating which orders were filled (and thus removed)

```
asset match (const call_order_object &call, const force_settlement_object &settle, const  
price &match_price, asset max_settlement)  
Return the amount of asset settled
```

Public Functions

```
void open (const fc::path &data_dir, std::function<genesis_state_type> > genesis_loader  
Open a database, creating a new one if necessary.
```

Opens a database in the specified directory. If no initialized database is found, genesis_loader is called and its return value is used as the genesis state when initializing the new database

genesis_loader will not be called if an existing database is found.

Parameters

- *data_dir*: Path to open or create database in
- *genesis_loader*: A callable object which returns the genesis state to initialize new databases on

```
void reindex (fc::path data_dir, const genesis_state_type &initial_allocation = genesis_state_type ())  
Rebuild object graph from block history and open database.
```

This method may be called after or instead of *database::open*, and will rebuild the object graph by replaying blockchain history. When this method exits successfully, the database will be open.

```
void wipe (const fc::path &data_dir, bool include_blocks)  
wipe Delete database from disk, and potentially the raw chain as well.
```

Will close the database before wiping. Database will be closed when this function returns.

Parameters

- *include_blocks*: If true, delete the raw chain as well as the database.

```
bool is_known_block (const block_id_type &id) const  
Return true if the block is in our fork DB or saved to disk as part of the official chain, otherwise  
return false
```

```
bool is_known_transaction (const transaction_id_type &id) const  
Only return true if the transaction has not expired or been invalidated. If this method is called with a  
VERY old transaction we will return false, they should query things by blocks if they are that old.
```

```
uint32_t witness_participation_rate () const  
Calculate the percent of block production slots that were missed in the past 128 blocks, not including  
the current block.
```

```
bool push_block (const signed_block &b, uint32_t skip = skip_nothing)  
Push block “may fail” in which case every partial change is unwound. After push block is successful  
the block is appended to the chain database on disk.
```

Return true if we switched forks as a result of this push.

```
processed_transaction push_transaction (const signed_transaction &trx, uint32_t skip =  
                          skip_nothing)
```

Attempts to push the transaction into the pending queue

When called to push a locally generated transaction, set the skip_block_size_check bit on the skip argument. This will allow the transaction to be pushed even if it causes the pending block size to exceed the maximum block size. Although the transaction will probably not propagate further now, as the peers are likely to have their pending queues full as well, it will be kept in the queue to be propagated later when a new block flushes out the pending queues.

```
processed_transaction push_proposal (const proposal_object &proposal)
```

Exceptions

- `fc::exception`: if the proposed transaction fails to apply.

```
void pop_block ()
```

Removes the most recent block from the database and undoes any changes it made.

```
uint32_t push_applied_operation (const operation &op)
```

This method is used to track applied operations during the evaluation of a block, these operations should include any operation actually included in a transaction as well as any implied/virtual operations that resulted, such as filling an order. The applied operations is cleared after applying each block and calling the block observers which may want to index these operations.

Return the op_id which can be used to set the result after it has finished being applied.

```
witness_id_type get_scheduled_witness (uint32_t slot_num) const
```

Get the witness scheduled for block production in a slot.

slot_num always corresponds to a time in the future.

If slot_num == 1, returns the next scheduled witness. If slot_num == 2, returns the next scheduled witness after 1 block gap.

Use the `get_slot_time()` and `get_slot_at_time()` functions to convert between slot_num and timestamp.

Passing slot_num == 0 returns GRAPHENE_NULL_WITNESS

```
fc::time_point_sec get_slot_time (uint32_t slot_num) const
```

Get the time at which the given slot occurs.

If slot_num == 0, return time_point_sec().

If slot_num == N for N > 0, return the Nth next block-interval-aligned time greater than head_block_time().

```
uint32_t get_slot_at_time (fc::time_point_sec when) const
```

Get the last slot which occurs AT or BEFORE the given time.

The return value is the greatest value N such that `get_slot_time(N) <= when`.

If no such N exists, return 0.

```
void initialize_indexes ()
```

Reset the object graph in-memory.

```
asset get_balance (account_id_type owner, asset_id_type asset_id) const
```

Retrieve a particular account's balance in a given asset.

Return owner's balance in asset

Parameters

- `owner`: Account whose balance should be retrieved
- `asset_id`: ID of the asset to get balance in

```
asset get_balance (const account_object &owner, const asset_object &asset_obj) const
This is an overloaded method.
```

```
void adjust_balance (account_id_type account, asset delta)
Adjust a particular account's balance in a given asset by a delta.
```

Parameters

- account: ID of account whose balance should be adjusted
- delta: Asset ID and amount to adjust balance by

```
optional<vesting_balance_id_type> deposit_lazy_vesting (const vesting_balance_id_type &ovbid, share_type amount, uint32_t req_vesting_seconds, account_id_type req_owner, bool require_vesting)
```

Helper to make lazy deposit to CDD VBO.

If the given optional VBID is not valid(), or it does not have a CDD vesting policy, or the owner / vesting_seconds of the policy does not match the parameter, then credit amount to newly created VBID and return it.

Otherwise, credit amount to ovbid.

Return ID of newly created VBO, but only if VBO was created.

```
void debug_dump ()
```

This method dumps the state of the blockchain in a semi-human readable form for the purpose of tracking down funds and mismatches in currency allocation

```
bool fill_order (const limit_order_object &order, const asset &pays, const asset &receives, bool cull_if_small)
```

Return true if the order was completely filled and thus freed.

```
bool check_call_orders (const asset_object &mia, bool enable_black_swan = true)
```

Starting with the least collateralized orders, fill them if their call price is above the max(lowest bid,call_limit).

This method will return true if it filled a short or limit

Return true if a margin call was executed.

Parameters

- mia: - the market issued asset that should be called.
- enable_black_swan: - when adjusting collateral, triggering a black swan is invalid and will throw if enable_black_swan is not set to true.

```
processed_transaction validate_transaction (const signed_transaction &trx)
```

This method validates transactions without adding it to the pending state.

Return true if the transaction would validate

Public Members

```
fc::signal<void (const signed_block&) > applied_block
```

This signal is emitted after all operations and virtual operation for a block have been applied but before the get_applied_operations() are cleared.

You may not yield from this callback because the blockchain is holding the write lock and may be in an “inconstant state” until after it is released.

`fc::signal<void (const signed_transaction&) > on_pending_transaction`

This signal is emitted any time a new transaction is added to the pending block state.

`fc::signal<void (const vector<object_id_type>&, const flat_set<account_id_type>&) > new_objects`

Emitted After a block has been applied and committed. The callback should not yield and should execute quickly.

`fc::signal<void (const vector<object_id_type>&, const flat_set<account_id_type>&) > changed_objects`

Emitted After a block has been applied and committed. The callback should not yield and should execute quickly.

`fc::signal<void (const vector<object_id_type>&, const vector<const object *>&, const flat_set<account_id_type>&) > removed_objects`

this signal is emitted any time an object is removed and contains a pointer to the last value of every object that was removed.

`std::deque<signed_transaction> _popped_tx`

when popping a block, the transactions that were removed get cached here so they can be reapplied at the proper time

`class dynamic_global_property_object`

`#include <global_property_object.hpp>` Maintains global state information (committee_member list, current fees)

This is an implementation detail. The values here are calculated during normal chain operations and reflect the current values of global blockchain properties.

Inherits from `graphene::db::abstract_object<dynamic_global_property_object>`

Public Members

`uint32_t recently_missed_count = 0`

Every time a block is missed this increases by RECENTLY_MISSED_COUNT_INCREMENT, every time a block is found it decreases by RECENTLY_MISSED_COUNT_DECREMENT. It is never less than 0.

If the recently_missed_count hits 2*UNDO_HISTORY then no new blocks may be pushed.

`uint64_t current_aslot = 0`

The current absolute slot number. Equal to the total number of slots since genesis. Also equal to the total number of missed slots plus head_block_number.

`fc::uint128_t recent_slots_filled`

used to compute witness participation.

`uint32_t dynamic_flags = 0`

dynamic_flags specifies chain state properties that can be expressed in one bit.

`template <typename DerivedEvaluator>`

`class evaluator`

`#include <evaluator.hpp>` Inherits from `graphene::chain::generic_evaluator`

Public Functions

`virtual operation_result evaluate (const operation &op)`

Note derived classes should ASSUME that the default validation that is independent of chain state should be performed by op.validate() and should not perform these extra checks.

```
class fba_accumulator_object
#include <fba_object.hpp> fba_accumulator_object accumulates fees to be paid out via buyback or other FBA mechanism.

Inherits from graphene::db::abstract_object< fba_accumulator_object >

struct fba_distribute_operation
#include <fba.hpp> Inherits from graphene::chain::base_operation

struct fee_schedule
#include <fee_schedule.hpp> contains all of the parameters necessary to calculate the fee for any operation
```

Public Functions

```
asset calculate_fee(const operation &op, const price &core_exchange_rate =
price::unit_price ()) const
Finds the appropriate fee parameter struct for the operation and then calculates the appropriate fee.
```

```
void validate() const
Validates all of the parameters are present and accounted for.
```

Public Members

```
flat_set<fee_parameters> parameters
```

Note must be sorted by fee_parameters.which() and have no duplicates

```
uint32_t scale = GRAPHENE_100_PERCENT
fee * scale / GRAPHENE_100_PERCENT
```

```
struct fill_order_operation
#include <market.hpp>
```

Note This is a virtual operation that is created while matching orders and emitted for the purpose of accurately tracking account history, accelerating a reindex.

Inherits from graphene::chain::base_operation

Public Functions

```
share_type calculate_fee(const fee_parameters_type &k) const
```

This is a virtual operation; there is no fee.

```
class force_settlement_object
```

#include <market_object.hpp> tracks bitassets scheduled for force settlement at some point in the future.

On the settlement_date the balance will be converted to the collateral asset and paid to owner and then this object will be deleted.

Inherits from graphene::db::abstract_object< force_settlement_object >

class fork_database

#include <fork_database.hpp> As long as blocks are pushed in order the fork database will maintain a linked tree of all blocks that branch from the start_block. The tree will have a maximum depth of 1024 blocks after which the database will start lopping off forks.

Every time a block is pushed into the fork DB the block with the highest block_num will be returned.

Public Functions

```
shared_ptr<fork_item> push_block (const signed_block &b)
```

Pushes the block into the fork database and caches it if it doesn't link

Return the new head block (the longest fork)

```
pair<fork_database::branch_type,fork_database::branch_type> fetch_branch_from (block_id_type
first,
block_id_type
second)
const
```

Given two head blocks, return two branches of the fork graph that end with a common ancestor (same prior block)

Public Static Attributes

```
const int MAX_BLOCK_REORDERING = 1024
```

The maximum number of blocks that may be skipped in an out-of-order push.

```
struct fork_item
#include <fork_database.hpp>
```

Public Members

```
bool invalid = false
```

Used to flag a block as invalid and prevent other blocks from building on top of it.

class generic_evaluator

```
#include <evaluator.hpp>
count_create_evaluator >, Subclassed by graphene::chain::evaluator< ac-
graphene::chain::evaluator< account_update_evaluator >, graphene::chain::evaluator<
account_upgrade_evaluator >, graphene::chain::evaluator< assert_evaluator >,
graphene::chain::evaluator< asset_claim_fees_evaluator >, graphene::chain::evaluator<
asset_create_evaluator >, graphene::chain::evaluator< asset_fund_fee_pool_evaluator >,
graphene::chain::evaluator< asset_global_settle_evaluator >, graphene::chain::evaluator<
asset_issue_evaluator >, graphene::chain::evaluator< asset_publish_feeds_evaluator >,
graphene::chain::evaluator< asset_reserve_evaluator >, graphene::chain::evaluator< as-
set_settle_evaluator >, graphene::chain::evaluator< asset_update_bitasset_evaluator >,
graphene::chain::evaluator< asset_update_evaluator >, graphene::chain::evaluator< bal-
ance_claim_evaluator >, graphene::chain::evaluator< blind_transfer_evaluator >,
graphene::chain::evaluator< call_order_update_evaluator >, graphene::chain::evaluator< com-
mittee_member_create_evaluator >, graphene::chain::evaluator< committee_member_update_evaluator >,
graphene::chain::evaluator< committee_member_update_global_parameters_evaluator >,
graphene::chain::evaluator< custom_evaluator >, graphene::chain::evaluator<
```

```
limit_order_cancel_evaluator >, graphene::chain::evaluator< limit_order_create_evaluator
>, graphene::chain::evaluator< override_transfer_evaluator >, graphene::chain::evaluator<
proposal_create_evaluator >, graphene::chain::evaluator< proposal_delete_evaluator >,
graphene::chain::evaluator< proposal_update_evaluator >, graphene::chain::evaluator<
transfer_evaluator >, graphene::chain::evaluator< transfer_from_blind_evaluator >,
graphene::chain::evaluator< transfer_to_blind_evaluator >, graphene::chain::evaluator<
vesting_balance_create_evaluator >, graphene::chain::evaluator< vesting_balance_withdraw_evaluator
>, graphene::chain::evaluator< withdraw_permission_claim_evaluator >,
graphene::chain::evaluator< withdraw_permission_create_evaluator >, graphene::chain::evaluator<
withdraw_permission_delete_evaluator >, graphene::chain::evaluator< withdraw_permission_update_evaluator >,
graphene::chain::evaluator< witness_create_evaluator >, graphene::chain::evaluator<
witness_update_evaluator >, graphene::chain::evaluator< worker_create_evaluator >, graphene::chain::evaluator< DerivedEvaluator >
```

Public Functions

```
virtual operation_result evaluate(const operation &op) = 0
```

Note derived classes should ASSUME that the default validation that is independent of chain state should be performed by op.validate() and should not perform these extra checks.

```
void pay_fee()
```

Routes the fee to where it needs to go. The default implementation routes the fee to the *account_statistics_object* of the fee_paying_account.

Before *pay_fee()* is called, the fee is computed by *prepare_fee()* and has been moved out of the fee_paying_account and (if paid in a non-CORE asset) converted by the asset's fee pool.

Therefore, when *pay_fee()* is called, the fee only exists in this->core_fee_paid. So *pay_fee()* need only increment the receiving balance.

The default implementation simply calls account_statistics_object->*pay_fee()* to increment pending_fees or pending_vested_fees.

```
struct genesis_state_type
#include <genesis_state.hpp>
```

Public Functions

```
chain_id_type compute_chain_id() const
```

Get the chain_id corresponding to this genesis state.

This is the SHA256 serialization of the genesis_state.

Public Members

```
chain_id_type initial_chain_id
```

Temporary, will be moved elsewhere.

```
struct initial_committee_member_type
#include <genesis_state.hpp>
```

Public Members

```
string owner_name
      Must correspond to one of the initial accounts.
```

```
struct initial_witness_type
  #include <genesis_state.hpp>
```

Public Members

```
string owner_name
      Must correspond to one of the initial accounts.
```

```
struct initial_worker_type
  #include <genesis_state.hpp>
```

Public Members

```
string owner_name
      Must correspond to one of the initial accounts.
```

```
class global_property_object
  #include <global_property_object.hpp> Maintains global state information (committee_member list, current fees)
```

This is an implementation detail. The values here are set by committee_members to tune the blockchain parameters.

Inherits from `graphene::db::abstract_object<global_property_object>`

```
class limit_order_cancel_evaluator
  #include <market_evaluator.hpp> Inherits from graphene::chain::evaluator<limit_order_cancel_evaluator>
```

```
struct limit_order_cancel_operation
  #include <market.hpp> 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.
```

Return the amount actually refunded

Inherits from `graphene::chain::base_operation`

Public Members

```
account_id_type fee_paying_account
      must be order->seller
```

```
class limit_order_create_evaluator
  #include <market_evaluator.hpp> Inherits from graphene::chain::evaluator<limit_order_create_evaluator>
```

Public Functions

void **pay_fee()**

override the default behavior defined by generic_evaluator which is to post the fee to fee_paying_account_stats.pending_fees

class limit_order_create_operation

#include <market.hpp> instructs the blockchain to attempt to sell one asset for another

The blockchain will attempt 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 block chain.

Inherits from *graphene::chain::base_operation*

Public Members

time_point_sec **expiration** = time_point_sec::maximum()

The order will be removed from the books if not filled by expiration Upon expiration, all unsold asset will be returned to seller

bool **fill_or_kill** = false

If this flag is set the entire order must be filled or the operation is rejected.

class limit_order_object

#include <market_object.hpp> an offer to sell a amount of a asset at a specified exchange rate by a certain time

This limit_order_objects are indexed by expiration and is automatically deleted on the first block after expiration.

Inherits from *graphene::db::abstract_object< limit_order_object >*

Public Members

share_type for_sale

asset id is sell_price.base.asset_id

struct linear_vesting_policy

#include <vesting_balance_object.hpp> Linear vesting balance with cliff.

This vesting balance type is used to mimic traditional stock vesting contracts where each day a certain amount vests until it is fully matured.

Note New funds may not be added to a linear vesting balance.

Public Members

fc::time_point_sec **begin_timestamp**

This is the time at which funds begin vesting.

```

uint32_t vesting_cliff_seconds = 0
    No amount may be withdrawn before this many seconds of the vesting period have elapsed.

uint32_t vesting_duration_seconds = 0
    Duration of the vesting period, in seconds. Must be greater than 0 and greater than vesting_cliff_seconds.

share_type begin_balance
    The total amount of asset to vest.

struct linear_vesting_policy_initializer
    #include <vesting.hpp>

```

Public Members

fc::time_point_sec begin_timestamp
 while vesting begins on begin_timestamp, none may be claimed before vesting_cliff_seconds have passed

```

struct memo_data
    #include <memo.hpp> defines the keys used to derive the shared secret

```

Because account authorities and keys can change at any time, each memo must capture the specific keys used to derive the shared secret. In order to read the cipher message you will need one of the two private keys.

If from == to and from == 0 then no encryption is used, the memo is public. If from == to and from != 0 then invalid memo data

Public Functions

```

void set_message (const fc::ecc::private_key &priv, const fc::ecc::public_key &pub, const
                  string &msg, uint64_t custom_nonce = 0)

```

Note custom_nonce is for debugging only; do not set to a nonzero value in production

Public Members

uint64_t nonce = 0

64 bit nonce format: [8 bits | 56 bits] [entropy | timestamp] Timestamp is number of microseconds since the epoch Entropy is a byte taken from the hash of a new private key

This format is not mandated or verified; it is chosen to ensure uniqueness of key-IV pairs only. This should be unique with high probability as long as the generating host has a high-resolution clock OR a strong source of entropy for generating private keys.

vector<char> message

This field contains the AES encrypted packed *memo_message*

```

struct memo_message

```

#include <memo.hpp> defines a message and checksum to enable validation of successful decryption

When encrypting/decrypting a checksum is required to determine whether or not decryption was successful.

```
class node_property_object
#include <node_property_object.hpp> Contains per-node database configuration.
```

Transactions are evaluated differently based on per-node state. Settings here may change based on whether the node is syncing or up-to-date. Or whether the node is a witness node. Or if we're processing a transaction in a witness-signed block vs. a fresh transaction from the p2p network. Or configuration-specified tradeoffs of performance/hardfork resilience vs. paranoia.

```
class op_evaluator
#include <evaluator.hpp> Subclassed by graphene::chain::op_evaluator_impl< T >
template <typename T>
class op_evaluator_impl
#include <evaluator.hpp> Inherits from graphene::chain::op_evaluator
```

struct op_wrapper
#include <operations.hpp> necessary to support nested operations inside the *proposal_create_operation*

```
class operation_history_object
#include <operation_history_object.hpp> tracks the history of all logical operations on blockchain state
```

All operations and virtual operations result in the creation of an *operation_history_object* that is maintained on disk as a stack. Each real or virtual operation is assigned a unique ID / sequence number that it can be referenced by.

Note by default these objects are not tracked, the account_history_plugin must be loaded before these objects to be maintained.

Note this object is READ ONLY it can never be modified

Inherits from graphene::db::abstract_object< operation_history_object >

Public Members

```
uint32_t block_num = 0
the block that caused this operation
```

```
uint16_t trx_in_block = 0
the transaction in the block
```

```
uint16_t op_in_trx = 0
the operation within the transaction
```

```
uint16_t virtual_op = 0
any virtual operations implied by operation in block
```

```
struct operation_validator
Used to validate operations in a polymorphic manner.
```

```
class override_transfer_evaluator
#include <transfer_evaluator.hpp> Inherits from graphene::chain::evaluator< override_transfer_evaluator >
```

```
class override_transfer_operation
#include <transfer.hpp> Allows the issuer of an asset to transfer an asset from any account to any account if they have override_authority.
```

Pre amount.asset_id->issuer == issuer

Pre issuer != from because this is pointless, use a normal transfer operation

Inherits from `graphene::chain::base_operation`

Public Members

`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.

struct price

`#include <asset.hpp>` The price struct stores asset prices in the Graphene system.

A price is defined as a ratio between two assets, and represents a possible exchange rate between those two assets. prices are generally not stored in any simplified form, i.e. a price of (1000 CORE)/(20 USD) is perfectly normal.

The assets within a price are labeled base and quote. Throughout the Graphene code base, the convention used is that the base asset is the asset being sold, and the quote asset is the asset being purchased, where the price is represented as base/quote, so in the example price above the seller is looking to sell CORE asset and get USD in return.

Public Static Functions

`price call_price (const asset &debt, const asset &collateral, uint16_t collateral_ratio)`

The black swan price is defined as debt/collateral, we want to perform a margin call before debt == collateral. Given a debt/collateral ratio of 1 USD / CORE and a maintenance collateral requirement of 2x we can define the call price to be 2 USD / CORE.

This method divides the collateral by the maintenance collateral ratio to derive a call price for the given black swan ratio.

There exists some cases where the debt and collateral values are so small that dividing by the collateral ratio will result in a 0 price or really poor rounding errors. No matter what the collateral part of the price ratio can never go to 0 and the debt can never go more than GRAPHENE_MAX_SHARE_SUPPLY

CR * DEBT/COLLAT or DEBT/(COLLAT/CR)

`static price unit_price (asset_id_type a = asset_id_type ())`

The unit price for an asset type A is defined to be a price such that for any asset m, m*A=m.

class price_feed

`#include <asset.hpp>` defines market parameters for margin positions

Unnamed Group

`price settlement_price`

Required maintenance collateral is defined as a fixed point number with a maximum value of 10.000 and a minimum value of 1.000. (denominated in GRAPHENE_COLLATERAL_RATIO_DENOM)

A black swan event occurs when value_of_collateral equals value_of_debt, to avoid a black swan a margin call is executed when value_of_debt * required_maintenance_collateral equals value_of_collateral using rate.

Default requirement is \$1.75 of collateral per \$1 of debt

BlackSwan > SQR > MCR -> SP Forced settlements will evaluate using this price, defined as BITAS-SET / COLLATERAL

`price core_exchange_rate`

Price at which automatically exchanging this asset for CORE from fee pool occurs (used for paying fees)

`uint16_t maintenance_collateral_ratio = GRAPHENE_DEFAULT_MAINTENANCE_COLLATERAL_RATIO`

Fixed point between 1.000 and 10.000, implied fixed point denominator is GRAPHENE_COLLATERAL_RATIO_DENOM

`uint16_t maximum_short_squeeze_ratio = GRAPHENE_DEFAULT_MAX_SHORT_SQUEEZE_RATIO`

Fixed point between 1.000 and 10.000, implied fixed point denominator is GRAPHENE_COLLATERAL_RATIO_DENOM

`price max_short_squeeze_price() const`

When updating a call order the following condition must be maintained:

`debt * maintenance_price() < collateral` `debt * settlement_price < debt * maintenance debt * maintenance_price() < debt * max_short_squeeze_price()` price maintenance_price()const; When selling collateral to pay off debt, the least amount of debt to receive should be `min_usd = max_short_squeeze_price() * collateral`

This is provided to ensure that a black swan cannot be triggered due to poor liquidity alone, it must be confirmed by having the `max_short_squeeze_price()` move below the black swan price.

`struct processed_transaction`

`#include <transaction.hpp>` captures the result of evaluating the operations contained in the transaction

When processing a transaction some operations generate new object IDs and these IDs cannot be known until the transaction is actually included into a block. When a block is produced these new ids are captured and included with every transaction. The index in operation_results should correspond to the same index in operations.

If an operation did not create any new object IDs then 0 should be returned.

Inherits from `graphene::chain::signed_transaction`

`class proposal_create_evaluator`

`#include <proposal_evaluator.hpp>` Inherits from `graphene::chain::evaluator< proposal_create_evaluator >`

`struct proposal_create_operation`

`#include <proposal.hpp>` 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.

Inherits from `graphene::chain::base_operation`

Public Static Functions

```
proposal_create_operation committee_proposal (const chain_parameters &param,
```

```
fc::time_point_sec head_block_time)
```

Constructs a `proposal_create_operation` suitable for committee proposals, with expiration time and review period set appropriately. No proposed_ops are added. When used to create a proposal to change chain parameters, this method expects to receive the currently effective parameters, not the proposed parameters. (The proposed parameters will go in proposed_ops, and proposed_ops is untouched by this function.)

```
class proposal_delete_evaluator
```

```
#include <proposal_evaluator.hpp> Inherits from graphene::chain::evaluator< proposal_delete_evaluator >
```

```
struct proposal_delete_operation
```

```
#include <proposal.hpp> 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.

Inherits from `graphene::chain::base_operation`

```
class proposal_object
```

```
#include <proposal_object.hpp> tracks the approval of a partially approved transaction
```

Inherits from `graphene::db::abstract_object< proposal_object >`

```
class proposal_update_evaluator
```

```
#include <proposal_evaluator.hpp> Inherits from graphene::chain::evaluator< proposal_update_evaluator >
```

```
struct proposal_update_operation
```

```
#include <proposal.hpp> The proposal_update_operation updates an existing transaction proposal
```

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.

Inherits from `graphene::chain::base_operation`

```
struct pts_address
#include <pts_address.hpp> Implements address stringification and validation from PTS
```

Public Functions

pts_address()

constructs empty / null address

pts_address(const std::string &base58str)

converts to binary, validates checksum

pts_address(const fc::ecc::public_key &pub, bool compressed = true, uint8_t version = 56)

converts to binary

bool is_valid() const

Checks the address to verify it has a valid checksum

operator std::string() const

converts to base58 + checksum

Public Members

fc::array<char, 25> addr

binary representation of address

```
struct refund_worker_type
```

#include <worker_object.hpp> A worker who returns all of his pay to the reserve.

This worker type pays everything he receives back to the network's reserve funds pool.

Public Members

share_type total_burned

Record of how much this worker has burned in his lifetime.

```
class required_approval_index
```

#include <proposal_object.hpp> tracks all of the proposal objects that require approval of an individual account.

This is a secondary index on the proposal_index

Note the set of required approvals is constant

Inherits from secondary_index

```
struct sign_state
```

Public Functions

bool signed_by(const public_key_type &k)

returns true if we have a signature for this key or can produce a signature for this key, else returns false.

```
bool check_authority (const authority *au, uint32_t depth = 0)
    Checks to see if we have signatures of the active authorites of the accounts specified in authority or
    the keys specified.
```

```
struct signed_block
    #include <block.hpp> Inherits from graphene::chain::signed_block_header
```

Subclassed by *graphene::wallet::signed_block_with_info*

```
struct signed_block_header
    #include <block.hpp> Inherits from graphene::chain::block_header
```

Subclassed by *graphene::chain::signed_block*

```
struct signed_transaction
    #include <transaction.hpp> adds a signature to a transaction
```

Inherits from *graphene::chain::transaction*

Subclassed by *graphene::chain::processed_transaction*

Public Functions

```
const signature_type &sign (const private_key_type &key, const chain_id_type &chain_id)
    signs and appends to signatures
```

```
signature_type sign (const private_key_type &key, const chain_id_type &chain_id) const
    returns signature but does not append
```

```
set<public_key_type> get_required_signatures (const chain_id_type &chain_id, const
flat_set<public_key_type> &available_keys, const std::function<const
authority *> account_id_type
> &get_active, const std::function<const authority *account_id_type> &get_owner, uint32_t
max_recursion = GRAPHENE_MAX_SIG_CHECK_DEPTH const The purpose of this method is
    to identify some subset of available_keys that will produce sufficient signatures for a transaction. The
    result is not always a minimal set of signatures, but any non-minimal result will still pass validation.
```

```
set<public_key_type> minimize_required_signatures (const chain_id_type &chain_id,
const flat_set<public_key_type>
&available_keys, const
std::function<const authority *> account_id_type
> &get_active, const std::function<const authority *account_id_type> &get_owner, uint32_t
max_recursion = GRAPHENE_MAX_SIG_CHECK_DEPTH const This is a slower replacement
    for get_required_signatures() which returns a minimal set in all cases, including some cases where
    get_required_signatures() returns a non-minimal set.
```

```
void clear ()
```

Removes all operations and signatures.

```
class special_authority_object
```

```
#include <special_authority_object.hpp> special_authority_object only exists to help with a specific
    indexing problem. We want to be able to iterate over all accounts that contain a special authority. However,
    accounts which have a special_authority are very rare. So rather than indexing account_object by
    the special_authority fields (requiring additional bookkeeping for every account), we instead maintain a
```

special_authority_object pointing to each account which has special_authority (requiring additional book-keeping only for every account which has special_authority).

This class is an implementation detail.

Inherits from `graphene::db::abstract_object< special_authority_object >`

struct stealth_confirmation

`#include <confidential.hpp>` When sending a stealth transfer we assume users are unable to scan the full blockchain; therefore, payments require confirmation data to be passed out of band. We assume this out-of-band channel is not secure and therefore the contents of the confirmation must be encrypted.

Public Functions

operator string() const

Packs *this then encodes as base58 encoded string.

stealth_confirmation(const std::string &base58)

Unpacks from a base58 string

struct transaction

`#include <transaction.hpp>` groups operations that should be applied atomically

Subclassed by *graphene::chain::signed_transaction*

Public Functions

digest_type **digest() const**

Calculate the digest for a transaction.

digest_type **sig_digest(const chain_id_type &chain_id) const**

Calculate the digest used for signature validation.

template <typename Visitor>

`vector<typename Visitor::result_type> visit(Visitor &&visitor)`

visit all operations

Public Members

uint16_t ref_block_num = 0

Least significant 16 bits from the reference block number. If relative_expiration is zero, this field must be zero as well.

uint32_t ref_block_prefix = 0

The first non-block-number 32-bits of the reference block ID. Recall that block IDs have 32 bits of block number followed by the actual block hash, so this field should be set using the second 32 bits in the *block_id_type*

fc::time_point_sec expiration

This field specifies the absolute expiration for this transaction.

class transaction_evaluation_state

`#include <transaction_evaluation_state.hpp>` Place holder for state tracked while processing a transaction. This class provides helper methods that are common to many different operations and also tracks which keys have signed the transaction

```
class transaction_object
#include <transaction_object.hpp> The purpose of this object is to enable the detection of duplicate transactions. When a transaction is included in a block a transaction_object is added. At the end of block processing all transaction_objects that have expired can be removed from the index.

Inherits from graphene::db::abstract_object< transaction_object >

class transfer_evaluator
#include <transfer_evaluator.hpp> Inherits from graphene::chain::evaluator< transfer_evaluator >

class transfer_from_blind_evaluator
#include <confidential_evaluator.hpp> Inherits from graphene::chain::evaluator< transfer_from_blind_evaluator >
```

Public Functions

void pay_fee()

Routes the fee to where it needs to go. The default implementation routes the fee to the *account_statistics_object* of the *fee_paying_account*.

Before *pay_fee()* is called, the fee is computed by *prepare_fee()* and has been moved out of the *fee_paying_account* and (if paid in a non-CORE asset) converted by the asset's fee pool.

Therefore, when *pay_fee()* is called, the fee only exists in *this->core_fee_paid*. So *pay_fee()* need only increment the receiving balance.

The default implementation simply calls *account_statistics_object->pay_fee()* to increment pending_fees or pending_vested_fees.

struct transfer_from_blind_operation

#include <confidential.hpp> Converts blinded/stealth balance to a public account balance.

Inherits from *graphene::chain::base_operation*

struct fee_parameters_type

#include <confidential.hpp>

Public Members

uint64_t fee = 5*GRAPHENE_BLOCKCHAIN_PRECISION

the cost to register the cheapest non-free account

struct transfer_operation

#include <transfer.hpp> Transfers an amount of one asset from one account to another.

Fees are paid by the “from” account

Pre amount.amount > 0

Pre fee.amount >= 0

Pre from != to

Post from account's balance will be reduced by fee and amount

Post to account's balance will be increased by amount

Return n/a

Inherits from `graphene::chain::base_operation`

Public Members

`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.

```
class transfer_to_blind_evaluator
#include <confidential_evaluator.hpp> Inherits from graphene::chain::evaluator< transfer_to_blind_evaluator >
```

Public Functions

`void pay_fee()`
Routes the fee to where it needs to go. The default implementation routes the fee to the `account_statistics_object` of the `fee_paying_account`.

Before `pay_fee()` is called, the fee is computed by `prepare_fee()` and has been moved out of the `fee_paying_account` and (if paid in a non-CORE asset) converted by the asset’s fee pool.

Therefore, when `pay_fee()` is called, the fee only exists in `this->core_fee_paid`. So `pay_fee()` need only increment the receiving balance.

The default implementation simply calls `account_statistics_object->pay_fee()` to increment `pending_fees` or `pending_vested_fees`.

```
class transfer_to_blind_operation
#include <confidential.hpp> Converts public account balance to a blinded or stealth balance.

Inherits from graphene::chain::base_operation

struct fee_parameters_type
#include <confidential.hpp>
```

Public Members

`uint64_t fee = 5*GRAPHENE_BLOCKCHAIN_PRECISION`
the cost to register the cheapest non-free account

```
class vesting_balance_create_evaluator
#include <vesting_balance_evaluator.hpp> Inherits from graphene::chain::evaluator< vesting_balance_create_evaluator >

struct vesting_balance_create_operation
#include <vesting.hpp> 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.

Return ID of newly created *vesting_balance_object*

Inherits from *graphene::chain::base_operation*

Public Members

account_id_type creator

Who provides funds initially.

account_id_type owner

Who is able to withdraw the balance.

class vesting_balance_object

#include <vesting_balance_object.hpp> Vesting balance object is a balance that is locked by the blockchain for a period of time.

Inherits from *graphene::db::abstract_object< vesting_balance_object >*

Subclassed by *graphene::wallet::vesting_balance_object_with_info*

Public Functions

void **deposit** (**const** fc::time_point_sec &*now*, **const** asset &*amount*)

Deposit amount into vesting balance, requiring it to vest before withdrawal.

void **deposit_vested** (**const** fc::time_point_sec &*now*, **const** asset &*amount*)

Deposit amount into vesting balance, making the new funds vest immediately.

void **withdraw** (**const** fc::time_point_sec &*now*, **const** asset &*amount*)

Used to remove a vesting balance from the VBO. As well as the balance field, coin_seconds_earned and coin_seconds_earned_last_update fields are updated.

The money doesn't "go" anywhere; the caller is responsible for crediting it to the proper account.

asset **get_allowed_withdraw** (**const** time_point_sec &*now*) **const**

Get amount of allowed withdrawal.

Public Members

account_id_type owner

Account which owns and may withdraw from this vesting balance.

asset **balance**

Total amount remaining in this vesting balance Includes the unvested funds, and the vested funds which have not yet been withdrawn

vesting_policy policy

The vesting policy stores details on when funds vest, and controls when they may be withdrawn.

```
class vesting_balance_withdraw_evaluator
#include <vesting_balance_evaluator.hpp> Inherits from graphene::chain::evaluator< vesting_balance_withdraw_evaluator >
```

struct vesting_balance_withdraw_operation

#include <vesting.hpp> Withdraw from a vesting balance.

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

Return Nothing

Inherits from *graphene::chain::base_operation*

Public Members**account_id_type owner**

Must be vesting_balance.owner.

struct vesting_balance_worker_type

#include <worker_object.hpp> A worker who sends his pay to a vesting balance.

This worker type takes all of his pay and places it into a vesting balance

Public Members**vesting_balance_id_type balance**

The balance this worker pays into.

struct vote_counter

#include <vote_count.hpp> Keep track of vote totals in internal authority object. See #533.

Public Functions**void finish (authority &out_auth)**

Write into out_auth, but only if we have at least one member.

struct vote_id_type

#include <vote.hpp> An ID for some votable object.

This class stores an ID for a votable object. The ID is comprised of two fields: a type, and an instance. The type field stores which kind of object is being voted on, and the instance stores which specific object of that type is being referenced by this ID.

A value of *vote_id_type* is implicitly convertible to an unsigned 32-bit integer containing only the instance. It may also be implicitly assigned from a uint32_t, which will update the instance. It may not, however, be implicitly constructed from a uint32_t, as in this case, the type would be unknown.

On the wire, a *vote_id_type* is represented as a 32-bit integer with the type in the lower 8 bits and the instance in the upper 24 bits. This means that types may never exceed 8 bits, and instances may never exceed 24 bits.

In JSON, a `vote_id_type` is represented as a string “type:instance”, i.e. “1:5” would be type 1 and instance 5.

Note In the Graphene protocol, `vote_id_type` instances are unique across types; that is to say, if an object of type 1 has instance 4, an object of type 0 may not also have instance 4. In other words, the type is not a namespace for instances; it is only an informational field.

Public Functions

`vote_id_type()`

Default constructor. Sets type and instance to 0.

`vote_id_type(vote_type type, uint32_t instance = 0)`

Construct this `vote_id_type` with provided type and instance.

`vote_id_type(const std::string &serial)`

Construct this `vote_id_type` from a serial string in the form “type:instance”.

`void set_type(vote_type type)`

Set the type of this `vote_id_type`.

`vote_type type() const`

Get the type of this `vote_id_type`.

`void set_instance(uint32_t instance)`

Set the instance of this `vote_id_type`.

`uint32_t instance() const`

Get the instance of this `vote_id_type`.

`vote_id_type &operator=(uint32_t instance)`

Set the instance of this `vote_id_type`.

`operator uint32_t() const`

Get the instance of this `vote_id_type`.

`operator std::string() const`

Convert this `vote_id_type` to a serial string in the form “type:instance”.

Public Members

`uint32_t content`

Lower 8 bits are type; upper 24 bits are instance.

`class withdraw_permission_claim_evaluator`

```
#include <withdraw_permission_evaluator.hpp> Inherits from graphene::chain::evaluator< withdraw_permission_claim_evaluator >
```

`struct withdraw_permission_claim_operation`

```
#include <withdraw_permission.hpp> 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

Inherits from [graphene::chain::base_operation](#)

Public Members

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_permission->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.

class withdraw_permission_create_evaluator

#include <withdraw_permission_evaluator.hpp> Inherits from [graphene::chain::evaluator< withdraw_permission_create_evaluator >](#)

struct withdraw_permission_create_operation

#include <withdraw_permission.hpp> 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.

Inherits from [graphene::chain::base_operation](#)

Public Members

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.

`class withdraw_permission_delete_evaluator`

`#include <withdraw_permission_evaluator.hpp>` Inherits from `graphene::chain::evaluator< withdraw_permission_delete_evaluator >`

`struct withdraw_permission_delete_operation`

`#include <withdraw_permission.hpp>` 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

Inherits from `graphene::chain::base_operation`

Public Members

`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.

`class withdraw_permission_object`

`#include <withdraw_permission_object.hpp>` Grants another account authority to withdraw a limited amount of funds per interval.

The primary purpose of this object is to enable recurring payments on the blockchain. An account which wishes to process a recurring payment may use a `withdraw_permission_claim_operation` to reference an object of this type and withdraw up to `withdrawal_limit` from `withdraw_from_account`. Only `authorized_account` may do this. 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.

Inherits from `graphene::db::abstract_object< withdraw_permission_object >`

Public Functions

`asset available_this_period(fc::time_point_sec current_time) const`

True if the permission may still be claimed for this period; false if it has already been used.

Public Members

`account_id_type withdraw_from_account`

The account authorizing `authorized_account` to withdraw from it.

`account_id_type authorized_account`

The account authorized to make withdrawals from `withdraw_from_account`.

`asset withdrawal_limit`

The maximum amount which may be withdrawn per period. All withdrawals must be of this asset type.

`uint32_t withdrawal_period_sec = 0`

The duration of a withdrawal period in seconds.

`time_point_sec period_start_time`

The beginning of the next withdrawal period.

`time_point_sec expiration`

The time at which this withdraw permission expires.

`share_type claimed_this_period`

tracks the total amount

`class withdraw_permission_update_evaluator`

`#include <withdraw_permission_evaluator.hpp>` Inherits from `graphene::chain::evaluator< withdraw_permission_update_evaluator >`

`struct withdraw_permission_update_operation`

`#include <withdraw_permission.hpp>` 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

Inherits from `graphene::chain::base_operation`

Public Members

`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.

```
class witness_create_evaluator
```

```
#include <witness_evaluator.hpp> Inherits from graphene::chain::evaluator< witness_create_evaluator >
```

```
struct witness_create_operation
```

```
#include <witness.hpp> 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.

Inherits from *graphene::chain::base_operation*

Public Members

```
account_id_type witness_account
```

The account which owns the witness. This account pays the fee for this operation.

```
class witness_object
```

```
#include <witness_object.hpp> Inherits from graphene::db::abstract_object< witness_object >
```

```
class witness_schedule_object
```

```
#include <witness_schedule_object.hpp> Inherits from graphene::db::abstract_object< witness_schedule_object >
```

```
class witness_update_evaluator
```

```
#include <witness_evaluator.hpp> Inherits from graphene::chain::evaluator< witness_update_evaluator >
```

```
struct witness_update_operation
```

```
#include <witness.hpp> Update a witness object's URL and block signing key.
```

Inherits from *graphene::chain::base_operation*

Public Members

```
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.

```
class worker_create_evaluator
```

```
#include <worker_evaluator.hpp> Inherits from graphene::chain::evaluator< worker_create_evaluator >
```

```
struct worker_create_operation
```

```
#include <worker.hpp> Create a new worker object.
```

Inherits from *graphene::chain::base_operation*

Public Members

`worker_initializer initializer`

This should be set to the initializer appropriate for the type of worker to be created.

`class worker_object`

`#include <worker_object.hpp>` Worker object contains the details of a blockchain worker. See The Blockchain Worker System for details.

Inherits from `graphene::db::abstract_object< worker_object >`

Public Members

`account_id_type worker_account`

ID of the account which owns this worker.

`time_point_sec work_begin_date`

Time at which this worker begins receiving pay, if elected.

`time_point_sec work_end_date`

Time at which this worker will cease to receive pay. Worker will be deleted at this time.

`share_type daily_pay`

Amount in CORE this worker will be paid each day.

`worker_type worker`

ID of this worker's pay balance.

`string name`

Human-readable name for the worker.

`string url`

URL to a web page representing this worker.

`vote_id_type vote_for`

Voting ID which represents approval of this worker.

`vote_id_type vote_against`

Voting ID which represents disapproval of this worker.

`struct worker_pay_visitor`

A visitor for `worker_type` which calls `pay_worker` on the worker within.

`namespace detail`

Functions

`template <typename T, int... Is>`

`void for_each(T &&t, const account_object &a, seq<Is...>)`

`template <typename Lambda>`

`void with_skip_flags(database &db, uint32_t skip_flags, Lambda callback)`

Set the `skip_flags` to the given value, call `callback`, then reset `skip_flags` to their previous value after `callback` is done.

`template <typename Lambda>`

```

void without_pending_transactions (database &db, std::vector<processed_transaction>
                                &&pending_transactions, Lambda callback)
    Empty pending_transactions, call callback, then reset pending_transactions after callback is done.

Pending transactions which no longer validate will be culled.

bool _is_authorized_asset (const database &d, const account_object &acct, const asset_object &asset_obj)
template <int... Is>
template<>
struct gen_seq<0, Is...>
    #include <database.hpp> Inherits from graphene::chain::detail::seq< Is... >

struct pending_transactions_restorer
    #include <db_with.hpp> Class used to help the without_pending_transactions implementation.

TODO: Change the name of this class to better reflect the fact that it restores popped transactions as
well as pending transactions.

struct skip_flags_restorer
    #include <db_with.hpp> Class used to help the with_skip_flags implementation. It must be defined
in this header because it must be available to the with_skip_flags implementation, which is a template
and therefore must also be defined in this header.

```

Graphene::Wallet

```
namespace graphene::wallet
```

Typedefs

```
typedef uint16_t transaction_handle_type
```

```
typedef multi_index_container<blind_receipt, indexed_by<ordered_unique<tag<by_commitment>, const_mem_fun<blind_re
```

```
typedef multi_index_container<key_label, indexed_by<ordered_unique<tag<by_label>, member<key_label, string, &key_la
```

Functions

```
template <typename T>
T from_which_variant (int which, const variant &v)
```

```
template <typename T>
static_variant_map create_static_variant_map ()
```

```
object *create_object (const variant &v)
```

This class takes a variant and turns it into an object of the given type, with the new operator.

```
struct blind_balance
    #include <wallet.hpp>
```

Public Members

```
public_key_type from
    the account this balance came from
```

```
public_key_type to
    the account this balance is logically associated with

public_key_type one_time_key
    used to derive the authority key and blinding factor

struct blind_confirmation
#include <wallet.hpp> Contains the confirmation receipt the sender must give the receiver and the meta data about the receipt that helps the sender identify which receipt is for the receiver and which is for the change address.

struct signed_block_with_info
#include <wallet.hpp> Inherits from graphene::chain::signed\_block

class utility
#include <wallet.hpp>
```

Public Static Functions

```
vector<brain_key_info> derive_owner_keys_from_brain_key (string brain_key, int number_of_desired_keys = 1)
```

Derive any number of *possible* owner keys from a given brain key.

NOTE: These keys may or may not match with the owner keys of any account. This function is merely intended to assist with account or key recovery.

See [suggest_brain_key\(\)](#)

Return A list of keys that are deterministically derived from the brainkey

Parameters

- *brain_key*: Brain key
- *number_of_desired_keys*: Number of desired keys

```
struct vesting_balance_object_with_info
```

#include <wallet.hpp> Inherits from [graphene::chain::vesting_balance_object](#)

Public Members

```
asset allowed_withdraw
```

How much is allowed to be withdrawn.

```
fc::time_point_sec allowed_withdraw_time
```

The time at which allowed_withdrawal was calculated.

```
class wallet_api
```

#include <wallet.hpp> This wallet assumes it is connected to the database server with a high-bandwidth, low-latency connection and performs minimal caching. This API could be provided locally to be used by a web interface.

Unnamed Group

```
bool 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

```
public_key_type create_blind_account (string label, string brain_key)
    Generates a new blind account for the given brain key and assigns it the given label.

vector<asset> 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

map<string, public_key_type> get_blind_accounts () const
Return all blind accounts

map<string, public_key_type> get_my_blind_accounts () const
Return all blind accounts for which this wallet has the private key

public_key_type get_public_key (string label) const
Return the public key associated with the given label
```

Public Functions

```
variant_object 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

uint64_t get_account_count () const
    Returns the number of accounts registered on the blockchain
    Return the number of registered accounts

vector<account_object> 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

map<string, account_id_type> 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)



vector<asset> 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's balances
Parameters

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



vector<asset_object> 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)

`vector<operation_detail> 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)

`vector<operation_detail> get_relative_account_history (string name, uint32_t stop, int limit, uint32_t start) const`

Returns the relative operations on the named account from start number.

Return a list of `operation_history_objects`

Parameters

- `name`: the name or id of the account
- `stop`: Sequence number of earliest operation.
- `limit`: the number of entries to return
- `start`: the sequence number where to start looping back through the history

`global_property_object 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

Return the global properties

`dynamic_global_property_object 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

Return the dynamic global properties

`account_object 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

`asset_object get_asset (string asset_name_or_id) const`

Returns information about the given asset.

Return the information about the asset stored in the block chain

Parameters

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

`asset_bitasset_data_object 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

`account_id_type 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

`asset_id_type 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 of an asset to look up

`variant 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

`string get_wallet_filename() const`

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

`string 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.

`bool 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

`bool 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

`void lock()`

Locks the wallet immediately.

`void 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()`

```
void set_password(string password)
```

Sets a new password on the wallet.

The wallet must be either ‘new’ or ‘unlocked’ to execute this command.

```
map<public_key_type, string> 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

```
string 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 multi-line string suitable for displaying on a terminal

```
string 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

```
bool 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

```
void 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.

```
void 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

```
brain_key_info 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).

Return a suggested brain_key

```
vector<brain_key_info> derive_owner_keys_from_brain_key (string brain_key, int number_of_desired_keys = 1)
const
```

Derive any number of *possible* owner keys from a given brain key.

NOTE: These keys may or may not match with the owner keys of any account. This function is merely intended to assist with account or key recovery.

See [suggest_brain_key\(\)](#)

Return A list of keys that are deterministically derived from the brainkey

Parameters

- *brain_key*: Brain key
- *numberOfDesiredKeys*: Number of desired keys

```
bool is_public_key_registered (string public_key) const
```

Determine whether a textual representation of a public key (in Base-58 format) is *currently* linked to any *registered* (i.e. non-stealth) account on the blockchain

Return Whether a public key is known

Parameters

- *public_key*: Public key

```
string serialize_transaction (signed_transaction tx) const
```

Converts a signed_transaction in JSON form to its binary representation.

TODO: I don't see a broadcast_transaction() function, do we need one?

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

```
bool 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

```
vector<signed_transaction> 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.

```
string 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

```
signed_transaction 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

```
signed_transaction 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

```
signed_transaction create_account_with_brain_key (string brain_key, string ac-  
count_name, string regis-  
trar_account, string refer-  
rer_account, bool broadcast =  
false)
```

Creates a new account and registers it on the blockchain.

See [suggest_brain_key\(\)](#)

See [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

```
signed_transaction 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

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

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

```
transaction_id_type get_transaction_id (const signed_transaction &trx) const
```

This method is used to convert a JSON transaction to its transactin ID.

```
vector<blind_receipt> blind_history (string key_or_account)
```

Return all blind receipts to/form a particular account

```
blind_receipt 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

- **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
- **confirmation_receipt:** - a base58 encoded stealth confirmation

```
blind_confirmation transfer_to_blind (string from_account_id_or_name, string as-  
set_symbol, vector<pair<string, string>> to_amounts,  
                                                                                                broadcast = false)
```

Parameters

- **to_amounts:** map from key or label to amount

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

```
blind_confirmation transfer_from_blind (string from_blind_account_key_or_label, string  
                                                                                                        to_account_id_or_name, string amount, string as-  
set_symbol, bool broadcast = false)
```

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

```
blind_confirmation 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

```
signed_transaction sell_asset (string seller_account, string amount_to_sell, string sym-  
bol_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 BTS with USD, you should sell USD for BTS.

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 block chain.

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

`signed_transaction sell (string seller_account, string base, string quote, double rate, double amount, bool broadcast)`

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.

`signed_transaction buy (string buyer_account, string base, string quote, double rate, double amount, bool broadcast)`

Place a limit order attempting to buy one asset with another.

This API call abstracts away some of the details of the `buy_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 `buy_asset` should be used instead.

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.

- The: signed transaction selling the funds.

`signed_transaction 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

`signed_transaction 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

`signed_transaction 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

`signed_transaction 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

signed_transaction **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

signed_transaction **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 on 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

signed_transaction **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

signed_transaction **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

```
signed_transaction 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

```
signed_transaction 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

```
signed_transaction 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

```
signed_transaction 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'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`.

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 settle on
- `broadcast`: true to broadcast the transaction on the network

```
signed_transaction 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 `whitelistAuthorities` and blacklisted by none of S's `blacklistAuthorities`. 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

```
signed_transaction 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

`map<string, witness_id_type> 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 witnesss, 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_witnesses()` call.

Return a list of witnesss 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 witnesss to return (max: 1000)

`map<string, committee_member_id_type> 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)

witness_object **get_witness** (string *owner_account*)

Returns information about the given witness.

Return the information about the witness stored in the block chain

Parameters

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

committee_member_object **get_committee_member** (string *owner_account*)

Returns information about the given committee_member.

Return the information about the committee_member stored in the block chain

Parameters

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

signed_transaction **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

```
signed_transaction update_witness (string witness_name, string url, string block_signing_key,  
                                bool broadcast = false)
```

Update a witness object owned by the given account.

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.

```
signed_transaction 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.

```
signed_transaction 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.

```
vector<vesting_balance_object_with_info> 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.

```
signed_transaction 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

```
signed_transaction 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

signed_transaction **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

signed_transaction **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

```
signed_transaction set_desired_witness_and_committee_member_count (string ac-
count_to_modify,
uint16_t
de-
sired_number_of_witnesses,
uint16_t
de-
sired_number_of_committee_mem-
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

```
signed_transaction 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

```
operation 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")

```
signed_transaction 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

```
signed_transaction 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

```
signed_transaction 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

```
blind_confirmation blind_transfer_help (string from_key_or_label, string to_key_or_label,  

                                         string amount, string symbol, bool broadcast =  

                                         false, bool to_temp = false)
```

Used to transfer from one set of blinded balances to another

```
struct wallet_data  
#include <wallet.hpp>
```

Public Functions

```
vector<object_id_type> my_account_ids () const  
Return IDs of all accounts in my_accounts
```

```
bool update_account (const account_object &acct)
    Add acct to my_accounts, or update it if it is already in my_accounts
    Return true if the account was newly inserted; false if it was only updated
```

Public Members

```
chain_id_type chain_id
    Chain ID this wallet is used with

vector<char> cipher_keys
    encrypted keys

map<account_id_type, set<public_key_type>> extra_keys
    map an account to a set of extra keys that have been imported for that account

namespace detail
```

Functions

```
template <class T>
optional<T> maybe_id (const string &name_or_id)

string address_to_shorthash (const address &addr)

fc::ecc::private_key derive_private_key (const std::string &prefix_string, int sequence_number)

string normalize_brain_key (string s)

namespace impl
```

Functions

```
std::string clean_name (const std::string &name)
```

5.1.3 Frequently Asked Questions

What is the standard Bitshares address structure and format?

address = ‘BTS’+base58(ripemd(sha512(compressed_pub))) (checksum obviated) But addresses are not used directly, instead you have an account (that can be controlled by one or more address, pubkey or another account). <https://bitshares.org/technology/dynamic-account-permissions/>

What public key system is used? If elliptic curve, then what is the curve?

Same as Bitcoin, secp256k1.

Is there a specification for Bitshares scripting language? (assuming there is one)

No scripting

Is the scripting language turing complete?

No scripting

What transaction types are natively supported?

Transactions are composed of operations (about ~40 different types). Example of operations are:

- transfer_operation
- limit_order_create_operation
- asset_issue_operation

Full list <https://github.com/cryptonomex/graphene/blob/master/libraries/chain/include/graphene/chain/protocol/operations.hpp>

How is accounting addressed in Bitshares? Is it a Nxt style accounting model or like Bitcoin's UTXO

Each account has a finite set of balances, one for each asset type.

What is the average size in Bytes of a Bitshares transaction?

Average wire size of operations is ~30 bytes. Average mem size of operations is ~100 bytes. https://github.com/cryptonomex/graphene/blob/master/programs/size_checker/main.cpp

How are transactions validated?

Each operation has a defined evaluator that checks for preconditions (do_evaluate) and modify the state (do_apply). (After signature verification)

```
class transfer_evaluator : public evaluator<transfer_evaluator>
{
public:
    typedef transfer_operation operation_type;

    void_result do_evaluate( const transfer_operation& o );
    void_result do_apply( const transfer_operation& o );
}
```

Are there any special affordances made for privacy?

... such as using CoinJoin or a ZK-SNARK based privacy scheme like Zerocash? If mixing is integrated at the protocol level are you using the standards set forth by the BNMCKF Mixcoin proposal

Confidential values (same as blockstream elements using the same secp256k1-zkp lib) + stealth addresses. https://github.com/ElementsProject/elementsproject.github.io/blob/master/confidential_values.md No mixing, No CoinJoin.

What data structures are used in the blockchain?

:: Blocks => transactions => operations => objects.

The blockchain state is contained in an object database that is affected by the operations. Example objects::

```
account_object  
asset_object  
account_balance_object  
...
```

```
class account_balance_object : public abstract_object<account_balance_object>  
{  
public:  
    static const uint8_t space_id = implementation_ids;  
    static const uint8_t type_id = impl_account_balance_object_type;  
  
    account_id_type owner;  
    asset_id_type asset_type;  
    share_type balance;  
  
    asset get_balance() const { return asset(balance, asset_type); }  
    void adjust_balance(const asset& delta);  
};
```

How can I transfer a single account to a cli wallet

If you have a need to run just the cli wallet, you will likely want to transfer an existing account from the Web wallet or your light wallet into the cli. This guide assumes that you already have your web wallet working and properly connected to a full node. Something like this:

```
./cli-wallet -s wss://bitshares.openledger.info/ws
```

You should get a prompt

```
new >>>
```

Now set a password for the wallet

```
new >>> set_password my_password  
locked >>> unlock my_password  
unlocked >>>
```

Now you need to go to your bitshares 2.0 webwallet or lite client and select the account you wish to bring across. Click on permissions followed by the key icon within Active Permissions. This will reveal your private key. Copy it to the clipboard.



development/faq/account-active-permissions.png

Now let's import the account into this wallet

```
new >>> import_key <accountname> THISISTHEKEYTHATYOUPIED
```

What is the format of the block header?

```
struct block_header
{
    digest_type           digest () const;
    block_id_type         previous;
    uint32_t               block_num() const { return num_from_id(previous) + _1; }
    fc::time_point_sec    timestamp;
    witness_id_type       witness;
    checksum_type          transaction_merkle_root;
    extensions_type        extensions;

    static uint32_t num_from_id(const block_id_type& id);
};
```

What is the maximum bitshares block size?

Configurable by chain parameters.

```
struct chain_parameters
{
    /* using a smart ref breaks the circular dependency created between operations
     * and the fee schedule */
    smart_ref<fee_schedule> current_fees;                                ///< current schedule
    // of fees
    uint8_t                  block_interval = GRAPHENE_DEFAULT_BLOCK_INTERVAL; // interval in seconds between blocks
    uint32_t                  maintenance_interval = GRAPHENE_DEFAULT_MAINTENANCE_INTERVAL; // interval in seconds between blockchain maintenance events
    uint8_t                  maintenance_skip_slots = GRAPHENE_DEFAULT_MAINTENANCE_SKIP_SLOTS;
    uint32_t                  committee_proposal_review_period = GRAPHENE_DEFAULT_COMMITTEE_PROPOSAL_REVIEW_PERIOD_SEC; // minimum time in seconds that a proposed transaction requiring committee authority may not be signed, prior to expiration
    uint32_t                  maximum_transaction_size = GRAPHENE_DEFAULT_MAX_TRANSACTION_SIZE; // maximum allowable size in bytes for a transaction
    uint32_t                  maximum_block_size = GRAPHENE_DEFAULT_MAX_BLOCK_SIZE; // maximum allowable size in bytes for a block
    uint32_t                  maximum_time_until_expiration = GRAPHENE_DEFAULT_MAX_TIME_UNTIL_EXPIRATION; // maximum lifetime in seconds for transactions to be valid, before expiring
    uint32_t                  maximum_proposal_lifetime = GRAPHENE_DEFAULT_MAX_PROPOSAL_LIFETIME_SEC; // maximum lifetime in seconds for proposed transactions to be kept, before expiring
    uint8_t                   maximum_asset_whitelistAuthorities = GRAPHENE_DEFAULT_MAX_ASSET_WHITELIST_AUTHORITIES; // maximum number of accounts which an asset may list as authorities for its whitelist OR blacklist
    uint8_t                   maximum_asset_feed_publishers = GRAPHENE_DEFAULT_MAX_ASSET_FEED_PUBLISHERS; // the maximum number of feed publishers for a given asset
    uint16_t                  maximum_witness_count = GRAPHENE_DEFAULT_MAX_WITNESSES; // maximum number of active witnesses
```

```

uint16_t maximum_committee_count = GRAPHENE_DEFAULT_MAX_
→COMMITTEE; ///< maximum number of active committee members
uint16_t maximum_authority_membership = GRAPHENE_DEFAULT_MAX_
→AUTHORITY_MEMBERSHIP; ///< largest number of keys/accounts an authority can have
uint16_t reserve_percent_of_fee = GRAPHENE_DEFAULT_
→BURN_PERCENT_OF_FEE; ///< the percentage of the network's allocation of a fee that is taken out of circulation
uint16_t network_percent_of_fee = GRAPHENE_DEFAULT_
→NETWORK_PERCENT_OF_FEE; ///< percent of transaction fees paid to network
uint16_t lifetime_referrer_percent_of_fee = GRAPHENE_DEFAULT_
→LIFETIME_REFERRER_PERCENT_OF_FEE; ///< percent of transaction fees paid to network
uint32_t cashback_vesting_period_seconds = GRAPHENE_DEFAULT_
→CASHBACK_VESTING_PERIOD_SEC; ///< time after cashback rewards are accrued before they become liquid
share_type cashback_vesting_threshold = GRAPHENE_DEFAULT_
→CASHBACK_VESTING_THRESHOLD; ///< the maximum cashback that can be received without vesting
bool count_non_member_votes = true; ///< set to false to restrict voting privileges to member accounts
bool allow_non_member_whitelists = false; ///< true if non-member accounts may set whitelists and blacklists; false otherwise
share_type witness_pay_per_block = GRAPHENE_DEFAULT_
→WITNESS_PAY_PER_BLOCK; ///< CORE to be allocated to witnesses (per block)
uint32_t witness_pay_vesting_seconds = GRAPHENE_DEFAULT_
→WITNESS_PAY_VESTING_SECONDS; ///< vesting_seconds parameter for witness VBO's
share_type worker_budget_per_day = GRAPHENE_DEFAULT_
→WORKER_BUDGET_PER_DAY; ///< CORE to be allocated to workers (per day)
uint16_t max_predicate_opcode = GRAPHENE_DEFAULT_MAX_
→ASSERT_OPCODE; ///< predicate_opcode must be less than this number
share_type fee_liquidation_threshold = GRAPHENE_DEFAULT_FEE_
→LIQUIDATION_THRESHOLD; ///< value in CORE at which accumulated fees in blockchain-issued market assets should be liquidated
uint16_t accounts_per_fee_scale = GRAPHENE_DEFAULT_
→ACCOUNTS_PER_FEE_SCALE; ///< number of accounts between fee scalings
uint8_t account_fee_scale_bitshifts = GRAPHENE_DEFAULT_
→ACCOUNT_FEE_SCALE_BITSHIFTS; ///< number of times to left bitshift account registration fee at each scaling
uint8_t max_authority_depth = GRAPHENE_MAX_SIG_
→CHECK_DEPTH;
extensions_type extensions;

/** defined in fee_schedule.cpp */
void validate()const;
};

```

Are there any sharding mechanics currently deployed?

No

How are SPV clients handled?

No SPV clients at the moment, each full node can expose a public websocket/http api.

Does the protocol provide mechanisms for overlay protocols to interact such as OR_RETURN?

Yes, using a custom_operation.

```
struct custom_operation : public base_operation
{
    struct fee_parameters_type {
        uint64_t fee = GRAPHENE_BLOCKCHAIN_PRECISION;
        uint32_t price_per_kbyte = 10;
    };

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

    account_id_type fee_payer() const { return payer; }
    void            validate() const;
    share_type      calculate_fee(const fee_parameters_type& k) const;
};
```

How is time addressed in the blockchain? Is NTP used or some other protocol?

NTP

How do new clients bootstrap into the network?

Trusted seed nodes. Knowledge of initial witness keys.

What is the average block time?

Current 3 seconds, configurable by chain parameters.

Is this done via a gossip protocol or through a federate relay?

Each node immediately broadcast the data it receives to its peers after validating it <https://github.com/cryptonomex/graphene/blob/master/libraries/p2p/design.md>

5.2 Testnets

This page is dedicated to *testnets*. We distinguish between our **public testnet** that has been deployed and is fully functional for anyone to use and is shared among developers, and **private testnets** that every developer could easily deploy at home to benefit from extra low latency and additional super powers over the network.

You can find all the relevant information about the public network on a dedicated page:

[Open Public Testnet](#) (technical details)

Tutorials about how to deploy a public, or a private network are presented in separated tutorials:

5.2.1 Public Testnet Howto

We took the opportunity to write the process of customizing and deploying a graphene blockchain as well as the faucet, web wallet and feed scripts into a tutorial for everyone to use.

Installation/Configuration of Witness

Fork CNX's code base

```
git clone https://github.com/cryptonomex/graphene
mv graphene/ graphene-testnet
cd graphene-testnet/
git branch testnet
git remote set-url origin https://github.com/BitSharesEurope/graphene-testnet
git push origin testnet
```

Installation according to/installation/Build

Configuration

Blockchain Parameters

The blockchain parameters can be modified in the `libraries/chain/include/graphene/chain/config.hpp` file:

```
vim libraries/chain/include/graphene/chain/config.hpp
```

Default Seed Node List

We can add a default list of seed nodes that the witness is supposed to try to connect to in `libraries/app/application.cpp` and will add the IP/Address and port of the machine we are going to setup later already:

`testnet.bitshares.eu:11010`

The full changeset can be seen in the corresponding [git commit](#)

Initial Compilation

```
cmake .
make
```

We first need to compile the graphene toolkit so that we can let it generate a plain genesis file in the proper format.

Genesis Configuration

We will create a genesis file named `my-genesis.json` that contains the genesis block:

```
mkdir -p genesis
programs/witness_node/witness_node --create-genesis-json genesis/my-genesis.json
vim genesis/my-genesis.json
```

The `my-genesis.json` is the initial state of the network.

Genesis editing

If you want to customize the network's initial state, edit `my-genesis.json`. This allows you to control things such as:

- The accounts that exist at genesis, their names and public keys
- Assets and their initial distribution (including core asset)
- The initial values of chain parameters
- The account / signing keys of the `init` witnesses (or in fact any account at all).

The chain ID is a hash of the genesis state. All transaction signatures are only valid for a single chain ID. So editing the genesis file will change your chain ID, and make you unable to sync with all existing chains (unless one of them has exactly the same genesis file you do).

Writing final genesis

We now copy our gensis template file over to the graphene root directory::

```
$ cp genesis/my-genesis.json genesis.json
$ vim genesis.json
$ git add genesis.json
$ git commit -m "Added genesis.json"
```

The **initial timestamp** needs to be pasted into `genesis.json` file in the `initial_timestamp` field. Choose it relatively close to the future where you can generate the genesis block (e.g. now plus 10 minutes).

Including Genesis into the binaries

To let the binaries know about your new genesis block, we need to recompile it and provide `cmake` with the parameter to identify the genesis block properly:

```
$ make clean
$ find . -name "CMakeCache.txt" | xargs rm -f
$ find . -name "CMakeFiles" | xargs rm -Rf
$ cmake -DGRAPHENE_EGENESIS_JSON="$(pwd) /genesis.json" .
```

You can add the `GRAPHENE_EGENESIS_JSON` to the default parameters by adding:

```
set(GRAPHENE_EGENESIS_JSON "${CMAKE_CURRENT_SOURCE_DIR}/genesis.json" )
```

to the `CMakeLists.txt` file. This way, you don't need to provide this parameter all the time.

Initializing Blockchain

Initializing the genesis block

We initialize the blockchain an generate our first blocks.

The `--enable-stale-production` flag tells the `witness_node` to produce on a chain with zero blocks or very old blocks. We specify the `--enable-stale-production` parameter on the command line as we will not normally need it (although it can also be specified in the config file).

```
programs/witness_node/witness_node --genesis-json genesis.json \
--enable-stale-production \
--data-dir data/testnet
```

We will already see our chain id::

```
Started witness node on a chain with 0 blocks.
Chain ID is cf307110d029cb882d126bf0488dc4864772f68d9888d86b458d16e6c36aa74b
```

Note: If other witness produces blocks and witness participation is high enough, subsequent runs which connect to an existing witness node over the p2p network, or which get blockchain state from an existing data directory, need not have the --enable-stale-production flag.

Setting up block production

Let's create a very basic configuration file in *data/testnet/config.ini*:

```
$ mkdir -p data/testnet
$ vim data/testnet/config.ini
```

All we put into the configuration file is the ids and the keys for the witnesses so that we can start producing blocks

```
witness-id = "1.6.1"
witness-id = "1.6.2"
witness-id = "1.6.3"
witness-id = "1.6.4"
witness-id = "1.6.5"
witness-id = "1.6.6"
witness-id = "1.6.7"
witness-id = "1.6.8"
witness-id = "1.6.9"
witness-id = "1.6.10"
# For each witness, add pubkey and private key:
private-key = ["GPH6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
← "5KQwrPbwL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"]
private-key = [<pubkey>, <privkey>]
```

This authorizes the `witness_node` to produce blocks on behalf of the listed `witness-id`'s, and specifies the private key needed to sign those blocks. Normally each witness would be on a different node, but for the purposes of this testnet, we will start out with all witnesses signing blocks on a single node.

Note: The setting `rpc-endpoint = 0.0.0.0:11011` will open up the RPC-port *11011* to connect a cli-wallet or web wallet to it. With the `p2p-endpoint = 0.0.0.0:11010` being accessible from the internet, this node

can be used as seed node.

Embedding the Genesis block (optional)

Now that we have the blockchain established and the used correct genesis block, we can have it embedded into the binaries directly. For that reasons we have moved it into the root directory and called it `genesis.json` for the default compile toolchain to catch it automatically. We recompile to include the genesis block with:

```
make clean
find . -name "CMakeCache.txt" | xargs rm -f
find . -name "CMakeFiles" | xargs rm -Rf
cmake -DCMAKE_BUILD_TYPE=Release .
```

Deleting caches will reset all `cmake` variables, so if you have used instructions like `./installation/Build` which tells you to set other `cmake` variables, you will have to add those variables to the `cmake` line above.

Embedding the genesis copies the entire content of genesis block into the `witness_node` binary, and additionally copies the chain ID into the `cli_wallet` binary. Embedded genesis allows the following simplifications to the subsequent instructions:

- You need **not** specify the genesis file on the witness node command line, or in the witness node configuration file.
- You need **not** specify the chain ID on the `cli_wallet` command line when starting a new wallet.

Connecting a CLI wallet

We will now show how to connect a cli-wallet to the new blockchain and generate our first transaction on the new blockchain.

Creating a wallet

In order to create a wallet, you must specify the previously setup server. With the witness node's default access control settings, a blank username and password will suffice:

```
programs/cli_wallet/cli_wallet --wallet-file my-wallet.json -s ws://127.0.0.1:11011 -H 127.0.0.1:8090 -r 127.0.0.1:8099
```

Note: The parameter `-H` is required so that we can interface with the cli wallet via RPC-HTTP-JSON, later while `-r` will open a port for the Ruby-based faucet.

Before continuing, we should set a password. This password is used to encrypt the private keys in the wallet. We will use the word `supersecret` in this example.:

```
>>> set_password supersecret
```

Gaining access to stake

In Graphene, balances are contained in accounts. To claim an account that exists in the Graphene genesis, use the `import_key` command::

```
>>> unlock supersecret
>>> import_key <name> "<wifkey>"
```

Funds are stored in genesis balance objects. These funds can be claimed, with no fee, using the `import_balance` command::

```
>>> import_balance <name> ["*"] true
```

Creating accounts

Creating an account requires lifetime member (LTM) status. To upgrade to LTM, use the `upgrade_account` command::

```
>>> upgrade_account faucet true
```

We can now register an account. The `register_account` command allows you to register an account using only a public key::

```
>>> register_account alpha GPH4zSJHx7D84T1j6HQ7keXWdtabBBWJxvfJw72XmEyqmgdoo1njF_
->GPH4zSJHx7D84T1j6HQ7keXWdtabBBWJxvfJw72XmEyqmgdoo1njF faucet faucet 0 true
>>> transfer faucet alpha 100000 CORE "here is the cash" true
```

We can now open a new wallet for alpha user::

```
>>> import_key alpha 5HuCDiMeESd86xrRvTbexLjkVg2BEoKrb7BAA5RLgXizkgV3shs
>>> upgrade_account alpha true
>>> create_witness alpha "http://www.alpha" true
```

The `get_private_key` command allows us to obtain the private key corresponding to the block signing key::

```
>>> get_private_key GPH6viEhYCQr8xKP3Vj8wfHh6WfZeJK7H9uhLPDYWLGRSj5kHQZM
```

Establishing a Committee

Our network, of course, needs a committee. We need to initially create new accounts and let them apply as committee member.

Creating members

```
create_account_with_brain_key com0 com0 faucet faucet true
create_account_with_brain_key com1 com1 faucet faucet true
create_account_with_brain_key com2 com2 faucet faucet true
create_account_with_brain_key com3 com3 faucet faucet true
create_account_with_brain_key com4 com4 faucet faucet true
create_account_with_brain_key com5 com5 faucet faucet true
create_account_with_brain_key com6 com6 faucet faucet true
```

Upgrading members

Since only lifetime members can be committee members, we need to fund these accounts and upgrade them accordingly:

```

transfer faucet com0 100000 CORE "some cash" true
transfer faucet com1 100000 CORE "some cash" true
transfer faucet com2 100000 CORE "some cash" true
transfer faucet com3 100000 CORE "some cash" true
transfer faucet com4 100000 CORE "some cash" true
transfer faucet com5 100000 CORE "some cash" true
transfer faucet com6 100000 CORE "some cash" true
upgrade_account com0 true
upgrade_account com1 true
upgrade_account com2 true
upgrade_account com3 true
upgrade_account com4 true
upgrade_account com5 true
upgrade_account com6 true

```

Registering as committee member

We can apply for committee with *create_committee_member*:

```

create_committee_member com0 "" true
create_committee_member com1 "" true
create_committee_member com2 "" true
create_committee_member com3 "" true
create_committee_member com4 "" true
create_committee_member com5 "" true
create_committee_member com6 "" true

```

Voting with faucet account

All we need to do know is vote for our own committee members:

```

vote_for_committee_member faucet com0 true true
vote_for_committee_member faucet com1 true true
vote_for_committee_member faucet com2 true true
vote_for_committee_member faucet com3 true true
vote_for_committee_member faucet com4 true true
vote_for_committee_member faucet com5 true true
vote_for_committee_member faucet com6 true true

```

Web Wallet

Since we need to provide a way for people to enter the network/blockchain, we need to install the web wallet into nginx.

Dependencies

We first install everything we need to compile the web wallet:

```

sudo apt-get install git nodejs-legacy npm
sudo npm install -g webpack coffee-script

```

Fetching the web wallet

Afterwards, we download the *graphene-ui* repository from Cryptonomex and install the Node dependencies.

```
git clone https://github.com/cryptonomex/graphene-ui
cd graphene-ui/
for I in dl web; do cd $I; npm install; cd ..; done
```

Configuration

What we need now is the *chain_id* of the chain we have running. We can get it by executing::

```
$ curl --data '{"jsonrpc": "2.0", "method": "get_chain_properties", "params": [], "id
˓→": 1}' http://127.0.0.1:11011/rpc && echo
```

The chain id is used to let the web wallet know to which network it connects and how to deal with it. For this we modify the file *dl/src/chain/config.coffee* and add our blockchain::

```
Test:
  core_asset: "TEST"
  address_prefix: "TEST"
  chain_id: "<chain-id>"
```

Furthermore, we need to tell our web wallet to which witness node to connect to. This can be done in the file *dl/src/stores/SettingsStore.js*.

```
connection: "ws://<host>/ws",
faucet_address: "https://<host>",

# also edit the "default" settings
```

Compilation

We compile the web wallet with

```
cd web
npm run build
```

which will generate the static files in the *dist/* folder.

Setting up the Faucet

In order to allow people that do not have funds yet to enter the system, we need to setup a faucet. Here, we will also use *mina* as our **deployment tool** for a productive installation.

Installation of Dependencies

we first install every other dependency that is needed and not yet installed

```
sudo apt-get install mysql-server libmysqlclient-dev
# put a master password for mysql
```

We also need to install a decently recent version of Ruby:

```
cd
git clone git://github.com/sstephenson/rbenv.git .rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
exec $SHELL

git clone git://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bashrc
exec $SHELL

git clone https://github.com/sstephenson/rbenv-gem-rehash.git ~/.rbenv/plugins/rbenv-
↪gem-rehash

sudo rbenv install 2.2.3
sudo rbenv global 2.2.3
sudo gem install bundler
```

Get the Source

One implementation of a faucet can be downloaded from Cryptonomex:

```
git clone https://github.com/cryptonomex/faucet
cd faucet
sudo bundle # ignore warnings
```

Configuration

We need to configure

- the faucet itself,
- rails API secrets,
- database access,
- and the deployment tool, mina.

Faucet

All required settings are in *config/faucet.yml* which has an example file:

```
cp config/faucet-example.yml config/faucet.yml
vim config/faucet.yml
```

Rails API

Rails needs to know a secret for their internals, we can get a new one with *rake secret*. Put it into the corresponding lines in the config file.

```
cp config/secrets-example.yml config/secrets.yml  
rake secret  
vim config/secrets.yml
```

Database Access

The default configuration for the database access expects an empty root password for MySQL. Hence, if you have set a different password, you need to provide it here.

```
vim config/database.yml
```

We generate our databases with:

```
rake db:create; rake db:migrate; rake db:seed  
RAILS_ENV=production bundle exec rake db:create db:schema:load
```

Database Settings

We also need to add an entry to the database so that page loads and referrals work properly:

1. Go to /www/current
2. execute: rails db, a mysql console will open
3. Execute: `insert into widgets set allowed_domains='testnet.bitshares.eu';` (replace the domain with your domain)

Mina deployment

Mina is used to deploy the faucet properly and move all the required files over to the production machine (may be the same machine).

```
cp config/deploy-example.yml config/deploy.yml  
vim config/deploy.yml
```

Make sure to add your ssh-pub key to your authorized file so that mina can deploy to the corresponding machine.

Now, all we need to do is create the public directly that is served and copy/link the configuration file to the deployment's shared files:

```
sudo mkdir /www  
sudo chown -R gph:gph /www  
  
mina setup  
ln -s $HOME/faucet/config/faucet.yml /www/shared/config/  
ln -s $HOME/faucet/config/secrets.yml /www/shared/config/  
ln -s $HOME/faucet/config/database.yml /www/shared/config/
```

We deploy mina and the wallet with:

```
ln -s $HOME/graphene-ui/web/dist $HOME/faucet/public/wallet  
mina deploy  
mina wallet
```

Update wallet

If you have modified something in the wallet, you need to recompile the Javascript/HTML files and re-deploy the wallet with:

```
# re-compile
cd ~/graphene-ui/web
git pull # fetch changes from origin
npm run build

# deploy
cd ~/faucet
mina wallet
```

Nginx Webserver

We here use Nginx to deal with Ruby and provide access to the wallet files and the faucet. We use *passenger* to do so which unfortunately requires to compile nginx “manually”. Passenger fortunately provides a script that handles all the installation.

Setup

```
sudo apt-get install curl libcurl4-gnutls-dev
cd ~/faucet
gem install passenger
sudo passenger-install-nginx-module
```

After using the recommended settings and waiting for the script to complete, we integrate nginx into our distribution via

```
sudo ln -s /opt/nginx/ /usr/local/nginx
sudo ln -s /opt/nginx/conf/ /etc/nginx
```

Configuration

Nginx is configured using the freshly installed *nginx.conf* file:

```
vi /opt/nginx/conf/nginx.conf
```

We have set it to be:

```
user gph;
worker_processes 4;

events {
    worker_connections 1024;
}

http {
    passenger_root /home/gph/.rbenv/versions/2.2.3/lib/ruby/gems/2.2.0/gems/passenger-
    ↪5.0.23;
    passenger_ruby /home/gph/.rbenv/versions/2.2.3/bin/ruby;
```

```
passenger_max_request_queue_size 1000;

include      mime.types;
default_type application/octet-stream;

access_log  /www/logs/access.log;
error_log   /www/logs/error.log;
log_not_found off;

sendfile     on;
#tcp_nopush  on;

keepalive_timeout 65;

gzip on;

upstream websockets {
    ## Put the witness node's websocket rpc port here:
    server localhost:11011;
}

server {
    listen 80;
    server_name localhost;

    location ~ ^/[\w\d\.-]+\.(js|css|dat|png|json)$ {
        root /www/current/public/;
        try_files $uri /wallet$uri =404;
    }

    location ~ /ws/? {
        access_log off;
        proxy_pass http://websockets;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }

    location / {
        passenger_enabled on;
        root /www/current/public/;
    }
}
}
```

Note: The parameters *passenger_root* and *passenger_ruby* may be different in your setup. Please compare with the default *nginx.conf* file to identify the proper directories.

We create an *upstream* called *websockets* which is used to proxy the queries to `http://host/ws` directly to the websocket server. This allows to have a websocket address be available from the same port as the web wallet.

Running nginx as service

We use a nginx serice script and can install it according to:

```
sudo wget https://raw.github.com/JasonGiedymin/nginx-init-ubuntu/master/nginx -O /etc/
  ↵init.d/nginx
sudo chmod +x /etc/init.d/nginx
sudo update-rc.d -f nginx defaults
```

After that, nginx can be launched with

```
sudo service nginx start
```

Installation of Python Library

Requirements

First, we need to install *pip3* which will deal with the packages dependencies:

```
sudo apt-get install python3-pip
```

Installation

Afterwards, we can fetch the python libraries from github and install it: .. code-block:: sh

```
git clone https://github.com/xeroc/python-graphenelib cd python-graphenelib/
pip3 install --user -r requirements.txt python3 setup.py install --user
```

Usage

The usage of this library is well documented on

- <https://readthedocs.org/projects/python-graphenelib/>

Create MPAs/UIAs

We now create some Market Pegged assets and construct the price feed.

- Create Market Pegged Assets: https://github.com/BitSharesEurope/testnet-pytonscripts/blob/master/create_mpa.py
- Fund *init** witnessses: <https://github.com/BitSharesEurope/testnet-pytonscripts/blob/master/fund-inits.py>
- Price feed scripts:
 - <https://github.com/BitSharesEurope/testnet-pytonscripts/blob/master/feed.last.py>
 - <https://github.com/BitSharesEurope/testnet-pytonscripts/blob/master/feed.parity.py>
 - <https://github.com/BitSharesEurope/testnet-pytonscripts/blob/master/feed.random.py>

We can use *cron* to run the price feed script on a regular basis:

```
*/15 * * * * /usr/bin/python3 /home/gph/testnet-pythonscripts/feed.last.py
*/15 * * * * /usr/bin/python3 /home/gph/testnet-pythonscripts/feed.random.py
0      * * * * /usr/bin/python3 /home/gph/testnet-pythonscripts/feed.parity.py
```

5.2.2 Private Testnet Howto

Some developers may want to deploy their own graphene blockchain locally for governance, and speed reasons. This page shows how this can be done.

Prerequisites

We assume that you have both `witness_node` and `cli_wallet` already compiled (or downloaded from [the official repository](#)).

Folder structure

Create a new folder (we will refer to it as `[Testnet-Home]`) in any location you like and copy `witness_node` and `cli_wallet` there. The `[Testnet-Home]` folder will contain all files and folders related to the testnet.

Open a *Command Prompt* window and switch the current directory to `[Testnet-Home]`.

The genesis file

The genesis file defines the initial state of the network.

Default genesis

The graphene code base has a default genesis block integrated that has all witnesses, committee members and funds and a single account called `nathan` available from a single private key:

```
5KQwrPbwL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkyFD3
```

See below how to use this key, or go ahead to learn about how to define your own genesis file

Customization of the genesis file

We create a new genesis json file named `my-genesis.json` by running this command:

```
$ witness_node --create-genesis-json my-genesis.json
```

The `my-genesis.json` file will be created in the `[Testnet-Home]` folder. Once this task is done, the witness node will terminate on its own.

If you want to customize the network's initial state, edit the newly created `my-genesis.json` file. This allows you to control things such as:

- The accounts that exist at genesis, their names and public keys
- Assets and their initial distribution (including core asset)
- The initial values of chain parameters (including fees)
- The account signing keys of the init witnesses (or in fact any account at all).

Get the blockchain id

The blockchain id is a hash of the genesis state. All transaction signatures are only valid for a single blockchain id. So editing the genesis file will change your blockchain id, and make you unable to sync with all existing chains (unless one of them has exactly the same genesis file you do).

Run this command:

```
witness_node --data-dir data    # to use the default genesis, or
witness_node --data-dir data --genesis-json my-genesis.json    # your own genesis block
```

and when a message like this shows up:

```
3501235ms th_a main.cpp:165 main] Started witness node on a chain with 0 blocks.
3501235ms th_a main.cpp:166 main] Chain ID is
→8b7bd36a146a03d0e5d0a971e286098f41230b209d96f92465cd62bd64294824
```

... it means the initialization is complete. Use `ctrl-c` to close the witness node.

As a result, you should get two items:

- A directory named `data` has been created with a file named `config.ini` located in it.
- Your blockchain id is now known - it's displayed in the message above.

Note: Note that your blockchain id will be different than the one used in the above example. Copy this id somewhere as you will be needing it later on.

Witness configuration

Open the [Testnet-Home]/data/config.ini file in your favorite text editor, and set the following settings, uncommenting them if necessary:

```
rpc-endpoint = 127.0.0.1:11011
genesis-json = my-genesis.json
enable-stale-production = true
```

Also, locate this entry in the config.ini file:

```
# ID of witness controlled by this node (e.g. "1.6.5", quotes are required, may
→specify multiple times)
```

... and add the following entries:

```
witness-id = "1.6.1"
witness-id = "1.6.2"
witness-id = "1.6.3"
witness-id = "1.6.4"
witness-id = "1.6.5"
witness-id = "1.6.6"
witness-id = "1.6.7"
witness-id = "1.6.8"
witness-id = "1.6.9"
witness-id = "1.6.10"
witness-id = "1.6.11"
```

The above list authorizes the witness node to produce blocks on behalf of the listed witness ids. Normally each witness would be on a different node, but for the purpose of this private testnet, we will start out with all witnesses signing blocks on a single node. The private keys for all those witness ids (needed to sign blocks) are already supplied in the config.ini file:

```
# Tuple of [PublicKey, WIF private key] (may specify multiple times)
private-key = ["BTS6MRyA...T5GDW5CV", "5KQwrPb...tP79zkvFD3"]
```

Start block production

This is the crucial moment - you are about to produce the very first blocks of your private blockchain. Just run the witness node with this command:

```
witness_node --data-dir data
```

and your block production should start at this stage. You should see this big message:

```
*****
*                               *
* ----- NEW CHAIN ----- *
* - Welcome to Graphene! - *
* -----                 *
*                               *
*****
```

and subsequently further messages indicating the successfull creation of blocks:

```
2322793ms th_a main.cpp:176      main      ] Started witness node on a chain with 0
blocks.
2322794ms th_a main.cpp:177      main      ] Chain ID is
8b7bd36a146a03d0e5d0a971e286098f41230b209d96f92465cd62bd64294824
2324613ms th_a witness.cpp:185   block_production_loo ] Generated block #1 with
timestamp 2016-01-21T22:38:40 at time 2016-01-21T22:38:40
2325604ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2342604ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2343609ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2344604ms th_a witness.cpp:185   block_production_loo ] Generated block #2 with
timestamp 2016-01-21T22:39:00 at time 2016-01-21T22:39:00
2345605ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2349616ms th_a witness.cpp:185   block_production_loo ] Generated block #3 with
timestamp 2016-01-21T22:39:05 at time 2016-01-21T22:39:05
2350602ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2353612ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2354605ms th_a witness.cpp:185   block_production_loo ] Generated block #4 with
timestamp 2016-01-21T22:39:10 at time 2016-01-21T22:39:10
2355609ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
2356609ms th_a witness.cpp:194   block_production_loo ] Not producing block because
slot has not yet arrived
```

CLI Usage

We are now ready to connect the CLI to your testnet witness node. Keep your witness node running and in another *Command Prompt* window run this command:

```
cli_wallet --wallet-file=my-wallet.json --chain-id=8b7bd36a146a03d0e5d0a971e286098f41230b209d96f92465cd62bd64294824 --server-rpc-endpoint=ws://127.0.0.1:11011
```

Note: Make sure to replace the above blockchain id `8b7bd36a...4294824` with your own blockchain id. The blockchain id passed to the CLI needs to match the id generated and used by the witness node.

If you get the `set_password` prompt, it means your CLI has successfully connected to the testnet witness node.

Create a new wallet

First you need to create a new password for your wallet. This password is used to encrypt all the private keys in the wallet. For this tutorial we will use the password `supersecret` but obviously you are free to come up with your own combination of letters and numbers. Use this command to create the password:

```
>>> set_password supersecret
```

Now you can unlock the newly created wallet:

```
unlock supersecret
```

Gain access to the genesis stake

In Graphene, balances are contained in accounts. To import an account into your wallet, all you need to know is its name and its private key. We will now import into the wallet an account called `nathan` using the `import_key` command:

```
import_key nathan 5KQwrPbwL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtp79zkvFD3
```

Note: Note that `nathan` happens to be the account name defined in the genesis file. If you had edited your `my-genesis.json` file just after it was created, you could have put a different name there. Also, note that `5KQwrPbwL...P79zkvFD3` is the private key defined in the `config.ini` file.

Now we have the private key imported into the wallet but still no funds associated with it. Funds are stored in genesis balance objects. These funds can be claimed, with no fee, using the `import_balance` command:

```
import_balance nathan ["5KQwrPbwL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtp79zkvFD3"] true
```

As a result, we have one account (named `nathan`) imported into the wallet and this account is well funded with BTS as we have claimed the funds stored in the genesis file. You can view this account by using this command:

```
get_account nathan
```

and its balance by using this command:

```
list_account_balances nathan
```

Create another account

We will now create another account (named alpha) so that we can transfer funds back and forth between nathan and alpha.

Creating a new account is always done by using an existing account - we need it because someone (i.e. the registrar) has to fund the registration fee. Also, there is the requirement for the registrar account to have a lifetime member (LTM) status. Therefore we need to upgrade the account nathan to LTM, before we can proceed with creating other accounts. To upgrade to LTM, use the `upgrade_account` command:

```
upgrade_account nathan true
```

Note: Due to a known [caching issue](#), you need to restart the CLI at this stage as otherwise it will not be aware of nathan having been upgraded. Stop the CLI by pressing `ctrl-c` and start it again by using exactly the same command as before, i.e.

```
cli_wallet --wallet-file=my-wallet.json --chain-id=8b7bd36a146a03d0e5d0a971e286098f41230b209d96f92465cd62bd64294824 --server-rpc-endpoint=ws://127.0.0.1:11011
```

Verify that nathan has now a LTM status:

```
get_account nathan
```

In the response, next to `membership_expiration_date` you should see `1969-12-31T23:59:59`. If you get `1970-01-01T00:00:00` something is wrong and nathan has not been successfully upgraded.

We can now register an account by using nathan as registrar. But first we need to get hold of the public key for the new account. We do it by using the `suggest_brain_key` command:

```
suggest_brain_key
```

And the response should be something similar to this:

```
suggest_brain_key
{
  "brain_priv_key": "MYAL SOEVER UNSHARP PHYSIC JOURNEY SHEUGH BEDLAM WLOKA FOOLERY",
  "wif_priv_key": "5JDh3XmHK8CDaQSxQZHh5PUV3zwzG68uVcrTfmq9yQ9idNisYnE",
  "pub_key": "BTS78CuY47Vds2nfw2t88ckjTaggPkwl6tLhcmsg4ReVx1WPr1zRL5"
}
```

The `create_account_with_brain_key` command allows you to register an account using brain key and will automatically import the corresponding private key.

```
create_account_with_brain_key "Brain ... key" <accountname> nathan nathan true
```

Transfer funds between accounts

As a final step, we will transfer some money from nathan to alpha. For that we use the `transfer` command:

```
transfer nathan alpha 2000000000 BTS "here is some cash" true
```

The text here is some cash is an arbitrary memo you can attach to a transfer. If you don't need it, just use "" instead.

And now you can verify that alpha has indeed received the money:

```
list_account_balances alpha
```


CHAPTER 6

Contribute

6.1 How to Contribute

6.1.1 Contribute to BitShares

Backend Development

The BitShares backend is open for anyone to contribute and improve. One repository is hosted at [github](#).

Pull requests to add new features and innovations are welcome, but a hard forking deployments require shareholder approval.

Frontend Development

A corresponding [frontend](#) implementation is licensed under an Open Source license as well and contributions are welcome as well.

6.1.2 Contribute to Graphene

Backend Development

The *Graphene* toolkit has been released under MIT license and is thus open for anyone to contribute and improve.

Due to the decentralized nature of development in combination with the powerful source repository system *git*, anyone is free to contribute to any graphene-based repository.

Since [Cryptonomex](#) is currently the main contributer, we recommend to fork off of [their](#) repository hosted at [github](#).

Frontend Development

A corresponding reference frontend implementation is licensed under an Open Source license as well and contributions are welcome. The reference implementation developed by Cryptonomex is available at [github](#).

6.1.3 Contributing to the Documentation

This documentation page is hosted at [github](#) and can be cloned with

```
git clone https://github.com/BitSharesEurope/docs.bitshares.eu
```

Sources are in the `master` branch whereas deployment (generated html pages) are located in the `gh-pages` branch.

Note: The `docs.bitshares.org` repository (`org`-domain) is a deployment, only!

Pull requests are welcome!

Requirements

- doxygen (run `doxygen` in the graphene source to generate required files)
- graphviz
- sphinx (<http://sphinx-doc.org>)
- breathe (<https://github.com/michaeljones/breathe>)

Quick Install in OSX

It's easy to get started on OSX using brew and `easy_install`.

```
brew install doxygen  
brew install graphviz  
  
easy_install sphinx  
easy_install breathe
```

Building

```
make html
```

Output

The resulting html files will be written to `build/html`.

Deployment

The Makefile can automatically deploy to several domains. Currently, a deployment installs the page simultaneously at

- docs.bitshares.org
- docs.bitshares.eu

```
git remote set-url origin github:BitSharesEurope/docs.bitshares.eu
git remote add org    github:BitSharesEurope/docs.bitshares.org
make deploy
```

Updating doxygen (optional)

in *doxygen/* there are files that have been generated by bitshares/graphene C++ code using doxygen. If you want to update them, run *doxygen* in bitshares and move over the *xml* folder in *doxygen/*.

A

asset_bitasset_data_object::max_force_settlement_volume
(C++ function), 272
asset_object::amount_from_string (C++ function), 275
asset_object::amount_to_string (C++ function), 275

G

graphene::app (C++ type), 241
graphene::app::abstract_plugin (C++ class), 242
graphene::app::abstract_plugin::plugin_initialize (C++
function), 242
graphene::app::abstract_plugin::plugin_set_app (C++
function), 242
graphene::app::abstract_plugin::plugin_set_program_options
(C++ function), 242
graphene::app::abstract_plugin::plugin_shutdown (C++
function), 242
graphene::app::abstract_plugin::plugin_startup (C++
function), 242
graphene::app::application (C++ class), 242
graphene::app::application::syncing_finished (C++ mem-
ber), 243
graphene::app::block_api (C++ class), 243
graphene::app::crypto_api::blind (C++ function), 230
graphene::app::crypto_api::blind_sign (C++ function),
230
graphene::app::crypto_api::blind_sum (C++ function),
230
graphene::app::crypto_api::range_get_info (C++ func-
tion), 230
graphene::app::crypto_api::range_proof_sign (C++ func-
tion), 230
graphene::app::crypto_api::unblind_signature (C++ func-
tion), 230
graphene::app::crypto_api::verify_range (C++ function),
230
graphene::app::crypto_api::verify_range_proof_rewind
(C++ function), 230
graphene::app::crypto_api::verify_sum (C++ function),

230
graphene::app::database_api (C++ class), 243
graphene::app::database_api::cancel_all_subscriptions
(C++ function), 214, 243
graphene::app::database_api::get_24_volume (C++ func-
tion), 220, 246
graphene::app::database_api::get_account_balances
(C++ function), 217, 245
graphene::app::database_api::get_account_by_name
(C++ function), 216
graphene::app::database_api::get_account_count (C++
function), 217, 245
graphene::app::database_api::get_account_references
(C++ function), 216, 244
graphene::app::database_api::get_accounts (C++ func-
tion), 216, 244
graphene::app::database_api::get_assets (C++ function),
218, 245
graphene::app::database_api::get_balance_objects (C++
function), 217, 245
graphene::app::database_api::get_blinded_balances (C++
function), 224, 249
graphene::app::database_api::get_block (C++ function),
215, 243
graphene::app::database_api::get_block_header (C++
function), 215, 243
graphene::app::database_api::get_block_header_batch
(C++ function), 243
graphene::app::database_api::get_call_orders (C++ func-
tion), 219, 246
graphene::app::database_api::get_chain_id (C++ func-
tion), 215, 244
graphene::app::database_api::get_chain_properties (C++
function), 215, 244
graphene::app::database_api::get_committee_member_by_account
(C++ function), 222, 248
graphene::app::database_api::get_committee_members
(C++ function), 222, 247
graphene::app::database_api::get_config (C++ function),
215, 244

graphene::app::database_api::get_dynamic_global_properties (C++ function),
(C++ function), 215, 244

graphene::app::database_api::get_full_accounts (C++ function), 216, 244

graphene::app::database_api::get_global_properties (C++ function), 215, 244

graphene::app::database_api::get_key_references (C++ function), 216

graphene::app::database_api::get_limit_orders (C++ function), 219, 245

graphene::app::database_api::get_margin_positions (C++ function), 219, 246

graphene::app::database_api::get_named_account_balances (C++ function), 217, 245

graphene::app::database_api::get_objects (C++ function), 214, 243

graphene::app::database_api::get_order_book (C++ function), 218, 246

graphene::app::database_api::get_potential_address_signatures (C++ function), 223

graphene::app::database_api::get_potential_signatures (C++ function), 223, 248

graphene::app::database_api::get_proposed_transactions (C++ function), 224, 249

graphene::app::database_api::get_recent_transaction_by_id (C++ function), 215, 243

graphene::app::database_api::get_required_fees (C++ function), 223, 249

graphene::app::database_api::get_required_signatures (C++ function), 223, 248

graphene::app::database_api::get_settle_orders (C++ function), 219, 246

graphene::app::database_api::get_ticker (C++ function), 220, 246

graphene::app::database_api::get_trade_history (C++ function), 220, 247

graphene::app::database_api::get_transaction (C++ function), 215, 243

graphene::app::database_api::get_transaction_hex (C++ function), 223, 248

graphene::app::database_api::get_vested_balances (C++ function), 217

graphene::app::database_api::get_vesting_balances (C++ function), 218

graphene::app::database_api::get_witness_by_account (C++ function), 221, 247

graphene::app::database_api::get_witness_count (C++ function), 221, 247

graphene::app::database_api::get_witnesses (C++ function), 221, 247

graphene::app::database_api::get_workers_by_account (C++ function), 222, 248

graphene::app::database_api::is_public_key_registered (C++ function), 244

graphene::app::database_api::list_assets (C++ function), 218, 245

graphene::app::database_api::lookup_account_names (C++ function), 216, 244

graphene::app::database_api::lookup_accounts (C++ function), 217, 245

graphene::app::database_api::lookup_asset_symbols (C++ function), 218, 245

graphene::app::database_api::lookup_committee_member_accounts (C++ function), 222, 248

graphene::app::database_api::lookup_vote_ids (C++ function), 223, 248

graphene::app::database_api::lookup_witness_accounts (C++ function), 221, 247

graphene::app::database_api::set_block_applied_callback (C++ function), 214

graphene::app::database_api::set_pending_transaction_callback (C++ function), 214

graphene::app::database_api::set_subscribe_callback (C++ function), 214

graphene::app::database_api::subscribe_to_market (C++ function), 220, 246

graphene::app::database_api::unsubscribe_from_market (C++ function), 220, 246

graphene::app::database_api::validate_transaction (C++ function), 223, 249

graphene::app::database_api::verify_account_authority (C++ function), 223, 248

graphene::app::database_api::verify_authority (C++ function), 223, 248

graphene::app::database_api::impl (C++ class), 249

graphene::app::database_api::impl::get_key_references (C++ function), 249

graphene::app::database_api::impl::get_limit_orders (C++ function), 249

graphene::app::database_api::impl::get_proposed_transactions (C++ function), 249

graphene::app::database_api::impl::on_applied_block (C++ function), 249

graphene::app::database_api::impl::on_objects_new (C++ function), 249

graphene::app::dejsonify (C++ function), 241

graphene::app::detail (C++ type), 253

graphene::app::detail::application_impl (C++ class), 253

graphene::app::detail::application_impl::connection_count_changed (C++ function), 255

graphene::app::detail::application_impl::get_block_ids (C++ function), 254

graphene::app::detail::application_impl::get_block_time (C++ function), 255

graphene::app::detail::application_impl::get_blockchainSynopsis (C++ function), 254

graphene::app::detail::application_impl::get_item (C++ function), 254

graphene::app::detail::application_impl::handle_block
 (C++ function), 254

graphene::app::detail::application_impl::has_item (C++
 function), 254

graphene::app::detail::application_impl::sync_status
 (C++ function), 255

graphene::app::detail::create_example_genesis (C++
 function), 253

graphene::app::get_relevant_accounts (C++ function),
 241

graphene::app::get_required_fees_helper (C++ class),
 249

graphene::app::history_api (C++ class), 249

graphene::app::history_api::get_account_history (C++
 function), 225, 250

graphene::app::history_api::get_account_history_operations
 (C++ function), 250

graphene::app::history_api::get_fill_order_history (C++
 function), 225

graphene::app::history_api::get_market_history (C++
 function), 225

graphene::app::history_api::get_market_history_buckets
 (C++ function), 226

graphene::app::history_api::get_relative_account_history
 (C++ function), 250

graphene::app::login_api (C++ class), 251

graphene::app::login_api::asset (C++ function), 251

graphene::app::login_api::block (C++ function), 251

graphene::app::login_api::crypto (C++ function), 251

graphene::app::login_api::database (C++ function), 251

graphene::app::login_api::debug (C++ function), 251

graphene::app::login_api::enable_api (C++ function),
 251

graphene::app::login_api::history (C++ function), 251

graphene::app::login_api::login (C++ function), 251

graphene::app::login_api::network_broadcast (C++ func-
 tion), 251

graphene::app::login_api::network_node (C++ function),
 251

graphene::app::network_broadcast_api (C++ class), 251

graphene::app::network_broadcast_api::broadcast_block
 (C++ function), 226

graphene::app::network_broadcast_api::broadcast_transac-
 tion (C++ function), 226, 251

graphene::app::network_broadcast_api::broadcast_transac-
 tion (C++ function), 252

graphene::app::network_broadcast_api::broadcast_transac-
 tion (C++ function), 226, 252

graphene::app::network_broadcast_api::on_applied_block
 (C++ function), 252

graphene::app::network_node_api (C++ class), 252

graphene::app::network_node_api::add_node (C++ func-
 tion), 227, 252

graphene::app::network_node_api::get_advanced_node_parameters
 (C++ function), 227, 252

graphene::app::network_node_api::get_connected_peers
 (C++ function), 227, 252

graphene::app::network_node_api::get_info (C++ func-
 tion), 227, 252

graphene::app::network_node_api::get_potential_peers
 (C++ function), 227, 252

graphene::app::network_node_api::set_advanced_node_parameters
 (C++ function), 227, 252

graphene::app::operation_get_impacted_accounts (C++
 function), 241

graphene::app::plugin (C++ class), 252

graphene::app::plugin::plugin_initialize (C++ function),
 253

graphene::app::plugin::plugin_set_app (C++ function),
 253

graphene::app::plugin::plugin_set_program_options
 (C++ function), 253

graphene::app::plugin::plugin_shutdown (C++ function),
 253

graphene::app::plugin::plugin_startup (C++ function),
 253

graphene::app::transaction_get_impacted_accounts (C++
 function), 241, 242

graphene::chain (C++ type), 255

graphene::chain::account_balance_id_type (C++ type),
 257

graphene::chain::account_balance_index (C++ type), 255

graphene::chain::account_balance_object (C++ class),
 264

graphene::chain::account_balance_object_multi_index_type
 (C++ type), 255

graphene::chain::account_create_evaluator (C++ class),
 264

graphene::chain::account_create_operation (C++ class),
 264

graphene::chain::account_create_operation::fee_parameters_type
 (C++ class), 264

graphene::chain::account_create_operation::fee_parameters_type::basic_fee
 (C++ member), 264

graphene::chain::account_create_operation::fee_parameters_type::premium
 (C++ member), 264

graphene::chain::account_create_operation::referrer
 (C++ member), 264

graphene::chain::account_create_operation::referrer_percent
 (C++ member), 264

graphene::chain::account_create_operation::registrar
 (C++ member), 264

graphene::chain::account_id_type (C++ type), 257

graphene::chain::account_index (C++ type), 255

graphene::chain::account_member_index (C++ class),
 264

graphene::chain::account_member_index::account_to_account_membership
 (C++ member), 265

graphene::chain::account_member_index::account_to_address
graphene::chain::account_member_index::active
graphene::chain::account_member_index::blacklisting_accounts
graphene::chain::account_member_index::cashback_vb
graphene::chain::account_member_index::is_annual_member
graphene::chain::account_member_index::is_basic_account
graphene::chain::account_member_index::is_lifetime_member
graphene::chain::account_member_index::is_member
graphene::chain::account_member_index::lifetime_referrer
graphene::chain::account_member_index::lifetime_referrer_fee_percentage
graphene::chain::account_member_index::membership_expiration_date
graphene::chain::account_member_index::name
graphene::chain::account_member_index::network_fee_percentage
graphene::chain::account_member_index::owner
graphene::chain::account_member_index::referrer
graphene::chain::account_member_index::referrer_rewards_percentage
graphene::chain::account_member_index::registrar
graphene::chain::account_member_index::statistics
graphene::chain::account_member_index::top_n_control_flags
graphene::chain::account_member_index::whitelisted_accounts
graphene::chain::account_member_index::whitelisting_accounts
graphene::chain::account_member_index::account_options (C++ class), 165, 259
graphene::chain::account_member_index::account_options (C++ class), 267

graphene::chain::account_options::memo_key (C++ member), 267
graphene::chain::account_options::num_committee (C++ member), 267
graphene::chain::account_options::num_witness (C++ member), 267
graphene::chain::account_options::votes (C++ member), 267
graphene::chain::account_options::voting_account (C++ member), 267
graphene::chain::account_referrer_index (C++ class), 267
graphene::chain::account_referrer_index::referred_by (C++ member), 267
graphene::chain::account_statistics_id_type (C++ type), 257
graphene::chain::account_statistics_object (C++ class), 267
graphene::chain::account_statistics_object::lifetime_fees_paid (C++ member), 268
graphene::chain::account_statistics_object::most_recent_op (C++ member), 268
graphene::chain::account_statistics_object::pay_fee (C++ function), 268
graphene::chain::account_statistics_object::pending_fees (C++ member), 268
graphene::chain::account_statistics_object::pending_vested_fees (C++ member), 268
graphene::chain::account_statistics_object::process_fees (C++ function), 268
graphene::chain::account_statistics_object::removed_ops (C++ member), 268
graphene::chain::account_statistics_object::total_core_in_orders (C++ member), 268
graphene::chain::account_statistics_object::total_ops (C++ member), 268
graphene::chain::account_transaction_history_id_type (C++ type), 257
graphene::chain::account_transaction_history_index (C++ type), 256
graphene::chain::account_transaction_history_multi_index_type (C++ type), 256
graphene::chain::account_transaction_history_object (C++ class), 268
graphene::chain::account_transaction_history_object::next (C++ member), 269
graphene::chain::account_transaction_history_object::operation_id (C++ member), 269
graphene::chain::account_transfer_operation (C++ class), 269
graphene::chain::account_update_evaluator (C++ class), 269
graphene::chain::account_update_operation (C++ class), 269

graphene::chain::account_update_operation::account
(C++ member), 269

graphene::chain::account_update_operation::active (C++
member), 269

graphene::chain::account_update_operation::new_options
(C++ member), 269

graphene::chain::account_update_operation::owner (C++
member), 269

graphene::chain::account_upgrade_evaluator (C++ class),
269

graphene::chain::account_upgrade_operation (C++
class), 269

graphene::chain::account_upgrade_operation::account_to_upg
(C++ member), 270

graphene::chain::account_upgrade_operation::fee_paramete
(C++ class), 270

graphene::chain::account_upgrade_operation::fee_paramete
(C++ member), 270

graphene::chain::account_upgrade_operation::upgrade_to_li
(C++ member), 270

graphene::chain::account_whitelist_evaluator (C++
class), 270

graphene::chain::account_whitelist_operation (C++
class), 270

graphene::chain::account_whitelist_operation::account_listi
(C++ type), 203

graphene::chain::account_whitelist_operation::account_to_li
(C++ member), 270

graphene::chain::account_whitelist_operation::authorizing_<graphene>
(C++ member), 270

graphene::chain::account_whitelist_operation::black_listed
(C++ class), 203

graphene::chain::account_whitelist_operation::fee (C++
member), 270

graphene::chain::account_whitelist_operation::new_listing
(C++ member), 271

graphene::chain::account_whitelist_operation::no_listing
(C++ class), 203

graphene::chain::account_whitelist_operation::white_and_b
(C++ class), 203

graphene::chain::account_whitelist_operation::white_listed
(C++ class), 203

graphene::chain::add_authority_accounts (C++ function),
262

graphene::chain::address (C++ class), 271

graphene::chain::address::address (C++ function), 271

graphene::chain::address::operator std::string (C++ func
tion), 271

graphene::chain::assert_evaluator (C++ class), 271

graphene::chain::assert_operation (C++ class), 271

graphene::chain::assert_operation::calculate_fee (C++
function), 271

graphene::chain::asset_bitasset_data_id_type (C++ type),
257

graphene::chain::asset_bitasset_data_index (C++ type),
255

graphene::chain::asset_bitasset_data_object (C++ class),
271

graphene::chain::asset_bitasset_data_object::current_feed
(C++ member), 272

graphene::chain::asset_bitasset_data_object::current_feed_publication_time
(C++ member), 272

graphene::chain::asset_bitasset_data_object::feeds (C++
member), 272

graphene::chain::asset_bitasset_data_object::force_settled_volume
(C++ member), 272

graphene::chain::asset_bitasset_data_object::has_settlement
(C++ function), 272

graphene::chain::asset_bitasset_data_object::is_prediction_market
(C++ member), 272

graphene::chain::asset_bitasset_data_object::options
(C++ member), 272

graphene::chain::asset_bitasset_data_object::settlement_fund
(C++ member), 272

graphene::chain::asset_bitasset_data_object::settlement_price
(C++ member), 272

graphene::chain::asset_bitasset_data_object_multi_index_type
(C++ type), 255

graphene::chain::asset_claim_fees_evaluator (C++ class),
272

graphene::chain::asset_claim_fees_operation (C++
class), 272

graphene::chain::asset_create_operation::extensions
(C++ member), 273

graphene::chain::asset_create_evaluator (C++ class), 273

graphene::chain::asset_create_operation (C++ class), 273

graphene::chain::asset_create_operation::bitasset_opts
(C++ member), 273

graphene::chain::asset_create_operation::common_options
(C++ member), 273

graphene::chain::asset_create_operation::is_prediction_market
(C++ member), 273

graphene::chain::asset_create_operation::issuer (C++
member), 273

graphene::chain::asset_create_operation::precision (C++
member), 273

graphene::chain::asset_create_operation::symbol (C++
member), 273

graphene::chain::asset_dynamic_data_id_type (C++
type), 257

graphene::chain::asset_dynamic_data_object (C++ class),
273

graphene::chain::asset_dynamic_data_object::accumulated_fees
(C++ member), 273

graphene::chain::asset_dynamic_data_object::confidential_supply
(C++ member), 273

graphene::chain::asset_dynamic_data_object::current_supply
(C++ member), 273

graphene::chain::asset_dynamic_data_object::fee_pool
(C++ member), 273

graphene::chain::asset_fund_fee_pool_evaluator (C++ class), 274

graphene::chain::asset_fund_fee_pool_operation (C++ class), 274

graphene::chain::asset_fund_fee_pool_operation::amount
(C++ member), 274

graphene::chain::asset_fund_fee_pool_operation::fee
(C++ member), 274

graphene::chain::asset_global_settle_evaluator (C++ class), 274

graphene::chain::asset_global_settle_operation (C++ class), 274

graphene::chain::asset_global_settle_operation::issuer
(C++ member), 274

graphene::chain::asset_id_type (C++ type), 257

graphene::chain::asset_index (C++ type), 256

graphene::chain::asset_issue_evaluator (C++ class), 274

graphene::chain::asset_issue_operation (C++ class), 274

graphene::chain::asset_issue_operation::issuer
(C++ member), 274

graphene::chain::asset_issue_operation::memo
(C++ member), 274

graphene::chain::asset_issuer_permission_flags
(C++ type), 206, 208, 259

graphene::chain::ASSET_ISSUER_PERMISSION_MASK
(C++ member), 264

graphene::chain::asset_object (C++ class), 274

graphene::chain::asset_object::amount (C++ function),
275

graphene::chain::asset_object::amount_to_pretty_string
(C++ function), 275

graphene::chain::asset_object::amount_to_string
(C++ function), 275

graphene::chain::asset_object::bitasset_data_id
(C++ member), 276

graphene::chain::asset_object::can_force_settle
(C++ function), 275

graphene::chain::asset_object::can_global_settle
(C++ function), 275

graphene::chain::asset_object::charges_market_fees
(C++ function), 275

graphene::chain::asset_object::dynamic_asset_data_id
(C++ member), 275

graphene::chain::asset_object::is_market_issued (C++
function), 275

graphene::chain::asset_object::is_transfer_restricted
(C++ function), 275

graphene::chain::asset_object::is_valid_symbol (C++
function), 276

graphene::chain::asset_object::issuer (C++ member), 275

graphene::chain::asset_object::precision (C++ member),
275

graphene::chain::asset_object::reserved (C++ function),
275

graphene::chain::asset_object::symbol (C++ member),
275

graphene::chain::asset_object_multi_index_type
(C++ type), 255

graphene::chain::asset_object_type (C++ class), 165, 259

graphene::chain::asset_options (C++ class), 205, 207, 276

graphene::chain::asset_options::blacklistAuthorities
(C++ member), 205, 208, 277

graphene::chain::asset_options::blacklistMarkets (C++
member), 206, 208, 277

graphene::chain::asset_options::core_exchange_rate
(C++ member), 205, 208, 276

graphene::chain::asset_options::description (C++ mem-
ber), 206, 208, 277

graphene::chain::asset_options::flags (C++ member),
205, 208, 276

graphene::chain::asset_options::issuer_permissions (C++
member), 205, 208, 276

graphene::chain::asset_options::market_fee_percent
(C++ member), 205, 207, 276

graphene::chain::asset_options::max_market_fee
(C++ member), 205, 208, 276

graphene::chain::asset_options::max_supply (C++ mem-
ber), 205, 207, 276

graphene::chain::asset_options::validate (C++ function),
205, 207, 276

graphene::chain::asset_options::whitelistAuthorities
(C++ member), 205, 208, 276

graphene::chain::asset_options::whitelistMarkets (C++
member), 205, 208, 277

graphene::chain::asset_publish_feed_operation
(C++ class), 277

graphene::chain::asset_publish_feed_operation::asset_id
(C++ member), 277

graphene::chain::asset_publish_feed_operation::fee (C++
member), 277

graphene::chain::asset_publish_feeds_evaluator
(C++ class), 277

graphene::chain::asset_reserve_evaluator (C++ class),
277

graphene::chain::asset_reserve_operation (C++ class),
277

graphene::chain::asset_settle_cancel_operation
(C++ class), 277

graphene::chain::asset_settle_cancel_operation::account
(C++ member), 278

graphene::chain::asset_settle_cancel_operation::amount
(C++ member), 278

graphene::chain::asset_settle_evaluator (C++ class), 278

graphene::chain::asset_settle_operation (C++ class), 278

graphene::chain::asset_settle_operation::account
(C++ member), 278

graphene::chain::asset_settle_operation::amount (C++ member), 278

graphene::chain::asset_settle_operation::fee_parameters_type (C++ class), 278

graphene::chain::asset_settle_operation::fee_parameters_type (C++ member), 278

graphene::chain::asset_symbol_eq_lit_predicate (C++ class), 278

graphene::chain::asset_symbol_eq_lit_predicate::validate (C++ function), 278

graphene::chain::asset_update_bitasset_evaluator (C++ class), 278

graphene::chain::asset_update_bitasset_operation (C++ class), 279

graphene::chain::asset_update_evaluator (C++ class), 279

graphene::chain::asset_update_feed_producers_evaluator (C++ class), 279

graphene::chain::asset_update_feed_producers_operation (C++ class), 279

graphene::chain::asset_update_operation (C++ class), 279

graphene::chain::asset_update_operation::new_issuer (C++ member), 280

graphene::chain::authority (C++ class), 280

graphene::chain::authority::address_auths (C++ member), 280

graphene::chain::balance_claim_evaluator (C++ class), 280

graphene::chain::balance_claim_evaluator::do_apply (C++ function), 280

graphene::chain::balance_claim_operation (C++ class), 280

graphene::chain::balance_id_type (C++ type), 257

graphene::chain::balance_object (C++ class), 280

graphene::chain::balance_object_type (C++ class), 165, 260

graphene::chain::base_object_type (C++ class), 165, 259

graphene::chain::base_operation (C++ class), 280

graphene::chain::bitasset_options (C++ class), 281

graphene::chain::bitasset_options::feed_lifetime_sec (C++ member), 281

graphene::chain::bitasset_options::force_settlement_delay_sec (C++ member), 281

graphene::chain::bitasset_options::force_settlement_offset_sec (C++ member), 281

graphene::chain::bitasset_options::maximum_force_settlement_sec (C++ member), 281

graphene::chain::bitasset_options::minimum_feeds (C++ member), 281

graphene::chain::bitasset_options::short_backing_asset (C++ member), 281

graphene::chain::bitasset_options::validate (C++ function), 281

graphene::chain::blind_input (C++ class), 281

graphene::chain::blind_input::owner (C++ member), 282

graphene::chain::blind_memo (C++ class), 282

graphene::chain::blind_memo::check (C++ member), 282

graphene::chain::blind_output (C++ class), 282

graphene::chain::blind_output::range_proof (C++ member), 282

graphene::chain::blind_transfer_evaluator (C++ class), 282

graphene::chain::blind_transfer_evaluator::pay_fee (C++ function), 282

graphene::chain::blind_transfer_operation (C++ class), 282

graphene::chain::blind_transfer_operation::fee_parameters_type (C++ class), 283

graphene::chain::blind_transfer_operation::fee_parameters_type::fee (C++ member), 283

graphene::chain::blind_transfer_operation::fee_payer (C++ function), 283

graphene::chain::blind_transfer_operation::validate (C++ function), 283

graphene::chain::blinded_balance_id_type (C++ type), 257

graphene::chain::blinded_balance_index (C++ type), 256

graphene::chain::blinded_balance_object (C++ class), 283

graphene::chain::blinded_balance_object_multi_index_type (C++ type), 256

graphene::chain::block_header (C++ class), 283

graphene::chain::block_id_predicate (C++ class), 283

graphene::chain::block_id_type (C++ type), 257

graphene::chain::block_summary_id_type (C++ type), 257

graphene::chain::block_summary_object (C++ class), 284

graphene::chain::budget_record_id_type (C++ type), 257

graphene::chain::budget_record_object (C++ class), 284

graphene::chain::burn_worker_type (C++ class), 284

graphene::chain::burn_worker_type::total_burned (C++ member), 284

graphene::chain::buyback_account_options (C++ class), 284

graphene::chain::buyback_account_options::asset_to_buy (C++ member), 284

graphene::chain::buyback_account_options::asset_to_buy_issuer (C++ member), 284

graphene::chain::buyback_account_options::markets (C++ member), 284

graphene::chain::buyback_id_type (C++ type), 257

graphene::chain::buyback_index (C++ type), 256

graphene::chain::buyback_multi_index_type (C++ type), 256

graphene::chain::buyback_object (C++ class), 284

graphene::chain::call_order_id_type (C++ type), 257

graphene::chain::call_order_index (C++ type), 256

graphene::chain::call_order_multi_index_type (C++ type), 256

graphene::chain::call_order_object (C++ class), 284

graphene::chain::call_order_object::call_price (C++ member), 285

graphene::chain::call_order_object::collateral (C++ member), 285

graphene::chain::call_order_object::debt (C++ member), 285

graphene::chain::call_order_object_type (C++ class), 165, 260

graphene::chain::call_order_update_evaluator (C++ class), 285

graphene::chain::call_order_update_operation (C++ class), 285

graphene::chain::call_order_update_operation::delta_collateral (C++ member), 285

graphene::chain::call_order_update_operation::delta_debt (C++ member), 285

graphene::chain::call_order_update_operation::fee_parameters_type (C++ class), 285

graphene::chain::call_order_update_operation::funding_account (C++ member), 285

graphene::chain::cdd_vesting_policy (C++ class), 285

graphene::chain::cdd_vesting_policy::compute_coin_seconds (C++ function), 286

graphene::chain::cdd_vesting_policy::start_claim (C++ member), 286

graphene::chain::cdd_vesting_policy::update_coin_seconds (C++ function), 286

graphene::chain::cdd_vesting_policy_initializer (C++ class), 286

graphene::chain::cdd_vesting_policy_initializer::start_claim (C++ member), 286

graphene::chain::chain_id_type (C++ type), 257

graphene::chain::chain_parameters (C++ class), 286

graphene::chain::chain_parameters::account_fee_scale_bits (C++ member), 288

graphene::chain::chain_parameters::accounts_per_fee_scale (C++ member), 288

graphene::chain::chain_parameters::allow_non_member_whitelist (C++ member), 287

graphene::chain::chain_parameters::block_interval (C++ member), 286

graphene::chain::chain_parameters::cashback_vesting_period_seconds (C++ member), 287

graphene::chain::chain_parameters::cashback_vesting_threshold (C++ member), 287

graphene::chain::chain_parameters::committee_proposal_regeneration_time (C++ member), 286

graphene::chain::chain_parameters::count_non_member_votes (C++ member), 287

graphene::chain::chain_parameters::current_fees (C++ member), 286

graphene::chain::chain_parameters::fee_liquidation_threshold (C++ member), 287

graphene::chain::chain_parameters::lifetime_referrer_percent_of_fee (C++ member), 287

graphene::chain::chain_parameters::maintenance_interval (C++ member), 286

graphene::chain::chain_parameters::maintenance_skip_slots (C++ member), 286

graphene::chain::chain_parameters::max_predicate_opcode (C++ member), 287

graphene::chain::chain_parameters::maximum_asset_feed_publishers (C++ member), 287

graphene::chain::chain_parameters::maximum_asset_whitelistAuthorities (C++ member), 287

graphene::chain::chain_parameters::maximum_authority_membership (C++ member), 287

graphene::chain::chain_parameters::maximum_block_size (C++ member), 287

graphene::chain::chain_parameters::maximum_committee_count (C++ member), 287

graphene::chain::chain_parameters::maximum_proposal_lifetime (C++ member), 287

graphene::chain::chain_parameters::maximum_time_until_expiration (C++ member), 287

graphene::chain::chain_parameters::maximum_transaction_size (C++ member), 287

graphene::chain::chain_parameters::maximum_witness_count (C++ member), 287

graphene::chain::chain_parameters::network_percent_of_fee (C++ member), 287

graphene::chain::chain_parameters::reserve_percent_of_fee (C++ member), 287

graphene::chain::chain_parameters::validate (C++ function), 286

graphene::chain::chain_parameters::witness_pay_per_block (C++ member), 287

graphene::chain::chain_parameters::witness_pay_vesting_seconds (C++ member), 287

graphene::chain::chain_parameters::worker_budget_per_day (C++ member), 287

graphene::chain::chain_property_id_type (C++ type), 257

graphene::chain::chain_property_object (C++ class), 288

graphene::chain::charge_market_fee (C++ class), 206, 208, 259

graphene::chain::checksum_type (C++ type), 257

graphene::chain::committee_fed_asset (C++ class), 206, 208, 259

graphene::chain::committee_member_create_evaluator (C++ class), 288

graphene::chain::committee_member_create_operation (C++ class), 288

graphene::chain::committee_member_create_operation::committee_member (C++ member), 288

graphene::chain::committee_member_id_type (C++ type), 257
graphene::chain::committee_member_object (C++ class), 288
graphene::chain::committee_member_object_type (C++ class), 165, 259
graphene::chain::committee_member_update_evaluator (C++ class), 288
graphene::chain::committee_member_update_global_params (C++ class), 288
graphene::chain::committee_member_update_global_params (C++ class), 288
graphene::chain::committee_member_update_operation (C++ class), 289
graphene::chain::committee_member_update_operation (C++ member), 289
graphene::chain::committee_member_update_operation::committee_member (C++ member), 289
graphene::chain::create_buyback_orders (C++ function), 261
graphene::chain::custom_evaluator (C++ class), 289
graphene::chain::custom_id_type (C++ type), 257
graphene::chain::custom_object_type (C++ class), 165, 260
graphene::chain::custom_operation (C++ class), 289
graphene::chain::cut_fee (C++ function), 260
graphene::chain::database (C++ class), 289
graphene::chain::database::_popped_tx (C++ member), 293
graphene::chain::database::adjust_balance (C++ function), 292
graphene::chain::database::applied_block (C++ member), 292
graphene::chain::database::apply_order (C++ function), 289
graphene::chain::database::changed_objects (C++ member), 293
graphene::chain::database::check_call_orders (C++ function), 292
graphene::chain::database::debug_dump (C++ function), 292
graphene::chain::database::deposit_lazy_vesting (C++ function), 292
graphene::chain::database::fill_order (C++ function), 292
graphene::chain::database::get_balance (C++ function), 291, 292
graphene::chain::database::get_scheduled_witness (C++ function), 291
graphene::chain::database::get_slot_at_time (C++ function), 291
graphene::chain::database::get_slot_time (C++ function), 291
graphene::chain::database::globally_settle_asset (C++ function), 289
graphene::chain::database::initialize_indexes (C++ function), 291
graphene::chain::database::is_known_block (C++ function), 290
graphene::chain::database::is_known_transaction (C++ function), 290
graphene::chain::database::match (C++ function), 289, 290
graphene::chain::database::new_objects (C++ member), 293
graphene::chain::database::on_pending_transaction (C++ member), 292
graphene::chain::database::open (C++ function), 290
graphene::chain::database::pop_block (C++ function), 291
graphene::chain::database::push_applied_operation (C++ member), 291
graphene::chain::database::push_block (C++ function), 290
graphene::chain::database::push_proposal (C++ function), 291
graphene::chain::database::push_transaction (C++ function), 290
graphene::chain::database::reindex (C++ function), 290
graphene::chain::database::removed_objects (C++ member), 293
graphene::chain::database::validate_transaction (C++ function), 292
graphene::chain::database::wipe (C++ function), 290
graphene::chain::database::witness_participation_rate (C++ function), 290
graphene::chain::debug_apply_update (C++ function), 261
graphene::chain::deprecate_annual_members (C++ function), 261
graphene::chain::detail (C++ type), 316
graphene::chain::detail::is_authorized_asset (C++ function), 317
graphene::chain::detail::for_each (C++ function), 316
graphene::chain::detail::gen_seq<0, Is...> (C++ class), 317
graphene::chain::detail::pending_transactions_restorer (C++ class), 317
graphene::chain::detail::skip_flags_restorer (C++ class), 317
graphene::chain::detail::with_skip_flags (C++ function), 316
graphene::chain::detail::without_pending_transactions (C++ function), 316
graphene::chain::digest_type (C++ type), 257
graphene::chain::disable_confidential (C++ class), 206, 208, 259
graphene::chain::disable_force_settle (C++ class), 206, 208, 259

graphene::chain::distribute_fba_balances (C++ function), 261
graphene::chain::dynamic_global_property_id_type (C++ type), 257
graphene::chain::dynamic_global_property_object (C++ class), 293
graphene::chain::dynamic_global_property_object::current_gsl (C++ member), 293
graphene::chain::dynamic_global_property_object::dynamic_gsl (C++ member), 293
graphene::chain::dynamic_global_property_object::recent_gsl (C++ member), 293
graphene::chain::dynamic_global_property_object::recently_gsl (C++ member), 293
graphene::chain::evaluate_buyback_account_options (C++ function), 260
graphene::chain::evaluate_special_authority (C++ function), 263
graphene::chain::evaluator (C++ class), 293
graphene::chain::evaluator::evaluate (C++ function), 293
graphene::chain::extensions_type (C++ type), 256
graphene::chain::fba_accumulator_id_blind_transfer (C++ class), 259
graphene::chain::fba_accumulator_id_count (C++ class), 259
graphene::chain::fba_accumulator_id_transfer_from_blind (C++ class), 259
graphene::chain::fba_accumulator_id_transfer_to_blind (C++ class), 258
graphene::chain::fba_accumulator_id_type (C++ type), 257
graphene::chain::fba_accumulator_object (C++ class), 294
graphene::chain::fba_distribute_operation (C++ class), 294
graphene::chain::fee_parameters (C++ type), 256
graphene::chain::fee_schedule (C++ class), 294
graphene::chain::fee_schedule::calculate_fee (C++ function), 294
graphene::chain::fee_schedule::parameters (C++ member), 294
graphene::chain::fee_schedule::scale (C++ member), 294
graphene::chain::fee_schedule::validate (C++ function), 294
graphene::chain::fee_schedule_type (C++ type), 256
graphene::chain::fill_order_operation (C++ class), 294
graphene::chain::fill_order_operation::calculate_fee (C++ function), 294
graphene::chain::force_settlement_id_type (C++ type), 257
graphene::chain::force_settlement_index (C++ type), 256
graphene::chain::force_settlement_object (C++ class), 294
graphene::chain::force_settlement_object_multi_index_type (C++ type), 256
graphene::chain::force_settlement_object_type (C++ class), 165, 259
graphene::chain::fork_database (C++ class), 294
graphene::chain::fork_database::fetch_branch_from (C++ function), 295
graphene::chain::fork_database::MAX_BLOCK_REORDERING (C++ member), 295
graphene::chain::fork_database::push_block (C++ function), 295
graphene::chain::fork_item (C++ class), 295
graphene::chain::fork_item::invalid (C++ member), 295
graphene::chain::future_extensions (C++ type), 256
graphene::chain::generic_evaluator (C++ class), 295
graphene::chain::generic_evaluator::evaluate (C++ function), 296
graphene::chain::generic_evaluator::pay_fee (C++ function), 296
graphene::chain::genesis_state_type (C++ class), 296
graphene::chain::genesis_state_type::compute_chain_id (C++ function), 296
graphene::chain::genesis_state_type::initial_chain_id (C++ member), 296
graphene::chain::genesis_state_type::initial_committee_member_type (C++ class), 296
graphene::chain::genesis_state_type::initial_committee_member_type::own (C++ member), 297
graphene::chain::genesis_state_type::initial_witness_type (C++ class), 297
graphene::chain::genesis_state_type::initial_witness_type::owner_name (C++ member), 297
graphene::chain::genesis_state_type::initial_worker_type (C++ class), 297
graphene::chain::genesis_state_type::initial_worker_type::owner_name (C++ member), 297
graphene::chain::get_config (C++ function), 261
graphene::chain::get_next_vote_id (C++ function), 263
graphene::chain::global_property_id_type (C++ type), 257
graphene::chain::global_property_object (C++ class), 297
graphene::chain::global_settle (C++ class), 206, 208, 259
graphene::chain::GRAPHENE_DECLARE_OP_BASE_EXCEPTIONS (C++ function), 261
graphene::chain::graphene_fba_accumulator_id_enum (C++ type), 258
graphene::chain::impl_account_balance_object_type (C++ class), 165, 260
graphene::chain::impl_account_statistics_object_type (C++ class), 165, 260
graphene::chain::impl_account_transaction_history_object_type (C++ class), 165, 260
graphene::chain::impl_asset_bitasset_data_type (C++ class), 165, 260

graphene::chain::impl_asset_dynamic_data_type (C++ class), 165, 260
graphene::chain::impl_blinded_balance_object_type (C++ class), 166, 260
graphene::chain::impl_block_summary_object_type (C++ class), 165, 260
graphene::chain::impl_budget_record_object_type (C++ class), 166, 260
graphene::chain::impl_buyback_object_type (C++ class), 166, 260
graphene::chain::impl_chain_property_object_type (C++ class), 166, 260
graphene::chain::impl_dynamic_global_property_object_type (C++ class), 165, 260
graphene::chain::impl_fba_accumulator_object_type (C++ class), 166, 260
graphene::chain::impl_global_property_object_type (C++ class), 165, 260
graphene::chain::impl_object_type (C++ type), 165, 260
graphene::chain::impl_reserved0_object_type (C++ class), 165, 260
graphene::chain::impl_special_authority_object_type (C++ class), 166, 260
graphene::chain::impl_transaction_object_type (C++ class), 165, 260
graphene::chain::impl_witness_schedule_object_type (C++ class), 166, 260
graphene::chain::implementation_ids (C++ class), 259
graphene::chain::int128_t (C++ type), 258
graphene::chain::is_authorized_asset (C++ function), 261
graphene::chain::is_cheap_name (C++ function), 262
graphene::chain::is_relative (C++ function), 263
graphene::chain::is_valid_name (C++ function), 261
graphene::chain::is_valid_symbol (C++ function), 262
graphene::chain::item_ptr (C++ type), 256
graphene::chain::limit_order_cancel_evaluator (C++ class), 297
graphene::chain::limit_order_cancel_operation (C++ class), 297
graphene::chain::limit_order_cancel_operation::fee_paying (C++ member), 297
graphene::chain::limit_order_create_evaluator (C++ class), 297
graphene::chain::limit_order_create_evaluator::pay_fee (C++ function), 298
graphene::chain::limit_order_create_operation (C++ class), 298
graphene::chain::limit_order_create_operation::expiration (C++ member), 298
graphene::chain::limit_order_create_operation::fill_or_kill (C++ member), 298
graphene::chain::limit_order_id_type (C++ type), 257
graphene::chain::limit_order_index (C++ type), 256
graphene::chain::limit_order_multi_index_type (C++ type), 256
graphene::chain::limit_order_object (C++ class), 298
graphene::chain::limit_order_object::for_sale (C++ member), 298
graphene::chain::limit_order_object_type (C++ class), 165, 259
graphene::chain::linear_vesting_policy (C++ class), 298
graphene::chain::linear_vesting_policy::begin_balance (C++ member), 299
graphene::chain::linear_vesting_policy::begin_timestamp (C++ member), 298
graphene::chain::linear_vesting_policy::vesting_cliff_seconds (C++ member), 298
graphene::chain::linear_vesting_policy::vesting_duration_seconds (C++ member), 299
graphene::chain::linear_vesting_policy_initializer (C++ class), 299
graphene::chain::linear_vesting_policy_initializer::begin_timestamp (C++ member), 299
graphene::chain::maybe_cull_small_order (C++ function), 261
graphene::chain::memo_data (C++ class), 299
graphene::chain::memo_data::message (C++ member), 299
graphene::chain::memo_data::nonce (C++ member), 299
graphene::chain::memo_data::set_message (C++ function), 299
graphene::chain::memo_message (C++ class), 299
graphene::chain::node_property_object (C++ class), 299
graphene::chain::null_object_type (C++ class), 165, 259
graphene::chain::object_type (C++ type), 165, 259
graphene::chain::OBJECT_TYPE_COUNT (C++ class), 165, 260
graphene::chain::op_evaluator (C++ class), 300
graphene::chain::op_evaluator_impl (C++ class), 300
graphene::chain::op_wrapper (C++ class), 300
graphene::chain::operation (C++ type), 101, 256
graphene::chain::operation_get_requiredAuthorities (C++ function), 262
graphene::chain::operation_history_id_type (C++ type), 257
graphene::chain::operation_history_object (C++ class), 300
graphene::chain::operation_history_object::block_num (C++ member), 300
graphene::chain::operation_history_object::op_in_trx (C++ member), 300
graphene::chain::operation_history_object::trx_in_block (C++ member), 300
graphene::chain::operation_history_object::virtual_op (C++ member), 300
graphene::chain::operation_history_object_type (C++ class), 165, 260
graphene::chain::operation_result (C++ type), 256

graphene::chain::operation_validate (C++ function), 263
graphene::chain::operation_validator (C++ class), 300
graphene::chain::operator = (C++ function), 262, 263
graphene::chain::operator* (C++ function), 262
graphene::chain::operator/ (C++ function), 262
graphene::chain::operator== (C++ function), 262, 263
graphene::chain::operator~ (C++ function), 262
graphene::chain::operator> (C++ function), 262
graphene::chain::operator>= (C++ function), 262
graphene::chain::operator>> (C++ function), 262
graphene::chain::operator< (C++ function), 262, 263
graphene::chain::operator<= (C++ function), 262
graphene::chain::operator<< (C++ function), 262
graphene::chain::override_authority (C++ class), 206, 208, 259
graphene::chain::override_transfer_evaluator (C++ class), 300
graphene::chain::override_transfer_operation (C++ class), 300
graphene::chain::override_transfer_operation::amount (C++ member), 301
graphene::chain::override_transfer_operation::from (C++ member), 301
graphene::chain::override_transfer_operation::memo (C++ member), 301
graphene::chain::override_transfer_operation::to (C++ member), 301
graphene::chain::parameter_extension (C++ type), 256
graphene::chain::predicate (C++ type), 256
graphene::chain::price (C++ class), 301
graphene::chain::price::call_price (C++ function), 301
graphene::chain::price::unit_price (C++ function), 301
graphene::chain::price_feed (C++ class), 301
graphene::chain::price_feed::core_exchange_rate (C++ member), 302
graphene::chain::price_feed::maintenance_collateral_ratio (C++ member), 302
graphene::chain::price_feed::max_short_squeeze_price (C++ function), 302
graphene::chain::price_feed::maximum_short_squeeze_ratio (C++ member), 302
graphene::chain::price_feed::settlement_price (C++ member), 302
graphene::chain::private_key_type (C++ type), 256
graphene::chain::processed_transaction (C++ class), 302
graphene::chain::proposal_create_evaluator (C++ class), 302
graphene::chain::proposal_create_operation (C++ class), 302
graphene::chain::proposal_create_operation::committee_program (C++ function), 303
graphene::chain::proposal_delete_evaluator (C++ class), 303
graphene::chain::proposal_delete_operation (C++ class), 303
graphene::chain::proposal_id_type (C++ type), 257
graphene::chain::proposal_index (C++ type), 256
graphene::chain::proposal_multi_index_container (C++ type), 256
graphene::chain::proposal_object (C++ class), 303
graphene::chain::proposal_object_type (C++ class), 165, 260
graphene::chain::proposal_update_evaluator (C++ class), 303
graphene::chain::proposal_update_operation (C++ class), 303
graphene::chain::protocol_ids (C++ class), 259
graphene::chain::pts_address (C++ class), 303
graphene::chain::pts_address::addr (C++ member), 304
graphene::chain::pts_address::is_valid (C++ function), 304
graphene::chain::pts_address::operator std::string (C++ function), 304
graphene::chain::pts_address::pts_address (C++ function), 304
graphene::chain::refund_worker_type (C++ class), 304
graphene::chain::refund_worker_type::total_burned (C++ member), 304
graphene::chain::relative_protocol_ids (C++ class), 259
graphene::chain::required_approval_index (C++ class), 304
graphene::chain::reserved_spaces (C++ type), 259
graphene::chain::scaled_precision_lut (C++ member), 264
graphene::chain::share_type (C++ type), 258
graphene::chain::sign_state (C++ class), 304
graphene::chain::sign_state::check_authority (C++ function), 304
graphene::chain::sign_state::signed_by (C++ function), 304
graphene::chain::signature_type (C++ type), 258
graphene::chain::signed_block (C++ class), 305
graphene::chain::signed_block_header (C++ class), 305
graphene::chain::signed_transaction (C++ class), 305
graphene::chain::signed_transaction::clear (C++ function), 305
graphene::chain::signed_transaction::get_required_signatures (C++ function), 305
graphene::chain::signed_transaction::minimize_required_signatures (C++ function), 305
graphene::chain::signed_transaction::sign (C++ function), 305
graphene::chain::smart_fee_schedule (C++ type), 258
graphene::chain::special_authority (C++ type), 256
graphene::chain::special_authority_id_type (C++ type), 257
graphene::chain::special_authority_index (C++ type),

258
graphene::chain::special_authority_multi_index_type
(C++ type), 258
graphene::chain::special_authority_object (C++ class),
305
graphene::chain::split_fba_balance (C++ function), 261
graphene::chain::stealth_confirmation (C++ class), 306
graphene::chain::stealth_confirmation::operator string
(C++ function), 306
graphene::chain::stealth_confirmation::stealth_confirmation
(C++ function), 306
graphene::chain::sum_below_max_shares (C++ function),
263
graphene::chain::symbol_type (C++ type), 257
graphene::chain::tmp (C++ member), 264
graphene::chain::transaction (C++ class), 306
graphene::chain::transaction::digest (C++ function), 306
graphene::chain::transaction::expiration (C++ member),
306
graphene::chain::transaction::ref_block_num (C++ member),
306
graphene::chain::transaction::ref_block_prefix (C++ member),
306
graphene::chain::transaction::sig_digest (C++ function),
306
graphene::chain::transaction::visit (C++ function), 306
graphene::chain::transaction_evaluation_state (C++ class),
306
graphene::chain::transaction_id_type (C++ type), 257
graphene::chain::transaction_index (C++ type), 258
graphene::chain::transaction_multi_index_type (C++ type),
258
graphene::chain::transaction_obj_id_type (C++ type),
257
graphene::chain::transaction_object (C++ class), 306
graphene::chain::transfer_evaluator (C++ class), 307
graphene::chain::transfer_from_blind_evaluator (C++ class),
307
graphene::chain::transfer_from_blind_evaluator::pay_fee
(C++ function), 307
graphene::chain::transfer_from_blind_operation (C++ class),
307
graphene::chain::transfer_from_blind_operation::fee_param
(C++ class), 307
graphene::chain::transfer_from_blind_operation::fee_param
(C++ member), 307
graphene::chain::transfer_operation (C++ class), 307
graphene::chain::transfer_operation::amount (C++ member),
308
graphene::chain::transfer_operation::from (C++ member),
308
graphene::chain::transfer_operation::memo (C++ member),
308
graphene::chain::transfer_operation::to (C++ member),
308
graphene::chain::transfer_restricted (C++ class), 206,
208, 259
graphene::chain::transfer_to_blind_evaluator (C++ class),
308
graphene::chain::transfer_to_blind_evaluator::pay_fee
(C++ function), 308
graphene::chain::transfer_to_blind_operation (C++ class),
308
graphene::chain::transfer_to_blind_operation::fee_parameters_type
(C++ class), 308
graphene::chain::transfer_to_blind_operation::fee_parameters_type::fee
(C++ member), 308
graphene::chain::UIA_ASSET_ISSUER_PERMISSION_MASK
(C++ member), 264
graphene::chain::uint128_t (C++ type), 258
graphene::chain::update_top_nAuthorities (C++ function), 261
graphene::chain::validate_special_authority (C++ function),
263
graphene::chain::verify_account_votes (C++ function),
260
graphene::chain::verify_authority (C++ function), 263
graphene::chain::verify_authority_accounts (C++ function),
260
graphene::chain::vesting_balance_create_evaluator (C++ class),
308
graphene::chain::vesting_balance_create_operation (C++ class),
308
graphene::chain::vesting_balance_create_operation::creator
(C++ member), 309
graphene::chain::vesting_balance_create_operation::owner
(C++ member), 309
graphene::chain::vesting_balance_id_type (C++ type),
257
graphene::chain::vesting_balance_index (C++ type), 258
graphene::chain::vesting_balance_multi_index_type
(C++ type), 258
graphene::chain::vesting_balance_object (C++ class),
309
graphene::chain::vesting_balance_object::balance (C++ member),
309
graphene::chain::vesting_balance_object::deposit (C++ function),
309
graphene::chain::vesting_balance_object::deposit_vested
(C++ function), 309
graphene::chain::vesting_balance_object::get_allowed_withdraw
(C++ function), 309
graphene::chain::vesting_balance_object::owner (C++ member),
309
graphene::chain::vesting_balance_object::policy (C++ member),
309
graphene::chain::vesting_balance_object::withdraw (C++ function),
309

graphene::chain::vesting_balance_object_type (C++ class), 165, 260

graphene::chain::vesting_balance_withdraw_evaluator (C++ class), 310

graphene::chain::vesting_balance_withdraw_operation (C++ class), 310

graphene::chain::vesting_balance_withdraw_operation::owner (C++ member), 310

graphene::chain::vesting_balance_worker_type (C++ class), 310

graphene::chain::vesting_balance_worker_type::balance (C++ member), 310

graphene::chain::vesting_policy (C++ type), 258

graphene::chain::vesting_policy_initializer (C++ type), 258

graphene::chain::VESTING_VISITOR (C++ function), 263

graphene::chain::visit_special_authorities (C++ function), 261

graphene::chain::vote_counter (C++ class), 310

graphene::chain::vote_counter::finish (C++ function), 310

graphene::chain::vote_id_type (C++ class), 310

graphene::chain::vote_id_type::content (C++ member), 311

graphene::chain::vote_id_type::instance (C++ function), 311

graphene::chain::vote_id_type::operator std::string (C++ function), 311

graphene::chain::vote_id_type::operator uint32_t (C++ function), 311

graphene::chain::vote_id_type::operator= (C++ function), 311

graphene::chain::vote_id_type::set_instance (C++ function), 311

graphene::chain::vote_id_type::set_type (C++ function), 311

graphene::chain::vote_id_type::type (C++ function), 311

graphene::chain::vote_id_type::vote_id_type (C++ function), 311

graphene::chain::weight_type (C++ type), 258

graphene::chain::white_list (C++ class), 206, 208, 259

graphene::chain::withdraw_permission_claim_evaluator (C++ class), 311

graphene::chain::withdraw_permission_claim_operation (C++ class), 311

graphene::chain::withdraw_permission_claim_operation::account (C++ member), 312

graphene::chain::withdraw_permission_claim_operation::fee (C++ member), 312

graphene::chain::withdraw_permission_claim_operation::memo (C++ member), 312

graphene::chain::withdraw_permission_claim_operation::withdrawn_from_main_account (C++ member), 312

graphene::chain::withdraw_permission_claim_operation::withdrawn_permission_object_multi_index_type (C++ type), 258

graphene::chain::withdraw_permission_claim_operation::withdrawn_permission_object::withdrawing (C++ member), 312

graphene::chain::withdraw_permission_create_evaluator (C++ class), 312

graphene::chain::withdraw_permission_create_operation (C++ class), 312

graphene::chain::withdraw_permission_create_operation::authorized_account (C++ member), 312

graphene::chain::withdraw_permission_create_operation::period_start_time (C++ member), 313

graphene::chain::withdraw_permission_create_operation::periods_until_expiry (C++ member), 313

graphene::chain::withdraw_permission_create_operation::withdraw_from_account (C++ member), 312

graphene::chain::withdraw_permission_create_operation::withdrawal_limit (C++ member), 313

graphene::chain::withdraw_permission_create_operation::withdrawal_period (C++ member), 313

graphene::chain::withdraw_permission_delete_evaluator (C++ class), 313

graphene::chain::withdraw_permission_delete_operation (C++ class), 313

graphene::chain::withdraw_permission_delete_operation::authorized_account (C++ member), 313

graphene::chain::withdraw_permission_delete_operation::withdraw_from_account (C++ member), 313

graphene::chain::withdraw_permission_delete_operation::withdrawal_permission (C++ member), 313

graphene::chain::withdraw_permission_id_type (C++ type), 257

graphene::chain::withdraw_permission_index (C++ type), 258

graphene::chain::withdraw_permission_object (C++ class), 313

graphene::chain::withdraw_permission_object::authorized_account (C++ member), 314

graphene::chain::withdraw_permission_object::available_this_period (C++ function), 313

graphene::chain::withdraw_permission_object::claimed_this_period (C++ member), 314

graphene::chain::withdraw_permission_object::expiration (C++ member), 314

graphene::chain::withdraw_permission_object::period_start_time (C++ member), 314

graphene::chain::withdraw_permission_object::withdraw_from_account (C++ member), 314

graphene::chain::withdraw_permission_object::withdrawal_limit (C++ member), 314

graphene::chain::withdraw_permission_object::withdrawal_period_sec (C++ member), 314

graphene::chain::withdraw_permission_object::withdrawn_from_main_account (C++ type), 258

graphene::chain::withdraw_permission_object_type
(C++ class), 165, 260

graphene::chain::withdraw_permission_update_evaluator
(C++ class), 314

graphene::chain::withdraw_permission_update_operation
(C++ class), 314

graphene::chain::withdraw_permission_update_operation::authorized_count
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::graphene_start_time::worker_object::vote_against
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::graphene_end_time::worker_object::vote_for
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::graphene_chain::worker_object::work_begin_date
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::graphene_chain::worker_object::work_end_date
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::withdrawn_chair::worker::worker
(C++ member), 314

graphene::chain::withdraw_permission_update_operation::withdrawn_chair::worker::worker_account
(C++ member), 314

graphene::chain::witness_create_evaluator (C++ class), 315

graphene::chain::witness_create_operation (C++ class), 315

graphene::chain::witness_create_operation::witness_account
(C++ member), 315

graphene::chain::witness_fed_asset (C++ class), 206, 208, 259

graphene::chain::witness_id_type (C++ type), 257

graphene::chain::witness_object (C++ class), 315

graphene::chain::witness_object_type (C++ class), 165, 259

graphene::chain::witness_schedule_id_type (C++ type), 257

graphene::chain::witness_schedule_object (C++ class), 315

graphene::chain::witness_update_evaluator (C++ class), 315

graphene::chain::witness_update_operation (C++ class), 315

graphene::chain::witness_update_operation::new_signing_key
(C++ member), 315

graphene::chain::witness_update_operation::new_url
(C++ member), 315

graphene::chain::witness_update_operation::witness
(C++ member), 315

graphene::chain::witness_update_operation::witness_account
(C++ member), 315

graphene::chain::worker_create_evaluator (C++ class), 315

graphene::chain::worker_create_operation (C++ class), 315

graphene::chain::worker_create_operation::initializer
(C++ member), 316

graphene::chain::worker_id_type (C++ type), 257

graphene::chain::worker_initializer (C++ type), 258

graphene::chain::worker_object (C++ class), 316

graphene::chain::worker_object::daily_pay (C++ member), 316

graphene::chain::worker_object::name (C++ member), 316

graphene::chain::worker_object::url (C++ member), 316

graphene::chain::worker_object::vote_against
(C++ member), 316

graphene::chain::worker_object::vote_for (C++ member), 316

graphene::chain::worker_object::work_begin_date (C++ member), 316

graphene::chain::worker_object::work_end_date (C++ member), 316

graphene::chain::worker_object::worker (C++ member), 316

graphene::chain::worker::worker_account (C++ member), 316

graphene::chain::worker_type (C++ type), 258

graphene::wallet (C++ type), 317

graphene::wallet::blind_balance (C++ class), 317

graphene::wallet::blind_balance::from (C++ member), 317

graphene::wallet::blind_balance::one_time_key (C++ member), 318

graphene::wallet::blind_balance::to (C++ member), 317

graphene::wallet::blind_confirmation (C++ class), 318

graphene::wallet::blind_receipt_index_type (C++ type), 317

graphene::wallet::create_object (C++ function), 317

graphene::wallet::create_static_variant_map (C++ function), 317

graphene::wallet::detail (C++ type), 336

graphene::wallet::detail::address_to_shorthash (C++ function), 336

graphene::wallet::detail::derive_private_key (C++ function), 336

graphene::wallet::detail::maybe_id (C++ function), 336

graphene::wallet::detail::normalize_brain_key (C++ function), 336

graphene::wallet::from_which_variant (C++ function), 317

graphene::wallet::impl (C++ type), 336

graphene::wallet::impl::clean_name (C++ function), 336

graphene::wallet::key_label_index_type (C++ type), 317

graphene::wallet::signed_block_with_info (C++ class), 318

graphene::wallet::transaction_handle_type (C++ type), 317
graphene::wallet::utility (C++ class), 318
graphene::wallet::utility::derive_owner_keys_from_brain_kg (C++ function), 318
graphene::wallet::vesting_balance_object_with_info (C++ class), 318
graphene::wallet::vesting_balance_object_with_info::allowwithdraw (C++ member), 318
graphene::wallet::vesting_balance_object_with_info::allowwithdraw (C++ member), 318
graphene::wallet::wallet_api (C++ class), 318
graphene::wallet::wallet_api::about (C++ function), 167, 319
graphene::wallet::wallet_api::add_operation_to_builder_transaction (C++ function), 188
graphene::wallet::wallet_api::approve_proposal (C++ function), 104, 173, 335
graphene::wallet::wallet_api::begin_builder_transaction (C++ function), 188
graphene::wallet::wallet_api::blind_history (C++ function), 187, 325
graphene::wallet::wallet_api::blind_transfer (C++ function), 187, 325
graphene::wallet::wallet_api::blind_transfer_help (C++ function), 335
graphene::wallet::wallet_api::borrow_asset (C++ function), 174, 327
graphene::wallet::wallet_api::buy (C++ function), 326
graphene::wallet::wallet_api::cancel_order (C++ function), 175, 327
graphene::wallet::wallet_api::create_account_with_brain_kg (C++ function), 170, 324
graphene::wallet::wallet_api::create_asset (C++ function), 176, 327
graphene::wallet::wallet_api::create_blind_account (C++ function), 186, 319
graphene::wallet::wallet_api::create_committee_member (C++ function), 180, 330
graphene::wallet::wallet_api::create_witness (C++ function), 182, 331
graphene::wallet::wallet_api::create_worker (C++ function), 182, 332
graphene::wallet::wallet_api::derive_owner_keys_from_brain_kg (C++ function), 323
graphene::wallet::wallet_api::dump_private_keys (C++ function), 167, 322
graphene::wallet::wallet_api::fund_asset_fee_pool (C++ function), 179, 329
graphene::wallet::wallet_api::get_account (C++ function), 173, 320
graphene::wallet::wallet_api::get_account_count (C++ function), 187, 319
graphene::wallet::wallet_api::get_account_history (C++ function), 173, 195, 320
graphene::wallet::wallet_api::get_account_id (C++ function), 173, 321
graphene::wallet::wallet_api::get_asset (C++ function), 179, 197, 320
graphene::wallet::wallet_api::get_asset_id (C++ function), 321
graphene::wallet::wallet_api::get_bitasset_data (C++ function), 179, 320
graphene::wallet::wallet_api::get_blind_accounts (C++ function), 186, 319
graphene::wallet::wallet_api::get_blind_balances (C++ function), 186, 319
graphene::wallet::wallet_api::get_block (C++ function), 187
graphene::wallet::wallet_api::get_call_orders (C++ function), 176
graphene::wallet::wallet_api::get_committee_member (C++ function), 181, 331
graphene::wallet::wallet_api::get_dynamic_global_properties (C++ function), 188, 320
graphene::wallet::wallet_api::get_global_properties (C++ function), 188, 320
graphene::wallet::wallet_api::get_key_label (C++ function), 186
graphene::wallet::wallet_api::get_limit_orders (C++ function), 176
graphene::wallet::wallet_api::get_market_history (C++ function), 175
graphene::wallet::wallet_api::get_my_blind_accounts (C++ function), 186, 319
graphene::wallet::wallet_api::get_object (C++ function), 188, 196, 321
graphene::wallet::wallet_api::get_private_key (C++ function), 168, 321
graphene::wallet::wallet_api::get_prototype_operation (C++ function), 190, 334
graphene::wallet::wallet_api::get_public_key (C++ function), 186, 319
graphene::wallet::wallet_api::get_relative_account_history (C++ function), 320
graphene::wallet::wallet_api::get_settle_orders (C++ function), 176
graphene::wallet::wallet_api::get_transaction_id (C++ function), 168, 325
graphene::wallet::wallet_api::get_vesting_balances (C++ function), 172, 332
graphene::wallet::wallet_api::get_wallet_filename (C++ function), 321
graphene::wallet::wallet_api::get_witness (C++ function), 180, 331
graphene::wallet::wallet_api::gethelp (C++ function), 166, 322
graphene::wallet::wallet_api::global_settle_asset (C++ function), 166, 322

function), 180, 329
`graphene::wallet::wallet_api::help` (C++ function), 166, 322
`graphene::wallet::wallet_api::import_account_keys` (C++ function), 168
`graphene::wallet::wallet_api::import_accounts` (C++ function), 168
`graphene::wallet::wallet_api::import_balance` (C++ function), 168, 323
`graphene::wallet::wallet_api::import_key` (C++ function), 167, 323
`graphene::wallet::wallet_api::info` (C++ function), 167
`graphene::wallet::wallet_api::is_locked` (C++ function), 167, 321
`graphene::wallet::wallet_api::is_new` (C++ function), 167, 321
`graphene::wallet::wallet_api::is_public_key_registered` (C++ function), 323
`graphene::wallet::wallet_api::issue_asset` (C++ function), 178, 327
`graphene::wallet::wallet_api::list_account_balances` (C++ function), 169, 191, 319
`graphene::wallet::wallet_api::list_accounts` (C++ function), 169, 319
`graphene::wallet::wallet_api::list_assets` (C++ function), 176, 319
`graphene::wallet::wallet_api::list_committee_members` (C++ function), 181, 331
`graphene::wallet::wallet_api::list_my_accounts` (C++ function), 169, 319
`graphene::wallet::wallet_api::list_witnesses` (C++ function), 181, 330
`graphene::wallet::wallet_api::load_wallet_file` (C++ function), 168, 322
`graphene::wallet::wallet_api::lock` (C++ function), 167, 321
`graphene::wallet::wallet_api::network_add_nodes` (C++ function), 167
`graphene::wallet::wallet_api::network_get_connected_peers` (C++ function), 167
`graphene::wallet::wallet_api::normalize_brain_key` (C++ function), 168, 323
`graphene::wallet::wallet_api::preview_builder_transaction` (C++ function), 189
`graphene::wallet::wallet_api::propose_builder_transaction` (C++ function), 103, 189
`graphene::wallet::wallet_api::propose_builder_transaction2` (C++ function), 103, 189
`graphene::wallet::wallet_api::propose_fee_change` (C++ function), 185, 335
`graphene::wallet::wallet_api::propose_parameter_change` (C++ function), 185, 334
`graphene::wallet::wallet_api::publish_asset_feed` (C++ function), 178, 328
`graphene::wallet::wallet_api::receive_blind_transfer` (C++ function), 187, 325
`graphene::wallet::wallet_api::register_account` (C++ function), 169, 323
`graphene::wallet::wallet_api::remove_builder_transaction` (C++ function), 190
`graphene::wallet::wallet_api::replace_operation_in_builder_transaction` (C++ function), 188
`graphene::wallet::wallet_api::reserve_asset` (C++ function), 179, 329
`graphene::wallet::wallet_api::save_wallet_file` (C++ function), 168, 322
`graphene::wallet::wallet_api::sell` (C++ function), 326
`graphene::wallet::wallet_api::sell_asset` (C++ function), 174, 325
`graphene::wallet::wallet_api::serialize_transaction` (C++ function), 190, 323
`graphene::wallet::wallet_api::set_desired_witness_and_committee_member` (C++ function), 184, 333
`graphene::wallet::wallet_api::set_fees_on_builder_transaction` (C++ function), 189
`graphene::wallet::wallet_api::set_key_label` (C++ function), 186, 318
`graphene::wallet::wallet_api::set_password` (C++ function), 167, 321
`graphene::wallet::wallet_api::set_voting_proxy` (C++ function), 184, 333
`graphene::wallet::wallet_api::set_wallet_filename` (C++ function), 322
`graphene::wallet::wallet_api::settle_asset` (C++ function), 175, 330
`graphene::wallet::wallet_api::sign_builder_transaction` (C++ function), 189
`graphene::wallet::wallet_api::sign_transaction` (C++ function), 190, 334
`graphene::wallet::wallet_api::suggest_brain_key` (C++ function), 168, 322
`graphene::wallet::wallet_api::transfer` (C++ function), 171, 193, 324
`graphene::wallet::wallet_api::transfer2` (C++ function), 172, 194, 325
`graphene::wallet::wallet_api::transfer_from_blind` (C++ function), 187, 325
`graphene::wallet::wallet_api::transfer_to_blind` (C++ function), 186, 325
`graphene::wallet::wallet_api::unlock` (C++ function), 167, 321
`graphene::wallet::wallet_api::update_asset` (C++ function), 177, 204, 207, 328
`graphene::wallet::wallet_api::update_asset_feed_producers` (C++ function), 177, 328
`graphene::wallet::wallet_api::update_bitasset` (C++ function), 177, 328
`graphene::wallet::wallet_api::update_witness` (C++ func-

tion), 182, 331
graphene::wallet::wallet_api::update_worker_votes (C++ function), 183, 332
graphene::wallet::wallet_api::upgrade_account (C++ function), 170, 324
graphene::wallet::wallet_api::vote_for_committee_member (C++ function), 183, 332
graphene::wallet::wallet_api::vote_for_witness (C++ function), 183, 333
graphene::wallet::wallet_api::whitelist_account (C++ function), 172, 203, 330
graphene::wallet::wallet_api::withdraw_vesting (C++ function), 172, 332
graphene::wallet::wallet_data (C++ class), 335
graphene::wallet::wallet_data::chain_id (C++ member), 336
graphene::wallet::wallet_data::cipher_keys (C++ member), 336
graphene::wallet::wallet_data::extra_keys (C++ member), 336
graphene::wallet::wallet_data::my_account_ids (C++ function), 335
graphene::wallet::wallet_data::update_account (C++ function), 336